

A Survey of Randomization Techniques Against Common Mode Attacks Λ

Angelos D. Keromytis
and
Vassilis Prevelakis

Technical Report DU-CS-05-04
Department of Computer Science
Drexel University
Philadelphia, PA 19104
April, 2005

A Survey of Randomization Techniques Against Common Mode Attacks*

Angelos D. Keromytis Vassilis Prevelakis
Columbia University Drexel University

Abstract

Software systems often share common vulnerabilities that allow a single attack to compromise large numbers of machines (“write once, exploit everywhere”). Borrowing from biology, several researchers have proposed the introduction of artificial diversity in systems as a means for countering this phenomenon. The introduced differences affect the way code is constructed or executed, but retain the functionality of the original system. In this way, systems that exhibit the same functionality have unique characteristics that protect them from common mode attacks. Over the years, several such approaches have been proposed. We examine some of the most significant

*Authors’ addresses: Angelos D. Keromytis, Department of Computer Science, Columbia University, M.C. 0401, 1214 Amsterdam Avenue, New York, NY 10027, USA, *angelos@cs.columbia.edu*; Vassilis Prevelakis, Computer Science Department, Drexel University, 3141 Chestnut Street, Philadelphia, PA 19104, USA, *vp@drexel.edu*

techniques and draw conclusions on how they can be used to harden systems against attacks.

1 Introduction

Recent widespread disruptions of systems across the Internet such as the Slammer and MSBlaster worms have underlined the inherent weakness of an infrastructure that relies on large numbers of effectively identical systems [29, 28, 50]. Common elements in these systems include the operating system, the system architecture (*e.g.*, Sparc64 processors), particular applications (*e.g.*, email clients, web browsers, document processing software), and the internal network architecture. Common-mode attacks occur when an attacker exploits vulnerabilities in one of these common elements to strike large numbers of victim machines.

Several researchers observed that if each of these systems were different, the attacker would have to customize their technique to the peculiarities of each system, thus reducing both the scope of the attack and the rate of exploitation [31]. However, running different systems in a network creates its own set of problems involving configuration, management, and certification of each new platform [55, 8]. In certain cases, running such multi-platform environments can decrease the overall security of the network [44]. Finally, creating a large number of different systems through techniques such as N-version programming [9] not only presents cer-

tain practical challenges [30] but often results in systems that are not truly diverse [13, 36].

Thus, a new set of work set out to retroactively introduce randomness in systems. The basic premise is that by introducing randomness in existing systems, we can vary their behavior sufficiently to prevent common mode attacks. In this way, our systems are similar enough to ease administration, but sufficiently different to resist common mode attacks. Randomization can be introduced in various parts of a system, such as the configuration of the network infrastructure so that remote attackers cannot target a specific host or service (*e.g.*, the White House site or the Microsoft software update server), the implementation of specific protocols (*e.g.*, changing certain aspects of the TCP/IP engine to reduce the risk of fingerprinting), or even the processor architecture to guard against foreign-code injection attacks.

In this paper, we describe various randomization techniques, examine how they can be used to strengthen the security of systems, and classify them based on the aspect of the system that is randomized.

2 Classification

Attacks that are mounted against networked systems can be separated into two groups: attacks against software typically attempt to subvert the execution logic, while attacks against protocols (*e.g.*, sending false TCP window advertisements

[46]) generally aim to modify system behavior.

2.1 Attacks against Software

In this class, we typically encounter code-injection attacks [5, 43, 39]. Such attacks aim to deposit executable code (typically machine code, but there are cases where intermediate or interpreted code has been used) within the address space of the victim process, and then transfer control to this code. In many instances, the code that the attacker wants to execute is already present in the address space of the victim (*e.g.*, code that invokes the system command-line interpreter). The attacker does not need to send any code, but merely to cause the program to jump to the desired part of the existing code (“jump-to-libc” attacks).

Buffer overflows are the primary method of executing such attacks. Since statically defined memory buffers are finite, if the attacker manages to overflow the buffer, the additional data will eventually overwrite some part of the address space that is critical to the correct execution of the victim task. At the very least, the process will be unable to continue functioning, thus resulting in a Denial of Service. The “ping of death” attack [3], whereby a single ICMP packet with a payload that was just one byte larger than the maximum allowed by the protocol and expected by the target system, caused the entire system to crash. A more serious buffer overflow attack is one that forces the victim process to execute code selected or

supplied by the attacker. The Slammer [4] and MSBlaster [16] worms used such attacks to inject code in computers all over the network.

While buffer overflows are the most common means for code-injection attacks, they are by no means the only. The “format-bugs” attack [18] uses specially crafted format strings that are used by the `printf`-like functions. Through what the paper terms “creative use” of the formatting directives, the victim process can be subverted.

2.2 Protocol Attacks

The second class of attacks are arguably more dangerous (but also more rare) because they target weaknesses in the protocols and as such can target systems irrespective of the particular protocol implementation. The SYN Flood attack [2, 53, 47] that caused widespread disruption in the Internet exploited a characteristic of the TCP protocol, namely the long timeout value during the initial three-way handshake.

Another technique that, although is not an attack, can be used as a precursor to one, is fingerprinting, a technique that allows remote attackers to gather enough information about a system so that they can determine its type and software configuration (version of operating system, applications *etc.*). This information can then be used to determine what vulnerabilities may be present in that configuration and

thus better plan an attack.

3 Achieving System Diversification

The recurrent instances of successful attacks against networked systems, demonstrate that it is very difficult to prevent such incidents. Our aim is to use diversification techniques to limit the spread of these attacks and thus contain the damage caused.

The diversification techniques that have been proposed over the years can be broadly classified into three categories: those that modify the structure of the system, those that modify the system's (execution) environment, and those that change the system's behavior. The first two are generally used in the context of software safety and security, *e.g.*, against buffer overflow attacks, while the last is used in hardening network protocols. Naturally, different techniques can be employed in systems that may be vulnerable to multiple different types of attacks, *e.g.*, an implementation of the TCP/IP stack. [20] gives an overview of various protection mechanisms, including randomization techniques, and makes recommendations on choosing obfuscation (of interface or implementation) *vs.* restricting the same.

In the rest of this section we describe each of these categories, and examine the techniques that can be used to enhance diversity.

3.1 Modifying the Structure

Structure modification techniques typically insert code that performs sanity or consistency checks at various points in the execution of the program. Since these techniques are independent of the application logic, *i.e.*, are not written with specific applications in mind and cannot check the semantic consistency of a program's variables and data structures, they depend on the use of a random value that an adversary cannot determine by means other than trial and error.

The work that first introduced the notion of biologically inspired diversity for software protection is [26]. The authors mention several possible approaches to implementing such diversity:

- Randomly adding non-functional code (*e.g.*, no-op machine instructions or instructions that do not affect program data and registers) will both affect timing relations and change the relative location of instructions in the process address space. This can make it harder for an attacker to exploit race conditions or mount remote timing attacks [14], or to cause a program's execution to jump to a particular code segment (also known as "jump-into-libc" attacks). Although fairly straightforward, this approach can be fairly expensive from a performance perspective, since additional instructions need to be executed, also requiring more memory (and, more importantly, space in the

CPU instruction cache).

- Randomly reorder code, while maintaining its execution semantics. This can more easily be done in terms of basic blocks (*i.e.*, code segments that do not contain jumps), but other schemes are possible. Such code re-arranging would make it more difficult to mount “jump-into-libc” attacks at the cost of performance, since the compiler has (presumably) produced a near-optimal instruction schedule.
- Pad each stack frame by a random amount, either on a per-compilation or per-execution basis. This can obscure the location of a procedure’s variables and return pointer, making it difficult (but not impossible) to mount a buffer overflow attack. Similarly, stack frames can be allocated in a non-contiguous manner, global variables can be re-arranged, *etc.* A practical implementation of this was presented by Alexander [6].
- Vary the names of files, dynamic library names, system-call numbers, *etc.* This can make *automated* exploitation of a system subsequent to a successful intrusion more difficult, since the system components (files, system calls, libraries, *etc.* needed by the attacker need to be located/named for each individual system. Note that this technique more properly belongs to the second category of techniques we examine. We shall discuss some variants

of it in the next section.

The authors implemented the frame-padding scheme, concluding that both the space and execution overheads were in the 10-15% range for the few applications they tested. However, the idea of randomizing aspects of the program itself caught on, spawning a variety of schemes to address (primarily) buffer overflow attacks.

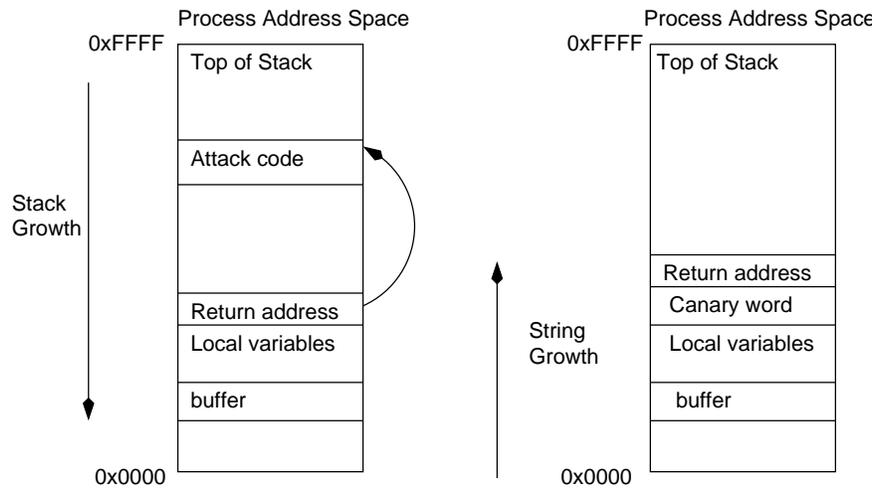


Figure 1: **Stack buffer overflow (left) and StackGuard-modified stack frame (right).**

Perhaps the best-known of these techniques is StackGuard [21], a system that protects against stack-based buffer overflows. It is implemented as a patch to the popular *gcc* compiler that inserts a *canary* word right before the return address in a function's activation record in the program stack, as shown in Figure 1. The canary is checked just before the function returns, and execution is halted if it is not the correct value, which would be the case if a stack-smashing attack had overwritten

it. This protects against simple stack-based attacks, although some attacks were demonstrated against the original approach [15], which has since been amended to address the problem. The performance penalty of StackGuard is generally fairly low, although in pathological cases (*e.g.*, programs with deep recursion) it can be as high as 125%. A related system, MemGuard [21], extends protection to memory regions allocated in the heap, preventing more powerful attacks (*e.g.*, [39]) at the cost of increased overhead. ProPolice is another instantiation of StackGuard, also modifying the location of variables in the program stack to avoid partial-overflow attacks and make exploitation of overflows still more difficult.

Another relevant approach is PointGuard [19], which encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. This is also implemented as an extension to the *gcc* compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme.

Address obfuscation [11], randomizes the absolute locations of all code and data, as well as the distances between different data items. Several transformations are used, such as randomizing the base addresses of memory regions (stack, heap, dynamically-linked libraries, routines, static data, *etc.*), permuting the order of variables/routines, and introducing random gaps between objects (*e.g.*, randomly pad stack frames or *malloc()*'ed regions). Although very effective against “jump-

into-libc” attacks, it is less effective against other common attacks, due to the fact that the amount of possible randomization is relatively small. However, address obfuscation can protect against attacks that aim to corrupt variables or other data. The performance impact of this approach is fairly negligible, since all the randomization is performed at compilation time. To avoid the cost of recompiling programs in order to change the randomization values, the PaX Address Space Layout Randomization (ASLR) technique allows the system to add at load time a random offset to the base address of the stack, heap, code and mmap’ed segments [52]. However, Shacham *et al.* [49] show that in many cases address-space randomization is not very effective, because 32-bit address spaces do not allow for sufficient entropy in the placement of data and control information in memory.

As we mentioned earlier, code obfuscation techniques [38] are also used to harden program binaries against static disassembly. These approaches generally inject “dead” code and rename variables or class names to hinder analysis. We do not further consider them here, as such attacks do not generally pose a danger to the security of networked systems (with the possible exception of various Digital Rights Management schemes).

A persistent concern in employing techniques such as the ones described above, is to maintain the efficiency of the application. In other words, the overheads associated with the use of these mechanisms must be minimized. Naturally, this

discourages the use of more exhaustive and hence more expensive techniques. If, however, we can identify the parts of the code where a bug has a higher probability of resulting in a security vulnerability, we can reserve the use of the more expensive mechanisms to these sensitive regions.

Tools developed under the DARPA funded CHATS/CoSAK project aim to facilitate the identification of such regions. This work is based on the assumption that a small percentage of functions near a source of input (such as file I/O) are the most likely to contain a security vulnerability [23]. Special mechanisms (*e.g.*, code emulation, execution under a virtual environment, or limitations on privileges) can be activated when the flow of control strays into the sensitive regions.

3.2 Modifying the Environment

Systems do not exist in isolation, but need to interact with their environment (be it the processor and memory subsystems, the operating system, the network topology, *etc.*). Techniques that randomly modify the environment of a system (as well as the system itself) can hinder an attacker by increasing the complexity of the attack. To see how randomization techniques can be used to influence the execution environment let us look a bit closer at the problem of code-injection attacks; in that context, by “system” we refer to the software (program) that is being protected.

Code-injection attacks can only succeed if the injected code is compatible with

the execution environment. For example, injecting *x86* machine code to a process running on a SUN/SPARC system may crash the process (either by causing the CPU to execute an illegal op-code, or through an illegal memory reference), but will not cause a security breach. Notice that in this example, there may well exist sequences of bytes that will crash on neither processor.

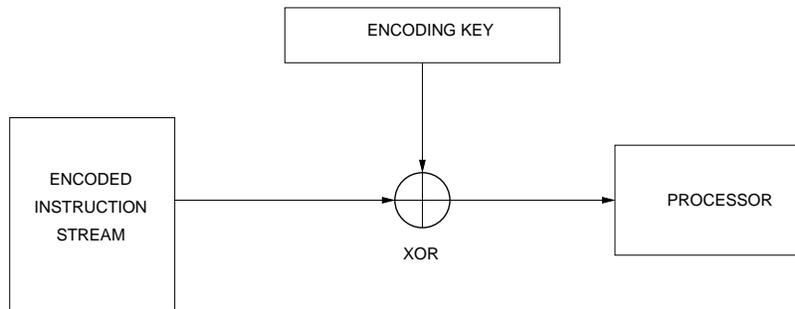


Figure 2: **Previously encoded instructions are decoded before being processed by the CPU.**

The instruction randomization technique [32] leverages this observation by creating an execution environment that is unique to the running process, so that the attacker does not know the “language” used and hence cannot “speak” to the machine. This is achieved by applying a reversible transformation between the processor and main memory, as shown in Fugyre 2. Effectively, new instruction sets are created for each process executing within the same system. Code-injection attacks against this system are unlikely to succeed, as the attacker cannot guess the transformation that has been applied to the currently executing process. Of course, if

the attackers had access to the machine and the randomized binaries through other means, they could easily mount a dictionary or known-plaintext attack against the transformation and thus “learn the language”. However, the system is primarily concerned with attacks against *remote services* (e.g., http, dhcp, DNS, and so on). Vulnerabilities in this type of server allow external attacks (*i.e.*, attacks that do not require a local account on the target system), and thus enable large-scale (automated) exploitation. The system assumes a suitably modified processor, and shows how an operating system can be modified to manage randomized processes at negligible performance overhead. In the absence of such a processor, a full-machine emulator such as Bochs [1] can be used, with considerable performance degradation.

The power of the technique can be demonstrated by its applicability to other settings, such as SQL injection attacks [12]. Such attacks target databases that are accessible through a web front-end, and take advantage of flaws in the input validation logic of Web components such as CGI scripts. The concept of instruction randomization has been applied to that setting, to create instances of the SQL language that are unpredictable to the attacker. Preliminary results indicate that the mechanism imposes negligible performance overhead to query processing, and can be easily retrofitted to existing systems. The same technique can easily be applied to any interpreted-language setting that is susceptible to code injection attacks.

Other similar work includes RISE [10], which applies a randomization technique similar to the one described above by using an emulator attached to specific processes (as opposed to a full system emulator). The inherent use of and dependency on emulation makes RISE simultaneously more practical for immediate use and inherently slower in the absence of hardware.

O'Donnell and Sethu [42] study algorithms for the assignment of distinct software packages (whether randomized or inherently different) to individual systems in a network, towards increasing the intrinsic value of available diversity. Their goal is to limit the ability of a malicious node to compromise a large number (or any) of its neighbors with a single attack.

In [17], the authors consider randomization of system-call mappings, global library entry points, and stack-frame placement. The last technique has the potential of preventing malicious code from being executed altogether, while the first two primarily make it difficult for the malicious code to perform its task. An attack aware of the use of these techniques can inject logic that tries to determine the correct system-call mappings and library entry points by examining the program code itself, at a considerable increase in the complexity of the injected code. The first two techniques impose negligible performance or memory overheads, since the modifications are made at compile time, while the last can be wasteful of memory.

In a different context, randomization of a system's environment has been used

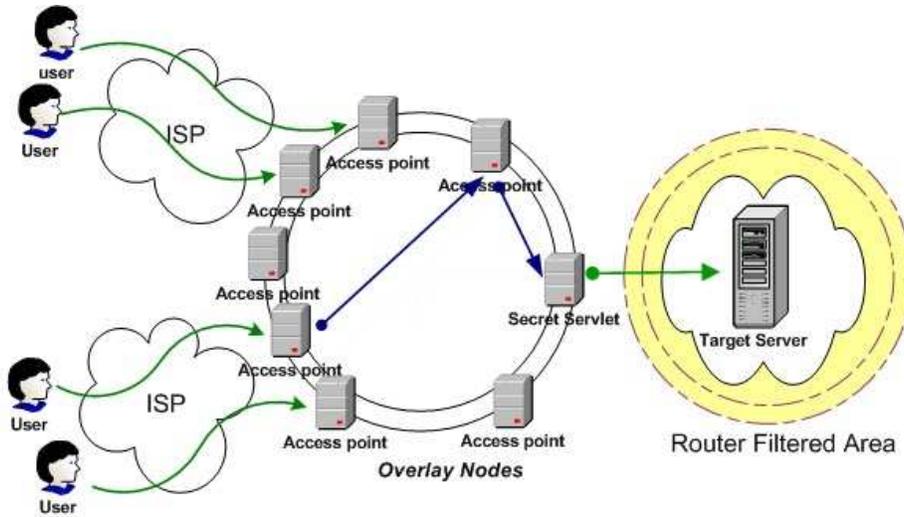


Figure 3: **Secure Overlay Services anti-DoS architecture.** Access Points represent an entry point to the overlay. Overlay nodes can simultaneously serve as access point and secret servlet. The overlay nodes are dedicated nodes (*i.e.*, they are not end-user-controlled nodes).

to combat network-based denial of service (DoS) attacks. The Secure Overlay Services (SOS) [33, 34] approach, depicted in Figure 3, addresses the problem of securing communication on top of today’s existing IP infrastructure from DoS attacks, where the communication is between a pre-determined location and users, located anywhere in the wide-area network, who have authorization to communicate with that location. The scheme was later extended to support unknown users, by using Graphic Turing Tests to discriminate between zombie machines and real humans [40].

In a nutshell, the portion of the network immediately surrounding the target (location to be protected) aggressively filters and blocks all incoming packets whose source addresses are not “approved”. The small set of source addresses that are “approved” at any particular time is kept secret so that attackers cannot use them to pass through the filter. These addresses are picked from among those within a distributed set of nodes throughout the wide area network, that form a *secure overlay*: any transmissions that wish to traverse the overlay must first be validated at entry points of the overlay. Once inside the overlay, the traffic is tunneled securely for several hops along the overlay to the “approved” (and secret from attackers) locations, which can then forward the validated traffic through the filtering routers to the target. The two main principles behind this design are: *(i)* elimination of communication “pinch” points, which constitute attractive DoS targets, via a combination of filtering and overlay routing to obscure the identities of the sites whose traffic is permitted to pass through the filter, and *(ii)* the ability to recover from random or induced failures within the forwarding infrastructure or among the overlay nodes.

The overlays are secure with high probability, given attackers who have a large but finite set of resources to perform the attacks. The attackers also know the IP addresses of the nodes that participate in the overlay and of the target that is to be protected, as well as the details of the operation of protocols used to perform

the forwarding. However, the assumption is that the attacker does not have unobstructed access to the network core. That is, the model allows for the attacker to take over an arbitrary (but finite) number of hosts, but only a small number of core routers. It is more difficult (but not impossible) to take control of a router than an end-host or server, due to the limited number of potentially exploitable services offered by the former. While routers offer very attractive targets to hackers, there have been very few confirmed cases where take-over attacks have been successful. Finally, SOS assumes that the attacker cannot acquire sufficient resources to severely disrupt large portions of the backbone itself (*i.e.*, such that all paths to the target are congested).

Under these assumptions, by periodically selecting a new “approved” overlay node at random, a site can allow only authorized clients to communicate with it. An attacker must either amass enough resources to subvert the infrastructure itself, or attempt to guess the identity of the current approved node. Effectively, SOS allows the creation and use of an arbitrary number of virtual topologies over the real network (which can, perhaps perversely, viewed as a monoculture), which only legitimate users can use. A similar approach is espoused by the MayDay system [7].

MOVE [51] eliminates the dependency on network filtering at the ISP POP routers by keeping the current *location* of the server secret and using process mi-

gration to move away from targeted locations. This is similar to the concept of “hidden servers” that was introduced by anonymity systems such as Tor [45, 25]. Another system similar to MOVE is described in [35]

3.3 Modifying the Behavior

Computer systems are to a large extent deterministic and this can be used as a means of identification (fingerprinting), or, worse, as means of subverting a system by anticipating its response to various events.

Having a system with predictable behavior can have devastating consequences for its security. The most celebrated example is the attack that exploited easy to guess TCP/IP packet sequence numbers [48]. By being able to guess the sequence number of a TCP connection with a remote system, we can construct and transmit replies to packets that we never receive (perhaps because a firewall prevents the remote system from talking to us, or because we use a spoofed source address in our packets).

More recently, a denial of service attack based on the TCP retransmission timeout [37], allowed an attacker to periodically send bursts of packets to the victim host, forcing the TCP subsystem on the victim host to repeatedly timeout causing near-zero throughput, exploiting the fact that receivers used the same minimum Retransmission TimeOut (minRTO). In this case as well, by changing the behavior

of the TCP implementation (randomizing the retransmission timeout), the attack can be mitigated. However, because TCP must be both “fair” to other network flows and exhibit good performance, and because the RTO value is fundamental to determining this “fairness,” there is a tradeoff between resistance to low-rate DoS attacks and performance.

[22] discusses low-bandwidth algorithmic-complexity denial of service attacks in which attackers exploit deficiencies in applications’ data structures that allow them to invoke worst-case running times. Examples of these include the IP fragment reassembly queue, various intrusion detection systems, Web caches, DNS servers, *etc.* These attacks work because of the uniformity and determinism of the algorithms used to construct and handle data structures such as hash tables: an attacker can force an implementation to always exhibit worst-case performance, which can be considerably worse than the average (expected-case) performance. The authors show how hashing and randomizing techniques (*e.g.*, by using a random seed in the hash function) can help protect against such attacks with little performance impact.

OpenBSD, an open-source unix variant with a focus on security, uses randomness in areas such as unpredictable RPC transaction IDs, inode generation numbers, IP datagram IDs, TCP Initial Send Sequence (ISS) numbers, process identifiers, local TCP/UDP ports used for new connections, DNS query IDs, temporary

file names, *etc.* [24] The goal is to introduce enough unpredictability in standard protocols and mechanisms that the attacker's task is made significantly harder. We mention a few instances of such randomization:

- **Process PIDs.** Programmers often use this value as if it is random, possibly because of the compellingly attractive argument that “pid numbers are effectively random on a busy enough system.” Code like “`srandom(getpid())`” is quite common, as is code similar to the following:

```
char buf[20];

sprintf(buf, "/tmp/foo-%d", getpid());

(void) mkdir(buf, 0600);
```

In a normal system, an attacker can easily predict the PID and thus the obvious race attack is trivial: the attacker creates the directory first, choosing the mode and ownership; subsequently it is possible to look at and replace files in the directory.

- **IP datagram IDs.** Each IP packet contains a 16-bit identifier which is used, if the packet has been fragmented, for correctly performing reassembly at the final destination. In most systems, this identifier is simply incremented every time a new packet is sent out. By looking at the identifier in a se-

quence of packets, an outsider can determine how busy the target machine is. OpenBSD uses a Linear Congruential Generator (LCG) seeded with random parameters to create the IP IDs.

- **Randomness added to the TCP ISS value for protection against spoofing attacks.** The predictability of TCP ISS values has been known to be a security problem for many years [41]. Typical systems added either 32K, 64K, or 128K to that value at various different times. OpenBSD adds a fixed amount plus a random amount, significantly decreasing the chances of an attacker guessing the value and thus being able to spoof connection contents.

The general Internet philosophy of "being conservative in what you send and liberal in what you accept" (RFC1341), while enhancing interoperability, sometimes creates vulnerabilities by allowing greater ambiguity in what a networked application may accept.

Especially in the case of the Internet Protocols these minor variations have been used as the basis of attacks (*e.g.*, the overlapping fragment attacks and the small packet attacks of the early 1990s), and more recently as a means to facilitate fingerprinting.

OpenBSD's packet filter, `pf(4)`, includes a "scrub" function that normalizes and de-fragments incoming packets. This allows applications and hosts on the internal network some for of protection against hand-crafted packets designed to

trigger vulnerabilities. Another approach is to apply a similar technique to outgoing packets in order to hide identifying features of the IP stack implementation [27]. A key part of the process of the obfuscation process is protection against time-dependent probes. Different TCP implementations have variations in their timeout counters, congestion avoidance algorithms, *etc.* By monitoring the response of the host under inspection to simulated packet loss, the timing probe can determine the version of the TCP implementation and by extension that of the OS. Also the use of various techniques for rate limiting ICMP messages by the victim system, can provide hints to the attacker. The effectiveness of such probes can be reduced, by homogenizing the rate of ICMP traffic going through the system that connects the trusted network to the outside world, or by introducing random delays to ICMP replies.

4 Discussion

So far we have discussed various randomization techniques and how they may be used to enhance security. However, for these techniques to be truly useful, care must be taken to ensure that they impose a significant barrier to a potential attacker.

For example, the use of randomization is recommended in cases where the search space is large enough to make an exhaustive search by the adversary infeasible. For example the Linux ASLR technique described earlier, is not effective in

systems with 32-bit address structures. This is because a 32-bit address space is small enough that every valid page in the address space can be easily found through exhaustive search [56, 54]. Moreover, in the case of ASLR the search space is far smaller, as PaX applies randomization (`delta_mmap`) only in bits 12–27 of the 32-bit address, resulting in 16 bits of search space.

Attackers may also try to obtain information from the target system that provides clues as to the random values being used. This in turn may increase the impact of known bugs that allow read-access to the victim system. Such bugs may have been considered benign since they cannot harm the victim system, but if a technique such as instruction randomization is in force, such a bug may leak enough information to allow the attacker to determine the value of the encoding key.

Moreover, randomization may reveal or even cause incompatibilities by creating random excursions from the expected behavior. These variations may be within the behavior described by, say, the protocol specifications, but may, nevertheless, cause applications to be confused. For example, assume that a client application randomly selects a timeout value and that value happens to be near the low bound of acceptable values, while a server application picks a delay value that is near the high end of the acceptable range. Under this scenario, the client may timeout before the server can respond, thus breaking the connection.

This brings us to the last point of this discussion, *i.e.*, the fact that randomization techniques generally prevent an enemy from taking over the victim application or system, by causing it to crash. This behavior, although generally preferable to ending up with a subverted asset, may not be acceptable since it may lead to a denial of service. In such cases randomization may be useful in detecting the failure, but needs to be complemented by additional mechanisms such as fault isolation, graceful degradation, *etc.*, that may allow the system under attack to continue running while avoiding being compromised.

4.1 Open Research Questions

Despite the considerable amount of work so far in the general area of system randomization, there remain several large open research questions and practical issues that require additional work. These include:

- Reconciling conflicts between the randomization technique used to protect a system and the needs of the system maintainer to monitor its operation and debug problems. All randomization techniques purposely obfuscate some aspect of the system; thus, standard monitoring and debugging tools may well not work correctly when applied to these. Although some of the protection techniques would appear to be easier than others in terms of compatibility with maintenance tools, the full impact of such techniques is not well

understood yet.

- Quantifying the strengths/weaknesses of the various techniques and their benefit to system security. Most techniques to date have been evaluated in terms of their impact on performance and their ability to stop (or not) specific attacks. A more realistic metric may be the increase on an attacker's workload, since many of these techniques operate in a finite (and usually relatively small) space of possible configurations.
- Developing a better understanding of the composition of different techniques, especially those addressing similar problems. In particular, we need to understand whether (and how) such composition may open new vulnerabilities and avenues for attack.
- Establishing formal and/or experimental frameworks for evaluating and comparing various techniques that address the same class of attacks.

Our hope is that our work here will serve as a starting point toward a better understanding of randomization techniques, and stimulate efforts to answer these open research problems.

As a first step toward achieving this understanding, we have attempted to characterize some of the randomization techniques that apply to software systems, as shown in Table 1. The diversity of existing randomization mechanisms makes it

difficult to compare all of them to each other; thus, we have limited ourselves to software-related protection mechanisms.

One important consideration is whether the use of the technique requires access to the source code of the system (*i.e.*, the software needs to be compiled with a special compiler). Another is the overhead associated with the use of the particular technique and we have looked at space overhead (significant increase in the size of the target system) and time overhead (increased execution time). Finally we have looked at the type of attacks that each technique attempts to protect against. One type, usually referred to as *jump-to-libc* attacks, redirect the execution of the program to another part of the code (already residing in the target application), while the other type cause the processor to execute new hostile code that has been injected into the address space of the victim, by the attacker.

5 Summary and Concluding Remarks

The commoditization of computer systems has dramatically lowered the cost of ownership of large collections of computers. It is, thus, no longer economically feasible to have one-off configurations for individual computers or networks, which, in turn, leads to monocultures, vulnerable to common-mode attacks. There is a lively debate going on as to the effects of a diverse computing environment on security. One camp claims that diversity is not required as it distracts from the

Techniques	Access to Source	Significant Space Overhead	Significant Time Overhead	Attack Type
Adding non-func. code	Yes	Yes	Yes (Note 1)	Existing Code
Code reordering	Yes	Yes	No	Existing Code
Stack-frame padding	Yes	Yes	No	Both
System Interface Rand.	No	No	No	Injected Code
Stackguard/ProPolice	Yes	No	No	Injected Code
MemGuard	Yes	No	Yes	Injected Code
PointGuard	Yes	No	No	Injected Code
ISR-software /RISE	No	No	Yes	Injected Code
ISR-hardware (Note 2)	No	No	Yes	Injected Code
PaX/ASLR	No	No	No	Existing Code

Note 1: If the processor executes the non-functional code (*e.g.*, NOP instructions), the overhead will depend on the number of additional instructions.

Note 2: ISR-software uses an emulated processor via Bochs, while ISR-hardware uses a specially modified processor.

Table 1: **Characteristics of various software randomization techniques.**

task of producing a single secure configuration that can then be widely deployed, thus spreading the development and security administration costs to a large number of machines. The other camp claims that by standardizing the interfaces between subsystems, multiple implementations can be deployed, thus reducing the risk of a single problem affecting all the deployed systems. Our view is that both sides are fundamentally wrong. Having potentially huge numbers of identically configured hosts invites disaster: no amount of effort can secure large software systems that have not been built with security in mind. Even in cases where formal methods have been used in the design, field upgrades and maintenance can weaken the security posture. On the other hand, attempting to introduce diversity through the

development of different software systems is not viable. Designing, developing and maintaining a system is so expensive that once we have a working version we tend to use it widely. Even in critical systems such as avionics, the same software is used on multiple hardware platforms (creating redundancy only at the hardware level). The failure of the inaugural flight of the Ariane 5 launcher due to a software bug crashing both navigation computers is proof that having the same software running on redundant hardware does not provide true redundancy.

Our intention has been to demonstrate that the effects of diversity can be introduced through automated means. The techniques described in this paper allow the introduction of small but critical variations to these off-the-shelf systems. While randomization techniques do not constitute the silver bullet that will solve the problem of generic software, or system exploits (these can only begin to be addressed if we abandon the current *ad hoc* design and development techniques) they do provide an effective means for mitigating attacks and exposing the bugs that make such attacks possible.

References

- [1] Bochs Emulator Web Page. <http://bochs.sourceforge.net/>.
- [2] Cert advisory ca-96.21: Tcp syn flooding. ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding,

September 1996.

- [3] Cert advisory ca-96.26: Denial-of-service attack via ping. ftp://info.cert.org/pub/cert_advisories/CA-96.26.ping, October 1996.
- [4] The Spread of the Sapphire/Slammer Worm. <http://www.silicondefense.com/research/worms/slammer.php>, February 2003.
- [5] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
- [6] S. Alexander. Improving Security With Homebrew System Modifications. *USENIX ;login.*, 29(6):26–32, December 2004.
- [7] D. G. Andersen. Mayday: Distributed Filtering for Internet Services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems USITS*, March 2003.
- [8] D. Aucsmith. Monocultures Are Hard To Find In Practice. *IEEE Security & Privacy*, 1(6):15–16, November/December 2003.
- [9] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

- [10] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.
- [11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.
- [12] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security Conference (ACNS)*, pages 292–302, June 2004.
- [13] S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2), February 1990.
- [14] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12 USENIX Security Symposium*, pages 1–14, August 2003.
- [15] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

- [16] CERT. Advisory CA-2003-21: W32/Blaster Worm. <http://www.cert.org/advisories/CA-2003-20.html>, August 2003.
- [17] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.
- [18] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–199, August 2001.
- [19] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.
- [20] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the National Information Systems Security Conference (NISSC)*, October 2000.
- [21] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

- [22] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, August 2003.
- [23] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the Security Vulnerability Likelihood of Software Functions . In *Proceedings of the 2003 International Conference on Software Maintenance (ICSM'03)*, pages 61–72, September 2003.
- [24] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proceedings of the 1999 USENIX Annual Technical Conference, Freenix Track*, pages 93–101, June 1999.
- [25] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–319, August 2004.
- [26] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS)*, 1997.
- [27] Fyodor. Remote OS detection via TCP/IP fingerprinting. <http://www.insecure.org/nmap/nmap-fingerprinting-article.html>,

1998.

- [28] D. E. Geer. Monopoly Considered Harmful. *IEEE Security & Privacy*, 1(6):14 & 17, November/December 2003.
- [29] G. Goth. Addressing the Monoculture. *IEEE Security & Privacy*, 1(6):8–10, November/December 2003.
- [30] J. Gray and D. Siewiorek. High-availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.
- [31] D. A. Holland, A. T. Lim, and M. I. Seltzer. An Architecture A Day Keeps The Hacker Away. In *Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA)*, pages 29–36, October 2004.
- [32] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, October 2003.
- [33] A. D. Keromytis, V. Misra, and D. Rubenstein. Secure overlay services. In *Proceedings of the ACM SIGCOMM Conference*, pages 61–72, August 2002.
- [34] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An Architecture For Mitigating DDoS Attacks. *IEEE Journal on Selected Areas of Communications (JSAC)*, 33(3):413–426, January 2004.

- [35] S. M. Khattab, C. Sangpachatanaruk, D. Moss, R. Melhem, and T. Znati. Roaming Honeypots for Mitigating Service-Level Denial-of-Service Attacks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 238–337, March 2004.
- [36] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.
- [37] A. Kuzmanovic and E. W. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants). In *Proceedings of the ACM SIGCOMM Conference*, August 2003.
- [38] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, October 2003.
- [39] M. Conover and w00w00 Security Team. w00w00 on heap overflows. <http://www.w00w00.org/files/articles/heaptut.txt>, January 1999.
- [40] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein. Using Graphic Turing Tests to Counter Automated DDoS

Attacks Against Web Servers. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 8–19, October 2003.

- [41] R. T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Computing Science Technical Report 117, AT&T Bell Laboratories, February 1985.
- [42] A. J. O’Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 121–131, October 2004.
- [43] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.
- [44] V. Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.
- [45] M. Reed, P. Syverson, and D. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications (JSAC)*, 16(4):482–494, May 1998.

- [46] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM Computer Communications Review*, 29(5):71–78, October 1999.
- [47] C. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. Analysis of a Denial of Service Attack on TCP. In *IEEE Security and Privacy Conference*, pages 208–223, May 1997.
- [48] Secure Networks Inc. A simple TCP spoofing attack. <http://niels.xtdnet.nl/papers/secret-spoof.txt>, February 1997.
- [49] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.
- [50] M. Stamp. Risks of Monoculture. *Communications of the ACM*, 47(3):120, March 2004.
- [51] A. Stavrou, A. D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein. MOVE: An End-to-End Solution To Network Denial of Service. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, February 2005.

- [52] P. Team. Documentation for the PaX project, Address Space Layout Randomization. <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
- [53] S. E. I. C. E. R. Team. CERT Advisory CA-96.21: TCP SYN Flooding and IP Spoofing Attacks. Technical report, SEI, Sept 1996. ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding.
- [54] Tyler Durden. Bypassing pax aslr protection. *Phrack*, 11(59), July 2002.
- [55] J. A. Whittaker. No Clear Answers on Monoculture Issues. *IEEE Security & Privacy*, 1(6):18–19, November/December 2003.
- [56] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer USENIX Technical Conference*, August 1993.