Building Diverse Computer Systems

Stephanie Forrest[†]
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

Anil Somayaji[†]
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
soma@cs.unm.edu

David H. Ackley
Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
ackley@cs.unm.edu

Abstract

Diversity is an important source of robustness in biological systems. Computers, by contrast, are notable for their lack of diversity. Although homogeneous systems have many advantages, the beneficial effects of diversity in computing systems have been overlooked, specifically in the area of computer security. Several methods of achieving software diversity are discussed based on randomizations that respect the specified behavior of the program. Such randomization could potentially increase the robustness of software systems with minimal impact on convenience, usability, and efficiency. Randomization of the amount of memory allocated on a stack frame is shown to disrupt a simple buffer overflow attack.

1 Introduction: Diversity is valuable

Diversity is an important source of robustness in biological systems. A stable ecosystem, for example, contains many different species which occur in highly-conserved frequency distributions. If this diversity is lost and a few species become dominant, the ecosystem becomes susceptible to perturbations such as catastrophic fires, infestations, and disease. Similarly, health problems can emerge when there is low genetic diversity within a species, as in the case of endangered species or animal breeding programs. The vertebrate immune system offers a third example, providing each individual with a unique set of immunological defenses, helping to control the spread of disease within a population.

Computers, by contrast, are notable for their lack of diversity. Manufacturers produce multitudes of identical copies from a single design, with the goal of making every hardware and software component identical. Beyond the economic leverage provided by the massive cloning of one design, such homogeneous systems have other advantages: They behave consistently, application software is more portable and more likely to run identically across machines, debugging is simplified, and distribution and maintenance tasks are eased. Standardization efforts are a further example of the almost universal belief that homogeneity is beneficial.

As computers increasingly become mass-market commodities, the decline in the diversity of available hardware and software is likely to continue, and as in biological systems, such a development carries serious risks. All the advantages of uniformity become potential weaknesses when they replicate errors or can be exploited by an attacker. Once a method is created for penetrating the security of one computer, all computers with the same configuration become similarly vulnerable. The potential danger grows with the population of interconnected and homogeneous computers.

In this paper we argue that the beneficial effects of diversity in computing systems have been overlooked, and we discuss methods by which diversity could be enhanced with minimal impact on convenience, usability, and efficiency. Although diversity considerations affect computing at many levels, here we focus primarily on computer security, and our emphasis is on diversity at the software level, particularly for operating systems, which are a common point of intrusion.

Computer security is a growing concern for open computing environments. Malicious intrusions are multiplying as huge numbers of people connect to the Internet, exchange electronic mail and commercially valuable data, download files, and run computer programs remotely, often across international boundaries. Traditional approaches to computer security—based on passwords, access controls, and so forth—are ineffective when an attacker is able to bypass them by exploiting some unintended property of a system. Finding ways to mitigate such attacks is likely to be an increasing concern for the operating systems community.

Deliberately introducing diversity into computer systems

[†]Current address: MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139.

can make them more robust to easily replicated attacks. More speculatively, it might also enhance early detection of timing problems in software and other faults. Today, each new discovery of a security hole in any operating system is a serious problem, because all of the installed base of that operating system—thousands, if not millions, of machines, running almost exactly the same system software—may well be vulnerable. An attack script developed on one machine is likely to work on thousands of other machines. If every intrusion, virus, or worm had to be crafted explicitly to a particular machine, the cost of trying to penetrate computer systems would go up dramatically. Only sites with high-value information would be worth attacking, and these could be secured using stronger methods. The relevance of diversity to computer security was recognized as early as 1989 in the aftermath of the Morris Worm, when it was observed that only a few machine types were vulnerable to infection [2]. Yet, this simple principle has not been adopted in any computer security system that we know of.

2 Strategy: Avoid Unnecessary Consistency

Our goal is to prevent widespread attacks by making intrusions much harder to replicate. Can we introduce diversity in a way that will tend to disrupt malicious attacks—even through security holes that have not yet been discovered—without compromising reliability, efficiency, and convenience for legitimate users? We believe that the answer is yes, because computers today are far more consistent than necessary. For example, all but the lowest-level computational tasks are now implemented in a high-level programming language, and for each such program there are many different translations into machine code that will accomplish the same task. Each aspect of a programming language that is "arbitrary" or "implementation dependent" is an opportunity for *randomized compilation* techniques to introduce diversity. Here we extend the term "compilation" beyond its usual meaning to include both load- and execution-time transformations [1]. Such diversity would preserve the functionality of well-behaved programs and be highly likely to disrupt others by removing unnecessary regularities. We refer to the strict virtual machine implied by a programming language's semantics as "the box." As far as possible all functional properties not required by a language's semantics should vary across individuals, a principle that we refer to as "surrounding the box with noise." In short, when a property is described by a programming language as "arbitrary," that should mean "random," not "unspecified but usually constant."

We have adopted the following guidelines to help us identify the most promising directions to explore:

1. Preserve high-level functionality. At the user level, the behavior of different systems should be predictable, and the input/output behavior of programs should be identical on different computers.

2. Introduce diversity in places that will be most disruptive to known or anticipated intrusion methods.

3. Minimize costs, both run-time performance costs and the cost of introducing and maintaining diversity. We believe that the latter is likely to be related directly to where the variations are introduced in the software development process. A load-time modification is likely to be less expensive than a compile-time modification which in turn is less expensive than requiring a developer to write multiple versions of application code.

4. Introduce diversity through randomization. Techniques based on prior knowledge of the semantics of the property being varied would also be possible, but they are unlikely to scale as well as methods based on randomization.

3 Possible Implementations

There are a wide variety of possible implementation strategies for introducing diversity. In this section, we discuss several of these and their implications for security. Our emphasis is on variability that can be introduced into software between the time that the software is written and when it is executed, and as we mentioned earlier, we believe that variations introduced late in the compilation process are most likely to be successful. The expense of producing a unique executable for every different machine is high, and there are many ways that variations could be introduced after an executable is written. In our initial explorations, however, we cover as many different kinds of transformations as possible. We consider methods ranging from those that produce variability in the physical location of executed instructions, the order in which instructions are executed, the location of instructions in memory at run-time, and the ability of executing code to access external routines, files, and other resources.

3.1 Adding or deleting nonfunctional code

Perhaps the simplest method is to insert no-ops or other nonfunctional sequences of instructions at random locations in compiled code. Depending on the architecture, this could potentially affect timing relations at execution-time and would slightly change the physical location of instructions. It would also interact with compiler optimizations that insert no-ops to preserve cache alignment, but it might be possible to insert the nonfunctional code in such a way as to respect cache alignment constraints.

The timing attacks reported on RSA [3] could potentially be disrupted using this method, although other remedies for this particular attack have also been proposed.

3.2 Reordering code

Optimizing and parallelizing compilers use many techniques to improve performance, and some of these could be used to generate code variations. For example,

1. Basic blocks: Rearrange the basic blocks of compiled code in random order. This would cause instructions to be stored in different locations but would not affect the order in which they are executed. However, basic-block placement is an important performance optimization [6], so the impact on execution-time efficiency for this method is likely to be large.

   Basic-block rearrangements could potentially disrupt some viruses. However, most file-infector viruses insert a single jump instruction that transfers control to the virus code (stored at the end of the program), and then return control to the original program. Thus, rearranging basic blocks in the program segment would be unlikely to affect this large class of viruses.

2. Optimizations for parallel processing: Many techniques exist for producing blocks of instructions that can be run simultaneously on multiple processors. These techniques could be applied to code intended for execution on a single processor, resulting in a unique order of execution. We do not know what if any intrusion methods this would disrupt. Further, the amount of variability that could be produced with this method would be limited to the amount of parallelism that could be extracted from the original program.

3. Instruction scheduling: Vary the order of instructions within a basic block, while respecting the data and control dependencies present in the source code. A preliminary study of the source code for the Linux kernel concluded that the number of different orderings that could be automatically generated was very high [5]. As in the case for basic-block rearrangements, interactions with code optimizations would need to be considered carefully to avoid serious degradations of execution-time performance.

3.3 Memory layout

There are standard ways of allocating memory when programs execute and of ordering the components of memory. These are arbitrary and could be varied in many ways. Here are a few examples:

1. Pad each stack frame by a random amount (so return addresses are not located in predictable locations). The amount of padding could be fixed for each compilation and varied between compilations, or it could be varied within a single compilation.

2. Randomize the locations of global variables, and the offsets assigned to local variables within a stack frame.

3. Assign each newly allocated stack frame in an unpredictable (e.g., randomly chosen) location instead of in the next contiguous location. This would have the effect of treating the stack as a heap, which would increase memory-management overhead. Many functional languages have this capability for constructs such as closures.

Some of these memory-layout schemes would likely disrupt a pervasive form of attack—the buffer overflow—in which an input buffer is intentionally overflowed to gain access to an adjacent stack frame.

There are several potential complications, however, including whether and how to preserve Application Binary Interface (ABI) compatibility, preserving the correct functionality for certain user functions (e.g., the C function "alloca"), and how to maintain compatibility with dynamic libraries. In spite of these complications, we consider memory-layout modifications to be a promising initial direction, because buffer overflows are such an important path of intrusion.

3.4 Other transformations

1. Process initialization: Instructions that are executed before user code could be varied. Such changes could involve varying object files such as `crt0.o` that are linked into every executable and are responsible for calling `main`. Alternatively, it would be possible to introduce variations in the kernel (e.g., in `execve`) such that data locations (e.g., command-line arguments and environment variables) are randomized.

2. Dynamic libraries and system calls: For a program to run on different machines, it must know the correct names and arguments for dynamic library routines and system calls. By varying names and permuting arguments, binaries could be made machine-specific. An importation process could also be developed that would allow users to convert foreign binaries into the local format. Such changes would make it much harder for viruses and worms to propagate.

3. Unique names for system files: Varying the names of common system files so they are difficult for intruding code to find would be highly effective against attacks

targeting these files. However, such changes would complicate system administration unreasonably unless authorized administrators were provided with a secure interface under the inverse mapping (from the randomized names back to their standard counterparts).

4. Magic numbers in certain files, e.g., executables: The type of information contained in many files can be (at least tentatively) identified by searching for characteristic signatures at the beginning of the file. Individual systems could re-map such signatures to randomly chosen alternatives and convert the signatures of externally obtained files via an explicit importation process.

5. Randomized run-time checks: Many successful intrusions could be prevented if all compiled code performed dynamic array bounds checking. However, such checks are rarely performed in production code because of perceived performance costs. Instead of requiring every program to pay the cost of doing complete dynamic checking, each executing program could perform some of these checks (potentially a very small number of them). Which checks were to be performed could be determined either at compile-time or at run-time.

4 Preliminary Results

As an initial demonstration of these ideas, we have implemented a simple method for randomizing the amount of memory allocated on a stack frame and shown that it disrupts a simple buffer overflow attack (item 1 from Section 3.3). Buffer overflow attacks arise because many programs statically allocate storage for input on the stack, and then do not ensure that their received input fits within the allotted space. Because C does not require array bounds to be checked dynamically, overflows can result in the corruption of variables and return addresses. Buffer overflows are problematic in the context of programs that run as root in UNIX, primarily because they provide a way for a non-privileged user to obtain root access. However, any script exploiting such vulnerabilities is brittle. To overwrite the return address, the distance between the start of the buffer and the function's return address on the stack must be known as well as the exact location of the code to be executed.

If every compilation produced an executable with a different stack layout, then exploit scripts developed on one executable would have a low probability of success on other executables. To change the layout of the stack, we increase the size of the stack frame by a random amount, by adding a random amount of space to certain stack slots. Such additions affect both the stack layout for the modified function and the exact locations of every function called by it. To implement this, we made a small modification to `gcc` (version 2.7.2.1), so that it adds a random number of bytes to any stack allocation request larger than 16 bytes, where the number of extra bytes is randomly selected to be between 8 and 64 in increments of 8. That is, on each new stack allocation request (above the 16-byte threshold), a random number is selected (one of 8, 16, 24, ..., 64) which designates the number of bytes of padding for that call. This gives one example of how a compiler could help users create unique systems, which are vulnerable to attack but vulnerable in ways different from every other computer.

The idea behind the 16-byte threshold is to minimize the amount of unnecessary padding. Because the buffer overflow technique requires a relatively large buffer in which to store the intrusion, it is unnecessary to pad stack allocations smaller than some threshold. We have not experimented with different threshold sizes but chose one that we believe is well below the threshold needed for buffer-overflow attacks.

The revised version of `gcc` produces a program that disrupts a simple buffer overflow attack against `lpr` on Linux 2.0.28, Debian Linux 1.1 [4]. This attack works by giving `lpr` a large argument for the `-C` (class) command-line switch. In the function `card`, `lpr` copies the command-line argument into a fixed size local buffer causing an overflow. As a result, `card` transfers control to the original copy (located in `argv`), which execs a shell running as root. This attack is disrupted by changing the size of the buffer, preventing `card`'s return address from being overwritten.

These modifications have a relatively small impact on execution-time performance. In tests of `gzip` and `gcc` (compiled by both the modified and unmodified versions of `gcc`) the differences in CPU time were negligible. Because our modifications cause a program to expand its use of stack memory, we expected some reduction in performance, which testing on additional programs might reveal. An important question is how much extra stack space is required for this method to be effective. In terms of static stack space, the answer appears to be 10-15%. In the `gzip` example, 17 slots exceed the 16-byte threshold (thus, qualifying for modification) out of a total of 125. Notably, these 17 slots consume most of the stack usage for the program (93%). Similarly, in the case of `gcc`, 313 slots exceed the threshold, out of 6183 total. The 313 slots account for 64.7% of the stack usage.

There are several parts to a buffer-overflow attack: (1) overflowing the original buffer to gain access to a return address, (2) transferring control to a known location containing intrusive instructions, and (3) executing the intrusive instructions. The stack-frame variations we described affect the first of these but not necessarily the second. For example, in Linux, command-line arguments passed to argv are stored in a predictable location determined by the kernel and are not affected by stack-frame modifications. The contents of argv are later copied into a stack frame (this is the buffer that is targeted for the overflow), but the attacker has the option of transferring control to the original copy (stored in a highly predictable location). Although our method successfully disrupts the overflow and subverts the attack, these considerations suggest yet another possible randomization—one that we plan to explore in future work.

5 Impact on Computer Security

Here we give a brief overview of common security problems and our assessment of which diversity methods would be most effective against them. Unfortunately, assessing and documenting the most common routes of intrusion is difficult: (1) new routes of intrusion are continually being discovered, (2) old routes of intrusion are sometimes patched, (3) there are few if any reliable statistics on successful intrusions, and (4) there is a distinction between the variety of intrusion methods and the frequency with which they are exploited.

Software errors (e.g., buffer overflows, insecurely processing command-line options, symlink errors, temp file problems, etc.) lead to several common forms of attack. Memory-layout variations, such as the one we implemented, would primarily affect buffer overflows. A race condition is an interaction between two normally operating programs via some shared resource (often, a file). Compilation techniques, such as the ones we have discussed, are unlikely to prevent race conditions. However, diversity at the level of the shared resource would likely be effective. For configuration problems (e.g., setup errors in how a service is provided or file permission problems), unique naming of system files would be highly effective. Denial-of-service attacks are sometimes due to software errors and sometimes due to lack of resource checking or poor policies. Thus, one diversity technique alone is unlikely to address all denial-of-service problems. For problems associated with insecure channels (e.g., IP spoofing, terminal hijacking, etc.), we expect that cryptography techniques are probably more helpful than diversity techniques, at least for diversity generated on a single host. Trust abuse, including key management problems and inappropriately trusted IP addresses, could be addressed by generating a unique profile of each computer's behavior and using it to establish identity. A final security problem that has been well-studied is that of covert channels. It might be possible to introduce diversity to prevent exploitation of covert channels, although we have not studied it well enough to have specific suggestions.

Within computer security there is widespread distrust of "security through obscurity"—for example, proprietary cryptographic algorithms that are kept secret on the grounds that publishing their algorithms would weaken their security. Such distrust is warranted—proprietary cryptographic algorithms, once revealed, often turn out to have serious flaws. Nevertheless, it is worth noting that at the level of whole systems, all security is ultimately based on making some aspect of the system obscure, whether it be passwords or private keys. Possession of a secret is the basis for granting differential access. By randomizing implementation dependencies, our approach can be thought of as adding a new level of automatically-generated "secrets" that are transparent to properly functioning code, but which misbehaving code must possess to crack the system successfully. Further, at the level of algorithms, our approach would actually reduce obscurity by eliminating obscure implementation-dependent consistencies of which the algorithm designer was unaware and certainly did not intend, but which, once discovered, might form the basis of an attack.

For some security applications it is important to certify that a computer system is trustworthy, through a combination of proving formal properties about the specification and testing and analyzing the implementation. Although the method we propose would complicate the testing procedure, any system that stayed within its formal specifications ("in the box") would be robust to variations outside the box. Thus, an implementation that successfully withstood random variations of the sort we propose would be more trustworthy than one that did not.

6 Conclusion

Diversity techniques such as those we have proposed here can serve an important role in the development of more robust and secure computing systems. They cannot, by themselves, solve all security problems, because many exploitable holes are created completely "within the box" of a program functioning under the semantics of the language in which it is written. And indeed, diversity techniques may sometimes disrupt legitimate use by unmasking unintended implementation dependencies (i.e., "bugs") in benign code. Nonetheless, the essential principles of diversity—"avoid unnecessary consistency," and "surround the box with noise"—express a strategy that is likely to find use in the computers of the future.

This approach can only be successful if it is low-cost, having minimal impact on run-time efficiency and maintainability. In this paper, we have concentrated on outlining a wide variety of possible approaches, to stimulate further ideas and suggestions. An important area of future research is to assess these and other ideas more systematically to determine which ones are worth implementing.

Acknowledgments

Over the past couple of years we have discussed the general idea of diversity with many people and solicited their comments and ideas for possible implementation strategies. In particular, A. Davis, T. Knight, B. Maccabe, M. Oprea, M. Seltzer, H. Shrobe, E. Stoltz, G. Sussman, and C. Young have all listened with more or less open minds and made helpful suggestions. The authors gratefully acknowledge support from the National Science Foundation (grant IRI-9157644), the Office of Naval Research (grant N00014-95-1-0364), Defense Advanced Research Projects Agency (grants N00014-96-1-0680 and N66001-96-C-8509), the MIT AI Lab., Interval Research Corp., and the Santa Fe Institute.

References

[1] J. B. Chen, M. Smith, and B. N. Bershad. Morph, a framework for platform-specific optimization. Technical Report TR-04-96, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA, 1996.

[2] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the IEEE Symposium on Research in Computer Security and Privacy*, Los Alamitos, CA, 1989. IEEE, IEEE Computer Society Press.

[3] E. English and S. Hamilton. Network security under siege: the timing attack. *Computer*, March 1996.

[4] V. Kolontsov. Bugtraq mailing list, Oct. 25, 1996. Linux & BSD's lpr exploit.

[5] M. Oprea. Towards compiler-induced object code variability. Unpublished Manuscript, June 1996.

[6] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, (to appear).