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. It supports backwards compatibility, so that older programs can run on newer systems. It provides a basis for programmers to customize their environment, because they know that they can port these changes to other instances of the operating system.

But predictability poses problems. Predictability allows vulnerabilities that are exploitable on one system to be exploitable on all systems of that type. This concept, called "monoculture," posits that diversity of software increases the work factor of attackers. Randomly perturbing systems can undermine the attacker's ability to predict whether an exploit will work against any particular instance of the system [1]. Essentially, diversity supplies a specific form of unpredictability.

The intent of diversity is independence, which means that multiple entities achieve a particular result in such a way that the *only* common factor are the inputs. An example of this in software is N-version programming [2], in which multiple teams create software to perform the same actions, but do so in different ways. That the software produces the same results adds credibility to those results. Fault tolerance work hypothesizes that faults are independent, and therefore can be compensated for by using voting or Byzantine protocols [3, 4].

At the system level, approaches to diversity generally involve randomness. For example, address space layout randomization (ASLR) causes the system to randomize the placement of pages of a program in memory during execution. A return-to-libc or ROP attack that relies on a buffer overflow causing a branch to a library function or gadget may fail, 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 [5] 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 configured to automatically restart daemons like a web server, 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 diverse system.

System constraints limit the effectiveness of diversity because many aspects of operating systems cannot be random without sacrificing efficiency or reliability. The obvious question is how to add sufficient uncertainty while maintaining those qualities.

Consider what "efficient" and "reliable" mean in the context of using 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 hypervisor tools to protect the application from a malicious operating system are rapidly evolving [6–9]. 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 size remains at an average of 125 lines of code [10]. 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. We argue that unpredictable behavior can be used as 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 looses 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 [9, 11–19] 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 areas which have been under-studied.

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 or not.

Researchers have studied building diverse computer systems. Forrest *et al.* [1] proposed guidelines and advocated the use of randomized compilation techniques, which motivated later work in this area [20]. These advocate breaking unnecessary consistencies in the generation of instructions to introduce costs for attack exploitation. These methods are passive, and do not adapt to changing attacks. Instruction randomization techniques are complementary and orthogonal to our work, as software generated with such techniques will make a computer system even more secure.

Chew and Song [21] proposed mitigating buffer overflows by employing randomization of system call mappings, global library entry points, and the stack placement. However, these methods do not employ inconsistent deception with probabilistic loss of functionality.

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 [9, 11–14, 17, 19], high-performance I/O systems [16–18], and other hardware access techniques [15] facilitate migration of kernel APIs into the application itself, and are, in some cases, implemented in higher level languages [14]. 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 [22–27].

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 [28]. 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 [29], and foreshadowed much of the honeypot work [30, 31]. Zhao and Mannan [32] 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 [33–37]. 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. The work assumed consistency of the deception.

Red-teaming experiments at Sandia tested the effectiveness of network deception on attackers working in groups [38]. The deception mechanisms at the network level successfully delayed attackers for a few hours. They apparently wore down those who were exposed to it and prompted some groups to quit before the experiments had ended.

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 [39]. But many command interpreters perform some of the requested actions directly, without invoking system calls and so bypassing the wrapper.

Almeshekah and Spafford [40] further investigated the adversaries' biases and proposed a model to integrate deception-based mechanisms in computer systems. While this model does not address challenges of inconsistent deception, the implementation of Chameleon will leverage it.

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 compromise its basic architecture or the attacker's steps to compromise the system.

True unpredictability requires randomness at a level that would cause the the attacker to get inconsistent results. This observation leads to the notion of *inconsistent deception* [41], a model of deception that challenges the cornerstone of projecting false reality with internal consistency. Neagoe and Bishop argued that an attacker will have no idea of whether she is exposed under a deception or a normal system is truly malfunctioning, but will feel disoriented and may withdraw from the situation. In this paper we tested inconsistency with automated attacks and proved their idea with some preliminary results from keyloggers and botnets. When a keylogger is running in the inconsistent deceptive environment, it meets with partial key loss or random key injection.

Iago attacks [42] are a good example of how such deception might work. An Iago attack occurs when a trusted program, designed to run on an untrusted system, is compromised by the untrusted kernel returning integer values to system calls designed to cause the trusted program to violate its security policy. This work focuses on how to attack trusted programs. The notion of monitoring system calls on which software depends was first studied by Forrest *et al.* [43, 44], who hypothesized that abnormal sequences of system calls might indicate malicious processes. Peisert and his colleagues extended this work to include function calls [45]. Both these works focused on the detection of attacks during runtime. Our aim is to protect the hosts rather than simply detect attacks. Taking a deeper view into monitoring system call parameters and randomly employing various strategies, we are going to build a more powerful spectrum-behavior OS with both inconsistent and consistent deception, and software diversity.

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 particular case study, these are often errors that are within the specification of the network or potential storage failure modes; a robust application would detect these issues with end-to-end checks [46].

We performed a preliminary study using `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 select the strategies below based on analysis of the types and frequency of system calls invoked by benign processes and malware. Ideally, we would like to se-

lect system calls that are more frequent in malware. We selected 39 benign software from sourceforge [47] and 86 malware samples for Linux, including 17 backdoors, 20 general exploits, 24 Trojan horses and 25 viruses and compared the system calls they invoke. We found that malware invokes a system call set that is smaller than benign software; approximately 50 different system calls. The most common system calls invoked by malware include `write()`, `wait()`, `clone()`, `close()`, `read()`, `open()`, `send()` and `fstat()`.

In selecting strategies for spectrum behavior, we perturb calls that still allow code to run, but do harm to malware. We found that the following calls are critical to process start-up and execution, and cannot be easily varied: `fstat()`, `getuid()`, `ioperm()`, `set_thread_area()`, and `mprotect()`. Other system calls, however, will just deviate from normal track innocuously when their parameters are changed. For instance, decreasing the number of bytes for `write()` will cause a keylogger to lose keystrokes, silencing a `send()` might cause a process to fail sending an e-mails, and extending time in `nanosleep()` will just slow down a process. The unpredictability coverage is defined as the set of system calls that are safe for spectrum behavior and are of strong relevance to malware execution. It currently includes the following system calls: `open()`, `read()`, `write()`, `lseek()`, `socket()`, `send()`, `recv()` and `nanosleep()`.

We discovered some initial strategies for spectrum behavior, varying several types of parameters that are widely used in system calls.

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 a scripts, and the reading or exfiltrating of 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 pointer: this approach simulates file corruption by randomly changing the file descriptor pointer between some invocations of `read()` and `write()`.

We first applied unpredictability to the Linux Keylogger (LKL) [48], a userspace 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.

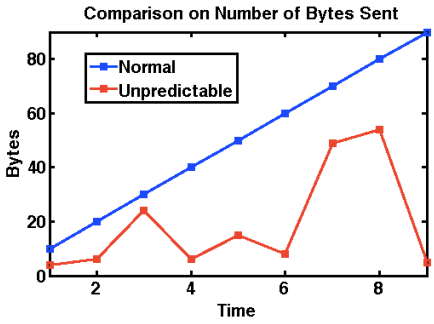

Figure 1: Comparison of email bytes sent from bots in predictable and unpredictable environments.

Next we applied unpredictability to the BotNET [49] 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` [50] to configure the botnet architecture with a botherder, bots and victims. The unpredictable strategies were applied to one of the bots.

We first tested commands that successfully reached the bot under the unpredictable environment, such as `adduser`, `deluser`, `list`, `identify`, `access`, `memo`, `sendMail` and `part`. The bot gets the command through calling `read()` for one byte each time, and one lost byte will destroy the command from being recognized. Here we randomly silenced the `read()` system calls and measured how many commands were correctly received by the bot. Nearly 40% of the commands from the bot herder were lost under our unpredictable environment.

We measured the effects of the unpredictable environments on the victim's ability to send spam emails (see 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.

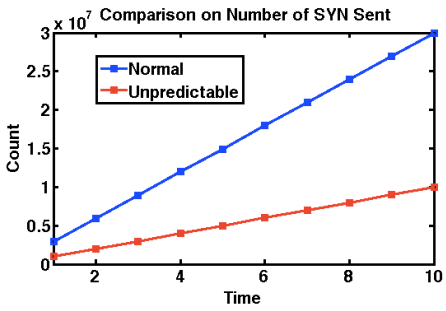

Figure 2: Comparison of SYN-flood attacks in standard and unpredictable environments. Unpredictability can increase the DDoS resource requirements.

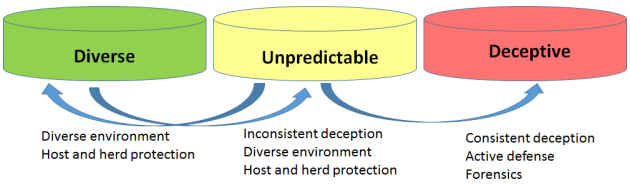

Figure 3: Chameleon can transition processes among three operating modes: *Diverse*, to protect benign software; *Unpredictable*, to disturb unknown software; and *Deceptive*, to analyze likely malware.

4 Spectrum-Behavior OS

Chameleon 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: (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. This is illustrated in Figure 3.

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. Unknown processes run in the **unpredictable** environment, where a subset of the system calls have their parameters modified or are silenced probabilistically. The execution of processes in this environment is unpredictable as they can lose some I/O data and functionality. An unpredictability threshold prevents processes from crashing. If a process running in this environment is malicious, it will have difficulty accomplishing its tasks as some system call options used to exploit operating system vulnerabilities might not be available all the time, some sensitive data being collected from and transferred to the system might get lost, and network connectivity with remote malicious hosts is not guaranteed. The idea is to create a foggy environment for the attacker, thereby protecting the host. It also offers herd protection to a community of hosts by raising the bar for the implementation of large-scale attacks, as bots or similar malware running in the unpredictable environment will not be reliable anymore. 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 false front while tracking down her actions. In this environment, files the attacker intends to leak will be honeyfiles, and any system privileges she thinks she has will be bogus. Connections and activities with malicious remote hosts will be intercepted and logged. The goal is to give the attacker the illusion that she controls the system, while forensic data is collected and forwarded to response teams such as CERT [51].

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.

The Chameleon prototype is ongoing work. Diversity is implemented with variations of the Graphene library OS [12]. Unpredictability is primarily implemented using the host ABI, although we expect some of this may be usefully implemented in the library OS as well.

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 [6–9]. 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 online will be tempted to whitelist the game if it isn't playable. We believe the types of variation can be adjusted dynamically, potentially with user feedback. If successful, sacrificing predictable behavior can finally give systems an edge over one of the primary sources of computer compromises [52]: malware installed by drive-by downloads and social engineering.

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)*, HOTOS '97, 1997.

[2] 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.

[3] 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.

[4] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché, "Upright cluster services," in *SOSP*, pp. 277–290, 2009.

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

[6] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP*, 2013.

[7] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *ASPLOS*, pp. 265–278, ACM, 2013.

[8] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 81–96, ACM, 2014.

[9] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 267–283, USENIX Association, 2014.

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

[11] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt, "Rethinking the library OS from the top down," in *ASPLOS*, pp. 291–304, 2011.

[12] 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.

[13] 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 *EuroSys*, 2013.

[14] 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 *ASPLOS*, 2013.

[15] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: safe user-level access to privileged cpu features," in *OSDI*, pp. 335–348, 2012.

[16] 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 *OSDI*, pp. 49–65, 2014.

[17] D. Schatzberg, J. Cadden, O. Krieger, and J. Appavoo, "Multilibos: An os architecture for cloud computing," 2014.

[18] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 1–16, USENIX Association, Oct. 2014.

[19] 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 *VEE*, pp. 44–54, 2007.

[20] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Proceedings of IEEE Security & Privacy*, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

[33] 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.

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

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

[36] 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.

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

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

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

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

[41] V. Neagoe and M. Bishop, "Inconsistency in deception for defense," in *Proceedings of the 2006 Workshop on New Security Paradigms*, NSPW '06, pp. 31–38, 2007.

[42] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 253–264, ACM, 2013.

[43] 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.

[44] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff., "A sense of self for unix processes," *IEEE Symposium on S&P*, pp. 120–128, 1996.

[45] 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.

[46] 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.

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

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

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

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

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

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