The Case for Less Predictable Operating System Behavior

Ruimin Sun Donald E. Porter[†] Daniela Oliveira Matt Bishop[‡]

University of Florida   Stony Brook University[†]   University of California at Davis[‡]

"No one is so brave that he is not disturbed by something unexpected." Julius Caesar

The operating system is increasingly regarded as untrustworthy. Applications, hardware, and hypervisors are erecting defenses to insulate themselves from the operating system. This paper explores the potential benefits if operating systems simply embraced these lowered expectations and deliberately varied API behavior. We argue that, even for trusted or benign applications, diversity roughly within the specification can improve resilience to attack and improve robustness. Malicious software tends to be brittle; a preliminary case study indicates that, for software of questionable origin, a somewhat hostile operating system may do more good than harm for system security. This paper describes the architecture of Chameleon, an ongoing project to implement spectrum-behavior as an operating system feature.

1 Introduction

A common goal of modern operating systems is to be predictable. This improves compatibility among different instances of the system, including older programs running on newer systems. But predictability poses problems. Predictability allows vulnerabilities that are exploitable on one system to be exploitable on all systems of that type. The paucity of different OS implementations is one facet of the software "monoculture" problem.

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. One example is N-version programming [1], in which multiple teams create different software implementations to perform the same actions. The results are considered more reliable when multiple versions produce the same results. Most fault-tolerant system designs require sufficient software diversity that faults are independent, and can be masked by voting or Byzantine protocols [2, 3].

At the system level, approaches to diversity generally involve randomness. For example, address space layout randomization (ASLR) randomizes the placement of pages of a program in memory during execution. A return-to-libc or ROP attack may fail when the attack relies on a buffer overflow causing a branch to a library function or gadget, as the address of that target will vary among instances of an operating system. But this randomization is often insufficient against knowledgeable attackers. A recent paper [4] demonstrated how, even without specific knowledge of the ASLR of a web server, one can quickly identify and exploit buffer overflows in it. The technique relied on the fact that systems are commonly configured to restart daemons such as web servers automatically, and that ASLR implementations do not re-randomize the address space after restarting. As a result, an attacker can incrementally explore the address space and probe application behavior. Although fixes to ASLR may mitigate this specific attack, the underlying lesson is that diversity without unpredictability is not enough. There is enough residual certainty that adversaries can craft attacks that will work reliably across multiple instances of a predictably diverse 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, "authorized" meaning "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 eliminates efficiency and reliability. An extension is a *spectrum* of predictability, so that the less actions conform to the security policy, the more unpredictable the results of those actions should be.

This paper explores the benefits and feasibility of making OS APIs less predictable on a spectrum from diversity within the specification to active deception of dodgy software. We argue that software robustness can actually be improved by being developed on a spectrum-behavior operating system. Even within POSIX, mature, portable software packages already handle considerable variations in system call behavior. Most of this maturity is the product of testing and bug reports across many platforms. Moreover, hardware, compiler, and hypervi-

sor tools to protect the application from a malicious OS are rapidly evolving [5–8]. Rather than require a software developer to manually test the software on multiple platforms, the development process could be facilitated by easily generating a range of different behaviors to test the software on—running the same test suite against different operating system behaviors. In essence, the operating system is a chameleon, taking on attributes appropriate to the user and use to which it is put.

Underlying this idea is the observation that systems tend to be fixed and do not adapt well to new conditions. A motivated attacker can bring great resources to find attack variations that will succeed. Despite the explosion of security software, malware remains at an average of 125 lines of code [9]. Thus, a "holy grail" of system design is the ability for the system to adapt with considerably less effort than the attacker must expend to explore the new system variants. Unpredictable behavior can be such a mechanism for active defense against an attacker.

Section 2 examines deception and diversity as mechanisms for introducing unpredictability into OSes. Section 3 presents preliminary results that indicate varying OS behavior affects malware, which loses data and functionality. Section 4 describes the design of **Chameleon**, a system that combines inconsistent and consistent deception with software diversity to provide a mechanism for active defense of computer systems and herd protection. Chameleon leverages recent work that pushes an increasing portion of system code to user level [8, 10–18] as a means to more quickly and easily mix-and-match system behavior transparently to the application.

2 Truths About Deception

This section summarizes how deception and diversity have been used previously in software design, and highlights under-studied areas.

2.1 Diversity

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.

One approach to diversifying software is at compile time. Several projects [19, 20] randomize 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 [21–25]. Holland *et al.* [26] 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 [27] specialized code dynamically using automatic compiler analysis and programmer annotations, primarily to improve performance; specialization has been proposed as a mechanism to block attacks [28]. Program slicing has also been used to bound the cost and complexity of automatic diversification [29]. Finally, several projects have combined existing diverse implementations of file systems [30], databases [31], and language implementations [32]. As discussed above, dynamic diversity reduces predictability but is often limited to easily-randomized features of software.

Although the focus of this work is not on diversity, we observe that much of the needed infrastructure for both diversity and deception are already being developed for other purposes. Recent library OS designs [8, 10–13, 16, 18] high-performance I/O systems [15–17], and other hardware access techniques [14] facilitate migration of kernel APIs into the application itself, in some cases implemented in higher level languages [13]. With some disciplined modularization of library OS subsystems, the otherwise daunting task of multi-version programming can be made feasible—a few hundred or thousand lines per component, possibly in different languages. 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 of instances will share the same combinations of vulnerabilities. When the implementation effort is smaller and well-defined, a single graduate operating system course could easily generate dozens of functional implementations of each subsystem.

2.2 Deception

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 deception as a way to confuse and stall the enemy, sap their morale, and decrease their maneuverability [33–38].

To a limited extent, deception has been an implicit technique for cyber warfare and defense. The best known example is Cliff Stoll's use of deception to keep an intruder on an international telephone line for several hours, downloading a bogus but interesting file [39]. The

authorities were able to trace the call, and broke up a spy ring. Cheswick's response to Berferd is another classic in this area [40], and foreshadowed much of the honeypot work [41, 42]. Zhao and Mannan [43] employed deception in system authentication by giving adversaries access to fake accounts in cases of password brute force attacks. Sandboxes and virtual machines limit the actions of the attackers while giving the appearance of unfettered access to resources.

Consistent deception strategies make the deceiver's system appear as indistinguishable as possible from another, real system. The attacker does not perceive the deception and believes in a consistent false reality. Stoll's actions were designed to make the attacker think he had found a system with classified documents on it. Cheswick created a falsity of a system that was old, slow, and vulnerable. Honeypots, honeynets, sandboxes, and virtual machines are designed to exhibit behavior consistent with production systems.

Several technologies for providing deception have been studied. Software decoys are agents that protect objects from unauthorized access [44–48] by creating 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. The work used consistent deception.

Red-teaming experiments at Sandia tested the effectiveness of network deception on attackers working in groups [49]. The network-level deception delayed attackers for a few hours, wearing down some groups to abandon the attack before the end of the experiments.

Deception at the host level modifies system behavior when an attacker is logged in. One implementation uses a wrapper that intercepts program execution requests and optionally runs a different program without the user detecting the switch [50]. But many command interpreters perform some of the requested actions directly, without invoking system calls and so bypassing the wrapper.

Almeshekah and Spafford [51] further investigated the adversaries' biases and proposed a model to integrate deception-based mechanisms in computer systems. Chameleon will extend this model, and investigate the utility of inconsistent deception.

In all these cases, the fictional systems are predictable to some degree; they act as would real systems given the attacker's inputs. Other inputs (such as hardware failures) introduce a degree of unpredictability with respect to the availability of the system, but do not affect the attacker's steps to compromise the system.

True unpredictability requires randomness at a level that would cause the attacker to get inconsistent results, or *inconsistent deception* [52]. Neagoe and Bishop argued that an attacker will have no idea of whether she is exposed under a deception or a normal system that is malfunctioning, but will feel disoriented and may withdraw from the situation. In this paper we present preliminary results from running keyloggers and botnets under inconsistent deception. For instance, a keylogger in the inconsistent deceptive environment loses some keystrokes and records some false keystrokes.

Iago attacks [53] are a good example of how such deception might work. 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 policies. We believe similar techniques can be employed for active system defense. Several papers have demonstrated that abnormal sequences of system calls or even function calls can indicate the presence of malware [54–56]. When malware is suspected, we propose to use Iago attack-style deception as active defense. Somayaji and Forrest [57] proposed a model of delaying system execution as one means of active defense. The Chameleon project aims to build a more powerful spectrum-behavior OS.

3 Malware Case Study

In this section, we show that common malware can be quite sensitive to relatively minor misbehavior by the operating system. In this case study, many, these errors are often within the specification of the network or potential storage failure modes; a robust application would detect most issues with end-to-end checks [58] such as checksumming files, or, in other cases, checks designed to shield against a malicious OS, such as MAC checks on an encrypted socket. A few cases, such as injected keystrokes, would be hard for any robust application to detect on current software stacks.

Our preliminary study uses `ptrace` to interpose on system calls invoked by a keylogger and a botnet, introducing unpredictable behavior into their execution. In these cases, the malware runs without crashing, but some I/O is corrupted.

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 [59] to 86 malware samples for Linux, including 17 backdoors, 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. The most commonly invoked by malware include `write()`, `wait()`, `clone()`, `close()`, `read()`, `open()`, `send()` and `fstat()`.

In selecting strategies for spectrum behavior, our aim is to perturb system calls that harm malware yet al-

low benign code to run. We found that the following system calls are critical to process start-up and execution, and cannot be easily varied: `fstat()`, `getuid()`, `ioperm()`, `set_thread_area()`, and `mprotect()`. In other cases, perturbing system call parameters leads 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 to fail sending an e-mail, 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 these 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.

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 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, for example 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) [60], 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 [61] 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 [62] 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.

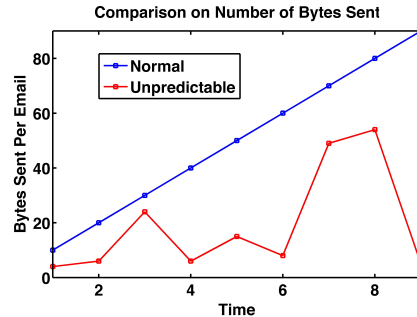We measured the impact of the unpredictable environ-

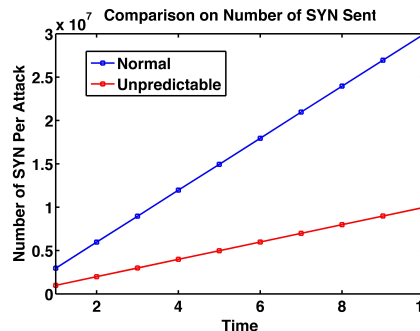Figure 1: Comparison of email bytes sent from bots in predictable and unpredictable environments.

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

ment 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 functionalities temporarily unavailable.

4 Spectrum-Behavior OS

Chameleon combines inconsistent and consistent deception with software diversity for active defense of com-
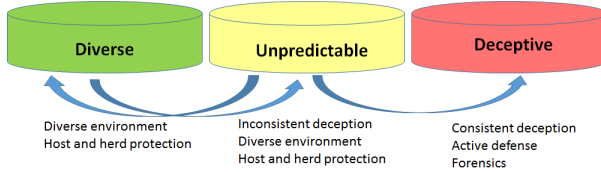
Figure 3: Chameleon can transition processes among three operating modes: *Diverse*, to protect benign software; *Unpredictable*, to disturb unknown software; and *Deceptive*, to analyze likely malware.

puter systems and herd protection. It provides three distinct environments for process execution (Figure 3): (i) a diverse environment for whitelisted processes, (ii) an unpredictable environment for unknown or suspicious processes (inconsistent deception), and (iii) a consistently deceptive environment for malicious processes. The Chameleon prototype is ongoing work.

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 [1], except that Chameleon doesn't run the versions in parallel, but rather diversifies 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 [11] 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 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 [63]. Shadow Honeypots [64] 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 honey-files, 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.

5 Conclusions

We currently 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 [5–8]. If applications code will trust the operating system less in the future, why not leverage this as a way to make malware and attacks harder to write?

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. Common, whitelisted applications, such as office software, run unperturbed with less risk of exploitation. If successful, sacrificing predictable behavior can finally give systems an edge over one of the primary sources of computer compromises [65]: malware installed by unwitting users.

Acknowledgments

References

[1] 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*, pp. 3–9, 1978.

[2] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, pp. 398–461, Nov. 2002.

[3] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché, "Upright cluster services," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 277–290, 2009.

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

[5] 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.

[6] 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)*, pp. 265–278, 2013.

[7] 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)*, pp. 81–96, 2014.

[8] 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)*, pp. 267–283, 2014.

[9] "Darpa's framework for the cyber security challenge https://www.youtube.com/watch?v=EgR44QXQLns."

[10] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt, "Rethinking the library OS from the top down," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 291–304, 2011.

[11] 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)*, pp. 9:1–9:14, 2014.

[12] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt, "Composing OS extensions safely and efficiently with Bascule," in *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[13] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged cpu features," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335–348, 2012.

[15] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 49–65, 2014.

[16] D. Schatzberg, J. Cadden, O. Krieger, and J. Appavoo, "MultiLibOS: An OS architecture for cloud computing," tech. rep., Boston University, 2014.

[17] 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)*, pp. 1–16, Oct. 2014.

[18] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, "Libra: A library operating system for a JVM in a virtualized execution environment," in *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pp. 44–54, 2007.

[19] 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.

[20] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 276–291, 2014.

[21] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," tech. rep., University of California, Berkeley, 2002.

[22] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *PLDI*, pp. 158–168, June 2006.

[23] 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.

[24] 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 *Proceedings of the USENIX Security Symposium*, 2006.

[25] 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.

[26] D. A. Holland, A. T. Lim, and M. I. Seltzer, "An architecture a day keeps the hacker away," *SIGARCH Comput. Archit. News*, vol. 33, pp. 34–41, Mar. 2005.

[27] 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, pp. 217–251, May 2001.

[28] C. Pu, A. P. Black, C. Cowan, J. Walpole, and C. Consel, "A specialization toolkit to increase the diversity of operating systems," in *Proceedings of the ICMAS Workshop on Immunity-Based Systems*, 1996.

[29] 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*, pp. 1–7, IEEE, 2013.

[30] 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*, pp. 7–7, 2009.

[31] 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)*, pp. 59–72, ACM, 2007.

[32] 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)*, pp. 1–3, 2010.

[33] R. Greene, *The 33 Strategies of War*. Viking Adult, 2006.

[34] S. Tzu, *The Art of War*. Filiquarian, 2007.

[35] A. Goldsworthy, *Caesar: Life of a Colossus*. Yale University Press, 2006.

[36] C. von Causewitz, *On War*. Princeton University Press, 2008.

[37] A. Roberts, *Napoleon, A Life*. Viking Adult, 2014.

[38] E. Montagu, *The Man Who Never Was*. J. B. Lippincott Company, 1954.

[39] C. Stoll, "Stalking the wily hacker," *Communications of ACM*, no. 5, pp. 484–497, 1988.

[40] W. Cheswick, "An evening with berferd, in which a cracker is lured, endured, and studied," *USENIX Conference*, no. 5, pp. 163–173, 1992.

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

[42] J. Yuill, M. Zapper, D. Denning, and F. Feer, "Honeyfiles: Deceptive Files for Intrusion Detection ," *IEEE Information Assurance Workshop*, 2004.

[43] L. Zhao and M. Mannan, "Explicit Authentication Response Considered Harmful," in *New Security Paradigms Workshop (NSPW)*, pp. 77–86, 2013.

[44] 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.

[45] J. Michael, M. Auguston, N. Rowe, and R. Riehle, "Software Decoys: Intrusion Detection and Countermeasures," *IEEE Workshop on Information Assurance*, 2002.

[46] N. Rowe, " Counterplanning Deceptions to Foil Cyber-Attack Plans," *IEEE Workshop on Information Assurance*, pp. 221–228, 2003.

[47] J. Michael, " On the Response Policy of Software Decoys: Conducting Software-based Deception in the Cyber Battlespace," *26th Annual International Computer Software and Applications Conference*, pp. 10–12, 2002.

[48] N. Rowe, J. Michael, M. Auguston, and R. Riehle, "Software Decoys for Software Counterintelligence," *IA Newsletter*, vol. 5, no. 1, pp. 10–12, 2002.

[49] F. Cohen, I. Marin, J. Sappington, C. Stewart, and E. Thomas, " Red Teaming Experiments with Deception Technologies," *IA Newsletter*, 2001.

[50] D. Rogers, "Host-Level Deception as a Defense Against Intruders," 2004.

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

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

[53] 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)*, pp. 253–264, 2013.

[54] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151–180, 1998.

[55] 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 (SP)*, pp. 120–128, 1996.

[56] S. Peisert, M. Bishop, S. Karin, and K. Marzullo, "Analysis of computer intrusions using sequences of function calls," *Dependable and Secure Computing, IEEE Transactions on*, vol. 4, pp. 137–150, April 2007.

[57] A. Somayaji and S. Forrest, "Automated response using system-call delays," in *Proceedings of the USENIX Security Symposium*, 2000.

[58] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, Nov. 1984.

[59] "SourceForge.net: Open Source Software (http://sourceforge.net)."

[60] "Linux keylogger (http://sourceforge.net/projects/lkl/)."

[61] "Botnet-1.0 (http://sourceforge.net/projects/botnet/)."

[62] "irssi (http://irssi.org/)."

[63] "CERT Advisories. (http://www.cert.org/advisories)."

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

[65] J. Carr, *Cyber Warfare*. O'Reily, 2011.