**SANDIA REPORT**

SAND2010-7055
Unlimited Release
Printed October 2010

# The Theory of Diversity and Redundancy in Information System Security: LDRD Final Report

Benjamin A. Allan, Robert C. Armstrong, Jackson R. Mayo, Lyndon G. Pierson,
Mark D. Torgerson, Andrea Mae Walker

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

| | |
|---|---|
| Telephone: | (865) 576-8401 |
| Facsimile: | (865) 576-5728 |
| E-Mail: | reports@adonis.osti.gov |
| Online ordering: | http://www.osti.gov/bridge |

| | |
|---|---|
| Telephone: | (800) 553-6847 |
| Facsimile: | (703) 605-6900 |
| E-Mail: | orders@ntis.fedworld.gov |
| Online ordering: | http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online |

SAND2010-7055
Unlimited Release
Printed October 2010

# The Theory of Diversity and Redundancy in Information System Security: LDRD Final Report

Jackson R. Mayo
Scalable Modeling & Analysis Systems

Robert C. Armstrong Benjamin A. Allan
Scalable & Secure Systems Research

Sandia National Laboratories, P.O. Box 969, Livermore, CA 94551-0969

Mark D. Torgerson Andrea Mae Walker
Analytics & Cryptography

Lyndon G. Pierson (retired)
Networked Systems Survivability & Assurance

Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-0672

**Abstract**

The goal of this research was to explore first principles associated with mixing of diverse implementations in a redundant fashion to increase the security and/or reliability of information systems. Inspired by basic results in computer science on the undecidable behavior of programs and by previous work on fault tolerance in hardware and software, we have investigated the problem and solution space for addressing potentially unknown and unknowable vulnerabilities via ensembles of implementations. We have obtained theoretical results on the degree of security and reliability benefits from particular diverse system designs, and mapped promising approaches for generating and measuring diversity. We have also empirically studied some vulnerabilities in common implementations of the Linux operating system and demonstrated the potential for diversity to mitigate these vulnerabilities. Our results provide foundational insights for further research on diversity and redundancy approaches for information systems.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

All software and hardware has an intended implementation, and often works reasonably well in its intended context. The problem is that the same hardware and software will behave in other ways outside of this intended context. Many of these behaviors are benign, some are faults, and some are vulnerabilities. This is a result of the inherent complexity of software and hardware in common use today. This complexity is a necessary artifact of the complicated tasks we require them to perform.

## 1.1 Defining Complexity

Too often complexity is taken to mean "profoundly complicated" or sometimes that which is simply complicated. This ephemeral definition suggests a sliding relative scale where one item can be said to be "more complex" than another subjectively. A more satisfying and exact approach is needed. For our purposes we will claim that a necessary condition for a system to be called formally complex is that the computational capability of the system is at least Turing complete. This definition coheres with a few of the properties that we look for in complex systems and that bear directly on cybersecurity:

1. Turing complete systems cannot be decided (i.e., predicted, or said to be bounded) ahead of running them to see what they do (Turing's Halting Problem and Rice's Undecidability Theorem [14]). This *irreducibility* is a hallmark of emergent behavior in complex systems. While this definition probably extends to non-cyber complex systems (as in biology and economics), it is clearly true for computers and networks of computers[1].
2. A complex system's emergent behavior cannot be predicted, short of "running it", from even a perfect knowledge of the constituent parts that compose the complex system – a ramification of undecidability.

This does not mean that there are not programs that are simple enough to be analyzed by formal methods and other tools. In those cases, boundedness properties of the code can be asserted

[1]Real computers are not strictly Turing complete because they have finite memory, but for any execution that "fits" in the computer's memory the question is moot.

**Figure 1.1.** Illustration of the asymmetry in cyber-defense.

and proved, making the behavior well-understood under a wide variety of circumstances. Formal verification [4] is accomplished by automatically following all salient execution paths to understand their consequences. However, probably the vast majority of codes are of the sort that are too complex for this analysis, the number of paths growing beyond the capacity of even the largest machines. It is this latter case on which we concentrate in this work. In this work we consider systems that are undecidable and for which vulnerabilities can only be discovered anecdotally but not thoroughly. Because their containing programs are unanalyzable, these vulnerabilities cannot be guaranteed to be found by any means.

The task we undertake is to reduce the hazard of vulnerabilities while recognizing that such vulnerabilities in most programs of interest are unknowable. Because of the enforced opacity of specific vulnerabilities, we hypothesize that any successful method must rely on statistical properties involving a real or conceptual ensemble of programs. Any other method would, by definition, require specific foreknowledge of the vulnerability, violating the undecidability principles. Diversity in the ensemble provides the means to detect an attack, leveraging the exploit itself to reveal the presence of a vulnerability without knowing the specifics of the vulnerability *a priori*.

## 1.2 Properties of Vulnerabilities

In general, then, it is impossible to find an arbitrary vulnerability in an arbitrary code [14]. This imparts a significant advantage to the attacker over the defender: The attacker needs to find only one vulnerability to subvert code or a machine, while the defender must patch *all* of the vulnerabilities to be safe (Figure 1.1). Due to the unavoidable undecidability of vulnerabilities in a general code, the defender is always in jeopardy. Given that we would like our results to be general, and thus apply to general hardware and software systems, some property must be found to either decrease the uncertainty of the defender or increase the uncertainty of the attacker. This work centers on exploiting diversity in software and hardware to accomplish this goal.

An implementation in hardware and software is not unique and there exists a diverse, possibly infinite, set of implementations from which to choose. The semantics and capability of constructing hardware and software are sufficiently powerful and redundant that there are myriad ways of accomplishing the same result. Usually vulnerabilities are an artifact of choosing one of many ways of accomplishing the specified purpose of the program or hardware. Diversity of implementation will never completely eliminate vulnerabilities, because of the general undecidability of untested characteristics of the implementation, but a random sample from a complete set of implementations will make it *uncertain* whether the sampled implementation has a particular flaw. Vulnerabilities unwittingly imparted to an implementation (hardware or software) are assumed to have two properties:

1. Vulnerabilities are incidental and not necessitated by the requirements and specification of the desired hardware or software. All vulnerabilities are, in principle, able to be eliminated once they are found. It is the impossibility of finding them that creates the problem.
2. Vulnerabilities are not an artifact of the environment with which the hardware/software is created or in which the hardware/software executes. There is no interaction between the compilers or infrastructure with the implementation itself that produces a vulnerability for all possible implementations.

If either of these conditions is not met then *every* implementation will have a vulnerability of some sort because either the implementation's specification or its environment require that the vulnerability be present. Our assumptions set the beginning of a mathematical foundation for understanding vulnerabilities and possible avenues toward mitigating their effects.

# Chapter 2

# The Meaning and Promise of Diversity in Hardware and Software

As discussed in Section 1.2, unknown vulnerabilities, like any property of software, are unforeseeable until they are revealed by an exploit. Because a vulnerability is just a fault in the software that can be exploited by an adversary, it is useful to examine the more fertile field of fault-tolerant computing. We will first define what is meant by diversity in hardware/software systems and then discuss ways to achieve it. In general, there is no "one size fits all" form of diversity that works for all possible vulnerabilities. We will find that vulnerabilities are best thought of as members of *classes* and that certain mechanisms for creating diverse software will be effective against all vulnerabilities in a class but leave other classes of vulnerabilities untouched.

## 2.1 Observation 1: Many Implementations for a Single Design

It is useful to observe that for most feature sets there is a large number of implementation programs – an infinite number if there is no other restriction. If as a part of the feature set we bound the implementation program size, then we can define a finite set of all implementation programs for that particular feature set. The total number of such programs is related only to the properties of the designed feature set and the program size bound. If, for example, the bound is set below the Kolmogorov complexity analogue of the feature set (i.e., the smallest program implementing the feature set) then the set of implementations is empty. The implementation "entropy" of the feature set $F$ can be considered to be $S_F = \log n(F)$, where $n(F)$ is the total number of possible implementations of feature set $F$ – a measure of how diverse $F$ is. This definition is consistent with the definition of entropy as "the number of ways of changing the inside such that the outside stays the same".

Assuming that $F$ itself does not require vulnerabilities in every implementation, we can be certain that within the complete set of implementations $I_F$, each will have diverse vulnerabilities not shared by all (Figure 2.1). It is important to note that if one vulnerability is shared by all possible implementations, this is the same as the feature set *requiring* a vulnerability: The designed feature set itself has a vulnerability. In this way we can unambiguously say that each vulnerability has a probability less than unity of being found in any member of $I_F$. To the extent that no implementation vulnerability is "encouraged" by the feature set (i.e., made more probable than any other

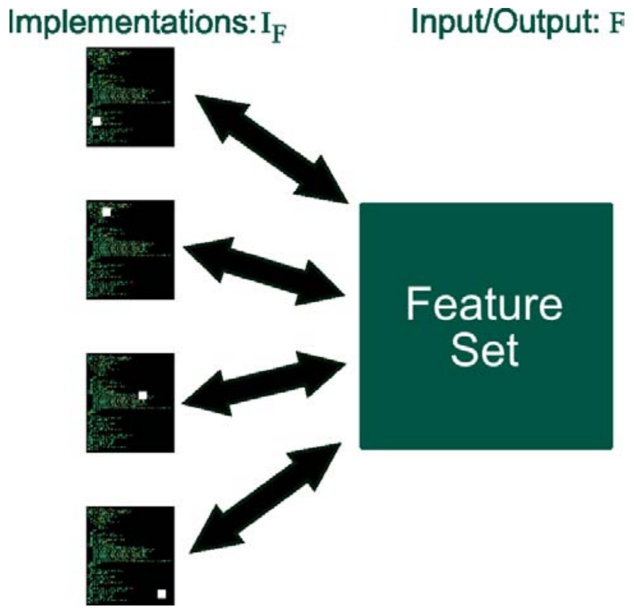

**Figure 2.1.** A feature set and its implementations. Here the "feature set" ($F$) is the collection of designed inputs and outputs that a program produces. There are many possible implementations of a given feature set, each different in their unrelated implementation particulars ($I_F$). Assuming that the design of the feature set is sound, security "holes" (white dots) in programs happen as a result of the implementation only, and not the feature set.

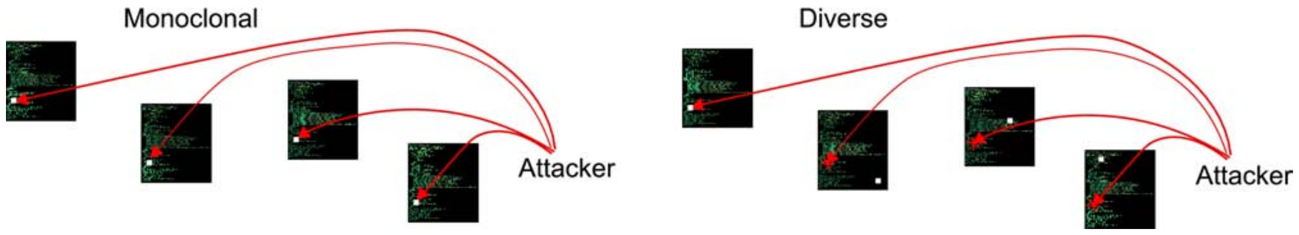

**Figure 2.2.** Identical versus diverse implementations. Turing completeness and undecidability cause security holes to be unknowable in the general case to programmers, users, and attackers. Implementations that are identical (monoclonal) are equally susceptible to the discovery of a security hole by an adversary. Diverse implementations are less so, depending on the number of variants.

vulnerability), vulnerabilities can be considered incidental and can be assumed to be uncorrelated. Since we stipulate that this software is complex, providing the opportunity for many different types of vulnerabilities, we can speculate that any particular vulnerability will be rare.

## 2.2 Observation 2: Ensembles of Undecidables Yield Meaningful Statistics

A second observation is that although every implementation is undecidable with regard to possible vulnerabilities, an ensemble of implementations may permit meaningful quantitative statistics. Consider an ensemble of implementations chosen from $I_F$. If each of these members is run with the same input history, then nominally all of the responses will be the same. If, as part of that input history, an exploit for a particular vulnerability is present, it will likely succeed only for a minority of the ensemble (Figure 2.2). It is clear that statistics can be formed relating the likelihood of compromise for a specific fraction of the ensemble.

Even though the vulnerabilities of each member implementation are unknowable, the ensemble itself is more predictable. We can say what percentage of the ensemble is vulnerable to attack and with what likelihood given the statistics of the implementation set. This property of ensembles of undecidables can be used to turn complexity against the attacker and in favor of the defender. A number of possibilities exist for exploiting this property and will be discussed.

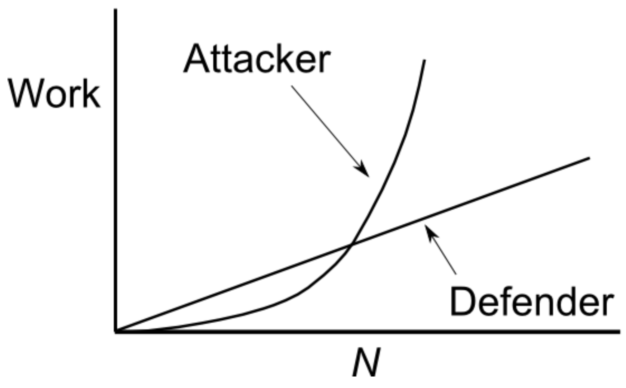

**Figure 2.3.** Attacker and defender effort for a diverse voting system. An idealized graph compares the linear scaling of defender effort with the exponential scaling of attacker effort for a system that votes *N* diverse software implementations.

## 2.3 N-Version Technique Adapted to Cybersecurity

An obvious application of ensembles is closely related to *N*-version software techniques [8] for high-reliability, fault-tolerant systems used in aerospace and other time-critical systems and recently applied to cybersecurity [10, 11]. Here we are seeking to identify wrong responses from an otherwise correct implementation [3]. In this technique, different versions implementing the same feature set compare responses to detect a fault and vote the collective response of the ensemble. In the past this concept has been restricted to fault-tolerant systems. Typically, practical application of *N*-version software requires that all versions be hand-coded in a Chinese-wall style where different programmers are given the specification and have no other communication. This extra effort is deemed worthwhile for critical control systems, as in spacecraft or aircraft, where the control software is fairly simple. The expense presents a problem for more complex software, where getting just one version is burdensome.

However, the benefits of using many diverse implementations can be very substantial (Figure 2.3), based on the theoretical scaling of the work required for a defender to produce such variants (linear) in comparison to the work required for an attacker to defeat a robust voting system by finding a *shared* vulnerability among most of the variants (exponential). Although in practice the variants will not be fully randomized as assumed in this comparison, the voting mechanism can still provide an increase in robustness as long as some diversity is available.

There are other differences in generalizing these ideas into the cybersecurity arena. Faults in the *N*-version technique are not considered hazardous to the overall system and faulting versions are considered benign as long as they are in the minority and lose the vote. In the case of a cyber attack, however, the fault becomes an exploited vulnerability and the compromised implementation must be removed from the ensemble or it will pose a danger to the entire system. Because of this necessity, it would be advantageous to generate new implementations automatically that will not repeat the same vulnerability. An automatic means for generating new implementation versions from one or a few existing implementations is needed. In the following section, we sketch a system for accomplishing these goals using a simple genetic algorithm for finding new implementations and a voting scheme for eliminating outliers.

## 2.4 Genetic Programming Techniques for Code Transformation

Generation of software variants by hand is time-consuming and expensive. Furthermore, people tend to approach tasks in similar ways and make similar mistakes, limiting the software diversity that can be achieved through human coding, even by separated developers [3]. On the other hand, the field of genetic programming has the ambitious goal of enabling software to be created automatically based on a specified feature set (objective function), by imitating the biological processes of mutation and natural selection; but such an approach has not been successful in creating realistically complex software from scratch.

We propose that an effective method for generating diverse software implementations is to start with an initial human-written implementation that passes the test suite for its feature set, and to use the mutation techniques of genetic programming for the sole purpose of introducing variations in implementation details, while *preserving* (but not *improving*) functionality as measured by the test suite. This application of genetic programming should be more tractable because, rather than exploring an astronomically large space in search of an optimum in the fitness "landscape", we are starting at an optimum (a functional program) and merely diffusing along the manifold of such optima. Natural selection among the mutated codes need only keep them as functional as the initial one. Each mutation may contribute only a small amount of diversity, but over many generations, substantial randomization may be achievable.

Possible mutations include various "semantically invariant" code transformations, some of which are implemented in current compilers. In general, any change in machine code has the potential to alter software behavior in some observable way, if only in execution timing. But under certain programming-model assumptions, there are transformations that can be proved not to affect the behavior of interest (the semantics). In the standard C programming model, for example, the order in which data values are stored on the stack does not affect behavior, and so stack randomization is considered semantically invariant. A more detailed description of stack randomization is provided in Section 3.2.

When the assumptions of a given programming model are violated (e.g., by a buffer overflow bug), two consequences occur: A potential vulnerability is introduced (e.g., stack smashing), and the corresponding "semantically invariant" transformations acquire nontrivial effects on software behavior (e.g., versions compiled with different stack randomization will not respond identically to attack). Thus, by constructing mutations from code transformations with various types of semantic invariance, and voting the resulting ensemble, robustness to various vulnerabilities can be achieved. Furthermore, the correlation between the responses of variants to a given malicious input, and their implementation differences, can help diagnose the type of vulnerability being exposed.

## 2.5 Component-Based Software for Combinatorial Leverage

Whether by hand or by an automated genetic approach, practical difficulties may limit the number of diverse implementations that can be generated. An effective way to exploit the availability of even a relatively small number of variants is by breaking software into "components", each of which has defined functionality, with its own feature set and test suite. This component-based architecture is supported by several currently used programming models. Variant implementations of a given component, like variant alleles of a biological gene, are in principle interchangeable without impairing overall functionality. The creation of variants for each component, and the arbitrary reshuffling of these variants into complete programs, can generate a much larger number of diverse implementations of the entire program.

A potential disadvantage of the component approach is that the interfaces between components can themselves be a source of vulnerabilities that are not diversified away. Because the components' interaction pattern is fixed in this example, a portion of the initial implementation has been encoded and frozen into a more complex (and likely less analyzable) feature-set specification. We expect that the most efficient generation of diversity will involve a preferred level of decomposition for a given system, a tradeoff between the combinatorial advantage of many fine-grained components and the greater flexibility and analyzability of fewer coarse-grained components.

It is useful to consider a program that appears monolithic but actually has a component architecture that is not visible to the diversity generator; arguably most non-obfuscated human-written programs embody such an architecture in some way. A single localized mutation to the program will typically affect only one underlying component, but verifying post-mutation functionality at the monolithic level will be time-consuming because it will effectively be integration testing as opposed to unit testing. Furthermore, without recognizing the components, there is no general way to leverage existing successful variants to generate new ones. Eventually, after many nonviable tries, some (probably most) of the successful variants of the monolithic program will have the property of changing the test-induced input-output behavior of one underlying component in a way that is compensated by a change in another component – i.e., effectively changing the interface between the components. These variants, obtained at great expense, which alter the component interfaces in different ways (invisible to the diversity generator), cannot have their mutations mixed and matched with any expectation of functionality.

On the other hand, a combination of the two approaches – monolithic and component-based – may have benefits. If the monolithic approach leads to even a single integration-test-passing variant that does *not* pass all component-level tests, this variant can be considered the progenitor of a new program "species". The correct answers for the component-level tests can be adjusted to reflect the new behavior, thus implicitly capturing the interface redefinition. Then a whole new set of component versions can be created to populate the new species. Programs from different species can fully participate in a system because they have the same overall functionality. The internal interface differences might turn out to be as simple as renumbering of return status codes, or as complicated as various types of code refactoring. This technique may recapture some of the diversity lost by freezing the interfaces.

Realistic software has a hierarchical structure of components, subcomponents, etc. The question may not be the single best level of component coarseness for "annealing" software, but the best way of cycling among the various levels – reminiscent of multigrid methods.

## 2.6 Voting

To realize the power of multiple diverse implementations there must be a method of comparing outputs of the implementations. That voting mechanism must be powerful enough to account for timing differences that might occur in the execution of a task across the members of the ensemble. Depending on the complexity of the functionality implemented and the amount of diversity, these delays may be significant and may pose a number of engineering challenges for system designers.

The voting mechanism must make a decision about how to proceed when a fault is detected. It is more than just getting the "right" answer and continuing. One has to decide what to do with a process that has been deemed faulty. One can assume that if a complicated process gets out of line with the others, it may be out of line for quite a while, if not indefinitely. There are a plethora of options as to how to proceed, which may be controlled to some extent by policy and design.

From a security standpoint the voting mechanism opens holes in the system that may not otherwise be there. It has its own feature set that needs to be verified. Depending on the situation, it may be that the voter is nearly as complex as the functions that it is judging. Vulnerabilities in the voter most likely translate directly into vulnerabilities in the over all system. Further, current architectures and thinking are not set up to easily support the notion of voting, so they would have to be modified to allow a voter to operate. The effect of such modifications to existing architectures would have to be evaluated.

Devising effective voting methods, sorting through the systems engineering questions, and developing policies and procedures for such areas was not part of the scope of this work. However, it is recognized that working through these issues is a significant research effort in its own right.

## 2.7 Summary and Future Research

We provide an analysis of software as an implementation of a feature set and argue that vulnerabilities not coerced by the feature-set design are random across the possible implementations. Nonetheless, a vulnerability, indeed any property of the software not already tested out, is undecidable and undetectable except anecdotally (via code analyzers, etc.). We show, however, that one can reason statistically about a diverse ensemble of such implementations. Although the scope of the work is to understand vulnerable software theoretically, we also speculate on how such an ensemble could be created in practice and the open questions related to its efficacy.

All of these potential applications of implementation ensembles rely on achieving sufficient diversity, and here there is no helpful theory. It is unlikely that the *complete* implementation set $I_F$ is needed for any feature set $F$; any of these potential applications can function with a small number of sufficiently diverse members. This brings up several research questions:

1. What is a valid metric for software diversity in this case? Here we seek a sort of "Hamming distance" for differing implementations of the same feature set.
2. What is sufficient diversity to foil an attack with some probability? This is likely dependent on the *class* of vulnerability as well as the inherent entropy $S_F$ of the feature set.
3. What automated means can be found to create new diverse implementations? Genetic programming holds out little hope for finding implementations of complex software *ab initio*, but there may be schemes to create diversity using one or a few hand-coded implementations as a starting point.
4. Under what circumstances is it possible to design and implement an effective voting mechanism?

The simple genetic algorithm example we present answers each of these questions in one way or another, with varying degrees of adequacy, and serves as a focus of discussion.

The same undecidability constraints that govern the ability to find vulnerabilities in a given implementation also govern the ability to fully measure how different two implementations of the same feature set are. So it is clear that having a perfect measure of diversity is not realistic. However, given details on the method that provides the diversity, one certainly should be able to measure what is done to the implementations. This may provide insight into the larger question of "distance". It may also be able to provide bounds for the value of the diversity as applied to the system.

A reasonable question, outside the scope of this work, is whether there is another construct beyond ensembles that might also circumvent the undecidability of complex software. It is a curiosity that ensembles in statistical physics are a theoretical artifice to reduce an overabundance of information into global averaged properties such as temperature and pressure. Conversely, here each member of the ensemble is unknowable but some relative information is gained by looking at them collectively. An interesting future investigation might leverage this observation to seek constructs other than ensembles that might have similar properties and arrive at them more straightforwardly.

# Chapter 3

# Exploiting Diversity in Hardware and Software

## 3.1 Types of Diverse Implementations

Because realistic software and hardware cannot be fully analyzed, vulnerabilities will inevitably exist. We seek to make them difficult to find and exploit, by making system behavior maximally unpredictable in aspects orthogonal to intended functionality. Then attacks to induce specific deviations from intended functionality will be unlikely to succeed. This ideal can be approached in stages involving different levels of functionality (i.e., semantics). Randomized implementations can be further leveraged by composing several into a "voting" system that compares outputs for a given input to detect and recover from attacks. The power of leveraging complexity against attackers lies in its effectiveness against not only vulnerabilities that are known, but those yet unknown.

Computer systems can be decomposed into the intended functionality and the orthogonal implementation incidentals:

- **The design** is the expected behavior and useful work that the application is designed to do. As one would expect, designed-in vulnerabilities are rare but *do* happen: The DNS specification was found to, in effect, require a vulnerability in 2008 [6]. Every faithful implementation of DNS had it. An automated system cannot protect against a system that is performing as designed, however ill-conceived.
- **The implementation incidentals** are most often the target of successful cyber attacks. The implementation complexity is larger, often much larger, than the designed user interface, offering many targets of opportunity. In hardware, a vulnerability is manifested as a transition that is enabled by an untested state (e.g., the famous Intel f00f bug [13]); in software, it may occur as an unguarded buffer overrun (e.g., the Sun RPC bug [5]), or a flaw in a network protocol implementation (again the Sun RPC bug [5]). In each case there was nothing intrinsically wrong with the design of the system; only the implementation provided the attacker the opportunity.

Presenting attackers with a "moving target" of time-varying implementation incidentals reduces their ability to count on these details for exploitation. A key question is where in software, hardware, and networking this implementation diversity is to be found.

- **Software:** Programming languages use semantics to specify behavior but leave machine-level implementation details to compilers. Vulnerabilities that depend on these details are "extra-semantic" and can be mitigated by randomization. For example, in C, stack randomization is a known defense against buffer overflow exploits. But typical SQL injection vulnerabilities are not extra-semantic in C – any compilation is equally vulnerable. SQL injection could become extra-semantic if SQL statements and user-supplied strings had unrelated character encodings. More generally, it will be useful to investigate program-obfuscation transformations that alter semantics within a language but preserve a higher level of semantics.

- **Hardware:** Many concepts for achieving a "moving target" in software implementation also apply to hardware, which is frequently designed in VHDL before "compilation" to blueprints (e.g., masks). Diversity could be applied at the time of fabrication (wafer-by-wafer or die-by-die); and increasing use of reconfigurable chips will support more convenient switching of designs during hardware lifetime. Randomized chip layouts would frustrate attacks on physical circuit locations. Processor support for instruction set randomization preserves assembly-language semantics but varies the encoding, making the results of a binary code injection attack unpredictable (extra-semantic).

- **Networks:** Techniques for implementation diversity and robustness in communications include frequent dynamic network reconfiguration, multipath routing, and use of multiple differently implemented protocols in each layer. Dynamic permutation of addresses and protocols would frustrate attacks that depend on consistent internal responses, while maintaining functionality needed by legitimate users. Sandia's Emulytics capability will allow testing these concepts on highly realistic virtualized networks of $10^6$ or more virtual machines.

A diverse set of implementations or configurations can be used against the attacker by changing out the particular instance drawn from that set (inducing time-varying uncertainty for the attacker) or running many instances in parallel (inducing space-varying uncertainty for the attacker). For example, a software program may be created with a vulnerability planted in the supply chain. If the programming model admits a changing implementation over time, this vulnerability can be "annealed" away regardless of whether the vulnerability was detectable in the first place. Alternatively, by comparing outputs of parallel instances, not all sharing the same vulnerabilities, an attack can be detected and a vote can allow execution to continue unaffected. Inspired by redundant hardware designs, these voting systems show promise for exponential reduction in likelihood of compromise [2]. The exponential increase in difficulty for the attacker compares favorably to the linear increase in difficulty of creating and running the diverse implementations, as shown in Figure 2.3. Such approaches have the potential not merely for making unknown vulnerabilities difficult to find but for engineering them out of existence. An evolutionary approach could eliminate individual compromised implementations and thus select for robustness over time.

## 3.2 Example: Stack Randomization

Buffer overflow exploits take advantage of specific programming errors that are possible in languages such as C, where some (typically larger than anticipated) inputs cause a program to attempt to write outside the memory allocated for an object (e.g., by using an out-of-bounds array index). For this to occur means that the program is not standard-conforming and, strictly, its behavior is undefined. What typically happens is either (1) the improper write is detected by the runtime environment and triggers a segmentation fault, or (2) the write is successful and changes the values stored in locations adjacent to the object, which may have been allocated to other objects. The second case can lead to the further result that (2a) the write is harmless because the locations are either never used or subsequently initialized such that the values written have no effect, or (2b) the values written alter the subsequent operation of the program in a way that the programmer did not intend.

For software vulnerable to scenario (2b), an attacker can potentially use a buffer overflow exploit to perform memory writes that lead to dangerous program behavior, including execution of malicious code. Such an exploit is facilitated by targeting the "stack" memory used for storage of local variables in functions, which is traditionally laid out in a fixed order by a given compiler. Thus an attacker may be able to consistently overwrite a specific variable of interest by providing a suitable malicious input that the program stores in another variable.

Stack randomization is a technique for thwarting buffer overflow exploits by laying out stack memory in an unpredictable order so that an attacker cannot rely on fixed relative locations of variables. It may be possible to craft a successful exploit for any given stack layout, but no single malicious input will be successful against all layouts. The layout could be chosen randomly once for a given software installation, or it could be varied over time by recompiling. It has been noted that an attacker successively trying buffer overflow exploit variations, attempting to find the one that works with the current stack layout, will on average require only twice as many attempts if the layout is varying over time as if it is random but static [12]. This assumes that a successful attack on a given system can be carried out all at once. More sophisticated and targeted attacks may involve separate exploration, testing, and deployment stages, and may want to return to the same system repeatedly over time to gain further benefits from the exploit. These scenarios could be hampered much more substantially by re-randomization.

Stack randomization is one mechanism, already supported by compilers, for generating software diversity. This dimension of diversity is called semantically invariant because all versions have the same C semantics. They differ only in an aspect that the C standard leaves to the discretion of compilers and that does not affect the observable behavior of standard-conforming programs. Correspondingly, the buffer overflow vulnerability is called extra-semantic because it relies on the undefined behavior of non-standard-conforming programs. Semantically invariant diversity can be effective only against extra-semantic vulnerabilities. The immediate value of a redundant system of stack-randomized versions is for detection of buffer-overflow exploits, since even if one or more versions are successfully compromised by an attack, other versions are very likely to exhibit observably different responses to the same input – such as segmentation faults.

# Chapter 4

# Fault Tolerance of Diverse Implementations with Voting

## 4.1 Framework

We assume that a system (software or hardware) is designed to transform inputs into outputs. The inputs are drawn from a finite set $X$, where input $x_i$ is chosen with probability $p_i$. The input distribution may reflect naturally occurring conditions in the case of a control system, or malicious hacking attempts in the case of a network-connected device. Each input is associated with a unique correct output (where "output" means all information that the system provides, either to a user or to other systems with which it interacts). Because a realistic implementation of the system is imperfect, we define an indicator function $A(x_i)$ that equals 1 if the given implementation produces an incorrect output (a "fault" or potential "vulnerability") for input $x_i$, and 0 otherwise. Clearly the probability of encountering a fault in the implementation on any one input is

$$F = \sum_i p_i A(x_i). \tag{4.1.1}$$

If the implementation is drawn from an ensemble of implementations (e.g., reflecting the distribution of outcomes of a particular development methodology), and if brackets $\langle\ \rangle$ denote averaging over this ensemble, then the overall probability of a fault is

$$\langle F \rangle = \sum_i p_i \langle A(x_i) \rangle. \tag{4.1.2}$$

Now we consider a collection of several implementations drawn independently from the ensemble. This form of "independence" does *not* imply that faults occur independently with respect to the distribution of inputs. If two particular implementations are given the same input, then the probability that both generate a fault is

$$F_2 = \sum_i p_i A_1(x_i) A_2(x_i), \tag{4.1.3}$$

and the average of this probability over implementation pairs from the ensemble is

$$\langle F_2 \rangle = \sum_i p_i \langle A_1(x_i) A_2(x_i) \rangle = \sum_i p_i \langle A_1(x_i) \rangle \langle A_2(x_i) \rangle = \sum_i p_i \langle A(x_i) \rangle^2. \tag{4.1.4}$$

Despite the independence between $A_1(x_i)$ and $A_2(x_i)$ with respect to the ensemble of implementations, we have in general $\langle F_2 \rangle \neq \langle F \rangle^2$. In fact, $\langle F_2 \rangle - \langle F \rangle^2$ can be identified as the variance of $\langle A(x_i) \rangle$ over the distribution of inputs $x_i$, so that

$$\langle F_2 \rangle \geq \langle F \rangle^2. \tag{4.1.5}$$

That is, the overall probability of a joint fault is typically *greater* than the product of the probabilities of a fault in each implementation. The strict inequality holds as long as some inputs are "harder" for the ensemble to handle correctly than others, i.e., $\langle A(x_i) \rangle$ is not the same for all possible inputs $x_i$. These conclusions were obtained previously [8].

We would like to determine if, nevertheless, the use of a large number $n$ of implementations can substantially reduce the probability of faults that lead to incorrect behavior of the collection as a whole. A straightforward design for integrating the $n$ implementations into a single system is to send each input to all of them and use the mode (most prevalent value) of their outputs to generate the final output. This process is naturally described as "voting" and is motivated by the expectation that faults are rare. We assume that the voting process itself is always reliable.

In principle, one must specify what to do when the mode is undefined (i.e., in case of a tie).

## 4.2 Majority or Supermajority Voting

Here we adopt the conservative requirement that the winning output be given by at least $k$ of the $n$ implementations, where $k \geq \lfloor n/2 \rfloor + 1$. Depending on the choice of $k$, this is either a majority or a supermajority rule; at most one output can win by this criterion. A fault in the complete system is deemed to occur when an incorrect output wins *or* no output wins. (The resulting fault probabilities will be pessimistic in comparison to the case where a plurality can win.) A convenience of these rules is that no distinction need be made among different incorrect outputs for a given input; the only question is whether at least $k$ of the outputs are correct.

For a given input $x_i$, the output of each implementation drawn from the ensemble is independently incorrect with probability

$$a_i \equiv \langle A(x_i) \rangle. \tag{4.2.1}$$

Thus the number of incorrect outputs among the $n$ implementations is a random variable $M_i \sim \text{Binomial}(n, a_i)$. The probability of a fault for the $k$-out-of-$n$ system on input $x_i$ is

$$f_i \equiv \Pr(M_i \geq n-k+1) = \sum_{j=n-k+1}^{n} \binom{n}{j} a_i^j (1-a_i)^{n-j} = \frac{\mathrm{B}(a_i, n-k+1, k)}{\mathrm{B}(n-k+1, k)}, \tag{4.2.2}$$

where B is the complete or incomplete beta function. The lowest-order term in $a_i$ gives a simple asymptotic approximation for very rare faults,

$$f_i \sim \binom{n}{n-k+1} a_i^{n-k+1} \qquad (a_i \to 0). \tag{4.2.3}$$

More useful would be an asymptotic approximation to $f_i$ for $n, k \to \infty$ with arbitrary $a_i$. This would give the scaling behavior of the fault probability for a large collection of implementations. We assume that

$$k = \lfloor \alpha n \rfloor + 1 \tag{4.2.4}$$

for a fixed parameter $\alpha \in [\frac{1}{2}, 1)$. Note first that if $a_i > 1 - \alpha$ (i.e., the probability of a fault in each implementation is greater than the allowed fraction of incorrect outputs), then it is clear by the law of large numbers that $f_i \to 1$ as $n \to \infty$. Thus, e.g., if a particular input is such that implementations drawn from the ensemble produce incorrect output more than half the time, then majority voting of a collection of implementations is of no benefit in reducing faults for this input.

On the other hand, if $a_i < 1 - \alpha$, we have $f_i \to 0$, and the following asymptotic approximation provides an upper bound as $n \to \infty$ [7]:

$$\begin{aligned} f_i &\sim \frac{1}{\mathrm{B}(n - \lfloor \alpha n \rfloor, \lfloor \alpha n \rfloor + 1)} \frac{1}{(1 - a_i)n - \lfloor \alpha n \rfloor - 1} a_i^{n - \lfloor \alpha n \rfloor} (1 - a_i)^{\lfloor \alpha n \rfloor + 1} \\ &= \binom{n}{n - \lfloor \alpha n \rfloor} \frac{n - \lfloor \alpha n \rfloor - 1}{(1 - a_i)n - \lfloor \alpha n \rfloor - 1} a_i^{n - \lfloor \alpha n \rfloor} (1 - a_i)^{\lfloor \alpha n \rfloor + 1} \qquad (n \to \infty). \end{aligned} \tag{4.2.5}$$

Expansion of the binomial coefficient gives a more explicit asymptotic approximation (no longer an upper bound):

$$f_i \sim \left( \frac{a_i}{1 - \alpha} \right)^{n - \lfloor \alpha n \rfloor} \left( \frac{1 - a_i}{\alpha} \right)^{\lfloor \alpha n \rfloor + 1} \frac{1}{1 - a_i - \alpha} \sqrt{\frac{\alpha(1 - \alpha)}{2\pi n}} \qquad (n \to \infty). \tag{4.2.6}$$

The result is then that $f_i = O(c_i^n)$, where

$$c_i = \left( \frac{a_i}{1 - \alpha} \right)^{1 - \alpha} \left( \frac{1 - a_i}{\alpha} \right)^{\alpha} < (1 - \alpha) \frac{a_i}{1 - \alpha} + \alpha \frac{1 - a_i}{\alpha} = 1, \tag{4.2.7}$$

by the weighted arithmetic-geometric mean inequality, which is strict because

$$\frac{a_i}{1 - \alpha} < 1 < \frac{1 - a_i}{\alpha}. \tag{4.2.8}$$

Since $c_i < 1$, it follows that $f_i$ decreases at least exponentially as $n \to \infty$ (for $a_i < 1 - \alpha$).

Combining $f_i$ for all inputs gives an asymptotic approximation to the fault probability for the $k$-out-of-$n$ system averaged over both the distribution of inputs and the ensemble of implementations,

$$\begin{aligned} \langle F_{n,k} \rangle &= \sum_i p_i f_i \\ &\sim \sum_{a_i > 1 - \alpha} p_i + \sum_{a_i < 1 - \alpha} p_i \left( \frac{a_i}{1 - \alpha} \right)^{n - \lfloor \alpha n \rfloor} \left( \frac{1 - a_i}{\alpha} \right)^{\lfloor \alpha n \rfloor + 1} \frac{1}{1 - a_i - \alpha} \sqrt{\frac{\alpha(1 - \alpha)}{2\pi n}} \quad (n \to \infty). \end{aligned} \tag{4.2.9}$$

(We neglect the vanishingly unlikely case $a_i = 1-\alpha$.) This result implies that the fault probability is asymptotically dominated by those inputs for which a fraction greater than $1-\alpha$ of implementations drawn from the ensemble produce incorrect output. If there are no such inputs, then the fault probability decreases at least exponentially as $n \to \infty$.

As a trivial example, if the "ensemble" consists of a single implementation, then each $a_i$ is either 0 (correct output) or 1 (incorrect output). In accordance with common sense, there is no benefit from voting a collection of identical copies of this implementation, since the inputs with $a_i = 1$ will produce faults just as surely. If the ensemble is broadened using a technique that can generate slightly different implementations, then it is hoped that for such inputs *some* implementations will produce correct output, so that the $a_i$ values decrease from 1. (New incorrect outputs will generally also be introduced, causing other $a_i$ values to increase from 0.) The nontrivial question is whether implementations can be effectively generated from an ensemble with $a_i < 1-\alpha$ (e.g., $a_i < \frac{1}{2}$ for a majority vote) holding simultaneously for all or almost all inputs. In this case, the desired exponential decrease in fault probability (and corresponding exponential increase in the difficulty of triggering a fault) could be achieved, at least as an intermediate asymptote until any residual "hard" inputs put a stop to the scaling.

## 4.3 Plurality Voting

In the previous section, we made no distinction among different incorrect outputs for a given input. As a result, to avoid an overall fault, the correct output was required to win by either a majority or a supermajority. This was a pessimistic assumption because in a realistic $n$-version system, even if the correct output has less than a majority, the incorrect outputs are typically not all identical and it is possible that no single incorrect output has more votes than the correct output. This is analogous to the "vote splitting" effect seen in political elections using a plurality system.

As before, we assume a given ensemble of implementations and denote by $a_i$ the probability that an implementation drawn from the ensemble gives an incorrect output for input $x_i$. We now represent the various possible incorrect outputs by an index $r$, so that $a_{i,r}$ is the probability of incorrect output number $r$ for input $x_i$, and $a_i = \sum_r a_{i,r}$.

The simplest case in which there exists a plurality that is not a majority is a system with $n = 4$ implementations. Here two implementations with correct outputs can win if the other two have *different* incorrect outputs. Previously we considered a $k$-out-of-$n$ system requiring at least $k = 3$ implementations with correct outputs. The previously derived fault probability for input $x_i$ reduces to

$$f_i = \sum_{j=2}^{4} \binom{4}{j} a_i^j (1-a_i)^{4-j} = 6a_i^2(1-a_i)^2 + 4a_i^3(1-a_i) + a_i^4. \tag{4.3.1}$$

In the plurality voting case, three or four incorrect outputs still lead to a fault, but two incorrect outputs lead to a fault only if they are identical. Thus the fault probability becomes

$$f_i' = 6(1-a_i)^2 \sum_r a_{i,r}^2 + 4a_i^3(1-a_i) + a_i^4. \tag{4.3.2}$$

Because each $a_{i,r}$ is nonnegative, we have $\sum_r a_{i,r}^2 \leq a_i^2$, and so the new fault probability is less than or equal to the old one. If we suppose that there are $H$ equally probable incorrect outputs, then $a_{i,r} = a_i/H$ and

$$f_i' = \frac{6a_i^2(1-a_i)^2}{H} + 4a_i^3(1-a_i) + a_i^4. \tag{4.3.3}$$

The $a_i^2$ term, which is the dominant one for small fault probabilities, is reduced by a factor of $H$. More generally, this factor can be viewed as a measure of the entropy of the distribution of incorrect outputs.

# Chapter 5

# Software Diversity at Scale in Commodity Operating Systems

In an operating system and application configuration monoculture, as is typical of corporate desktop operations, all machines subjected to the same attack would be expected to have the same symptoms. An intentionally stealthy attacker could easily go unnoticed if successful everywhere. A wide variety of general purpose and special purpose GNU/Linux desktop software environments is available. We conducted a small experiment to assess the feasibility of detecting software flaws or malicious software by running one application across a spectrum of Linux desktop configurations and watching for differences among the behaviors of the systems.

## 5.1 Evaluating Diversity of Linux Distributions for Detecting Malicious or Buggy Code

There are many ways in which Linux and application distributions vary. Packager and distributor choices include, among others:

1. What gcc compiler and linker options affecting security to use
2. What kernel security features to enable
3. What default resource access policies and policy options to give the user
4. What components (programs or drivers) to include in the distribution
5. What form to distribute code in primarily (source or binary)
6. What hardware to support

Our selection of Linux variants is based primarily on convenience and on popularity of distributions. With the small sample of commodity Linux versions listed in Table 5.1, sufficient variation is obtained to conclude that the many more expensive ways of obtaining diversity present a diminishing return on investment within the scope of this effort.

**Table 5.1.** Commodity Linux versions tested.

| Vendor | Release | Build |
|---|---|---|
| Redhat | 5.4 | 64 bit desktop |
| Redhat | 5.3 | 64 bit desktop |
| Fedora | 12 | 64 bit desktop |
| Fedora | 12 | 32 bit desktop |
| Ubuntu | 9.10 | 32 bit desktop |
| Ubuntu | 9.10 | 64 bit desktop |
| Ubuntu | 9.10 | 64 bit server |
| OpenSUSE | 10.2 | 64 bit desktop |
| OpenSUSE | 11.2 | 64 bit desktop |

**Table 5.2.** Security features of Linux versions.

| Distribution | RELRO | Stack canary | NX | PIE | ASLR | Selinux | map0 |
|---|---|---|---|---|---|---|---|
| Redhat 5.x (2) | None | Some | Y | Little | Some | Mostly | 4k |
| Fedora 12 (2) | None | Most | Y | Little | All | Weak | 4k |
| Suse 10.2 | Some | Little | Y | Some | Some | Weak | No |
| Suse 11.2 | Some | Some | Y | Little | Some | Some | 4k |
| Ubuntu 9.10 (3) | Some | All | Y | Little | All | None | 64k |

Firefox, Java Runtime, and other user-level tools are expected to be functionally identical (and be identically vulnerable) across OS variants for a given release, but vendor-specific builds of them skip many and various security features. Table 5.2 summarizes a survey of the prepackaged security features with each distribution. Distributors and packagers vary surprisingly in defense feature use, dependent in part on the distributor target market and in part on distribution history. Each column is explained in the following.

Several of these features are applicable on a per-file basis. In these cases, we looked at a sampling of widely used applications or libraries and rated the consistency of application as one of *None, Little, Some, Most, All.*

- *RELRO* Relocatable table (ELF sections) Read-Only: At linking time, some ELF sections can be marked as writable pages only during application loading and read-only to the application thereafter. Control of this property is by ld option ”-z relro” to gnu ld. Partial RELRO means BIND_NOW is not used, which keeps open the window of vulnerability during the lazy symbol resolution process. Controlled by gcc/ld options on a per-executable basis.

- *Stack canary* Gcc stack smashing protection. Buggy in gcc3 c++, but adds a random canary for each process. Reimplemented more reliably in gcc4.

- *NX/PaX* Non-executable page bit for data on 64-bit capable architectures with hardware support. Slows down many buffer overflow attacks by keeping data pages separate from code pages and code pages marked unwritable. Defeatable for bugs which simply inject bad data into code which interprets the data and does bad things through normal mechanisms. Controlled (normally) in hardware, though possibly shut off in kernel.

- *PIE* Position independent executables loaded at random addresses. In-process attacker no longer knows ahead of time where function entry points will be. Other sources of address info (e.g., /proc) must also be hidden for this to be effective. Controlled by gcc/ld options.
- *ASLR* Address space layout randomization changes address and order in which libraries get loaded so function entries are not as easily known to attackers. A weak version is still used in older Linux kernels; some tuning of this feature is possible from the Linux sysctl facility. Address randomization is done in more recent Linux versions with PaX, a software-based implementation of NX and ASLR ideas.
- *Selinux* A broken (complex and easily misconfigured) policy-based attempt at managing process privileges on a per-resource basis. Most desktop users (if given a choice) simply turn off or put selinux in advisory-only mode because many application writers are not adequately selinux aware and many vendor-supplied policy files are inadequately aware of application requirements. More recent distributions typically have more aggressive enforcement defaults while older ones have weak defaults.
- *Map0 Protection* prevents attacks based on kernel bugs that incorrectly pass control to invalid pointers at low addresses. The attacker typically mmaps their payload to a low address and provokes the kernel bug to get the attack code executed. Recent kernels have modified mmap to disallow mmap below a minimum threshold (defined by parameter mmap_min_addr). The map0 table entries are the default value of this parameter. This parameter is controlled by kernel configuration and adjustable by the root user.

### 5.1.1 Linux Variants Not Considered

Briefly we can characterize all the generally available omitted Linux variants as follows:

- Older and newer versions of the distributions tested
- Other mass-market oriented binary-packaged distributions
- Niche market distributions such as for scientific or embedded computing
- Source-based distributions such as Gentoo Linux which may be targeted to mass or niche markets

Of these, the source-based distributions could be most easily modified by some automated system-wide refactoring of data structures or call sequences to defeat or make more easily visible attacks based on injection of precompiled code. Furthermore, all these distributions, in their source code form, could be altered by changing the security related options passed to compilers and linkers in building the binaries. Both approaches to obtaining diversity are much too expensive to apply to a typical desktop environment in this project: The former approach requires nonexistent automated software refactoring tools that scale to tens and hundreds of millions of line of code in five or more implementation languages, and the latter approach requires automated refactoring to modify build options in hundreds or thousands of packages. Today's desktops are massive, entangled software collections where most automatically generated build variations either (a) fail to build or (b) expose code bugs because most of the build scripts are very brittle (very likely to fail after any unanticipated change at all).

### 5.1.2 Automated Intrusion Detection in a Diverse Platform Environment

In this experiment we set aside the construction of an advanced intrusion detection system capable of running a single interactive desktop job on multiple diverse systems (where each system ostensibly supports the same functionality) and watching for variance in program behaviors. We instead apply a manual approach to job launch and data collection to first determine if sufficient diversity exists to justify a greater data collection effort.

To implement diversity-based intrusion detection, a substantial effort is needed to develop and deploy customized device drivers, event loggers, and web proxies across a suite of disparate Linux platforms. Loggers are needed that, without interfering substantially with application performance, monitor and detect significant differences in access patterns to all system resources (files, network, memory). Due to the functionality of desktop and web-based applications, "significant differences" may be difficult to define algorithmically. Device drivers are needed that allow a single set of I/O (mouse, keyboard, video) to be multiplexed to many operating systems. Applications interacting with web services, particularly secured services, require a proxy server capable of intercepting and scheduling once the many identical queries that will come from each of the monitored systems.

## 5.2 Experimental Application and Results

We aim to demonstrate that a threatening or subtly defective program is detectable using a diversity approach. Noticeable differences in response to the same program and same input should indicate the program is worthy of deeper inspection. Our chosen test program is a publicly available application encapsulating a small suite of attacks aimed at obtaining local elevation of privilege from user level to root level or at extracting normally private information from a running kernel useful in subsequent attacks [1]. We reviewed the entire application code to verify the absence of undocumented side effects or persistent side effects before running all the tests in a private network. The documented intent of the application is to shame Linux developers and Linux vendors into becoming more secure; thus, the variance in its output is a printed message rather than execution of a malicious payload.

The results are shown in Table 5.3. A note of caution: Do not conclude that newer is always

**Table 5.3.** Results of vulnerability testing on Linux versions.

| Distribution | Application Output |
|---|---|
| RH 5.4 | Got root! |
| RH 5.3 | Got root! |
| Suse 10.2 | Got root! |
| Suse 11.2 | No suid root. |
| Fedora 12 (all) | No suid root. |
| Ubuntu 9.10 (all) | No suid root. |

better. For example, the Ubuntu distributions happened to be invulnerable to this particular application, but all shipped with another extremely easy-to-exploit vulnerability [9].

## 5.3 Conceptual Soundness

From this demonstration we can conclude that desktop malware detection based on subsequent behavior differences (file, network, or other device accesses) is possible. A great deal more work is necessary than was possible in this limited LDRD effort to realize such detection. In particular, future work is needed to address:

1. Application behavior capture mechanisms that go beyond conventional intrusion detection
2. Timing-aware and process-boundary-aware difference detection algorithms
3. Reducing the scope from an entire desktop environment to perhaps a critical application
4. Developing benefit metrics beyond bugs detected and attacks discovered

# Chapter 6

# Conclusion

The notions of diversity and redundancy are found throughout the literature and in practice. They are known to provide benefits in real-world systems. Individually and to some extent in combination, these notions have been applied in the fault-tolerance community as a method providing for correct execution of certain functional implementations in either hardware or software.

This project has examined various aspects of redundant, diverse implementations from a security perspective. Our measures of the benefit of such a combination show that with reasonable assumptions on how vulnerabilities overlap, there is an exponential benefit in the application of these ideas at a linear cost to implement. Depending on how the diversity is achieved, there is potential to provide a method of using the basic undecidability constraints to the advantage of the defender over the attacker. In all other known paradigms, undecidability works for the attacker against the defender.

Our simple experiments with the diversity found in commodity implementations of a particular feature set (the Linux operating system) show that even basic levels of diversity are enough to detect some adversarial manipulations. On the other hand, this research has experienced firsthand the difficulty of implementing these ideas. Obtaining real diversity in an actual implementation of specific functionality is hard to do. Automated tools to generate diversity are some time away.

We believe that, given a specific ensemble, it may not always be possible to measure the diversity found within the collection and thus it may not be possible to measure the actual benefit provided, even though the benefit may be substantial. There is also a great deal of work needed in the area of secure voting. With given bounds on vulnerability distribution, we show that it is possible to define a voting scheme and predict the probability that a faulty output may pass the vote. However, this is much different from producing an actual secure voter in an actual system.

Our results indicate that combining diversity and redundancy has great potential as a paradigm for future security efforts. There is much research needed to sort through basic details of metrics and development, and even more effort needed to bring theory and practice in line.

# References

[1] `http://grsecurity.org/~spender/enlightenment.tgz`.

[2] R. C. Armstrong and J. R. Mayo. Leveraging complexity in software for cybersecurity. In *Proc. 5th Cyber Security and Information Intelligence Research Workshop*, Oak Ridge, TN, April 2009.

[3] S. S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of faults in an $N$-version software experiment. *IEEE Transactions on Software Engineering*, 16:238–247, 1990.

[4] E. M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[5] Computer Emergency Readiness Team. CERT® advisory CA-2002-25: Integer overflow in XDR library. `http://www.cert.org/advisories/CA-2002-25.html`.

[6] Computer Emergency Readiness Team. Multiple DNS implementations vulnerable to cache poisoning. `http://www.kb.cert.org/vuls/id/800113`.

[7] J. Dutka. The incomplete beta function – a historical profile. *Archive for History of Exact Sciences*, 24:11–29, 1981.

[8] B. Littlewood, P. Popo, and L. Strigini. Modeling software design diversity. *ACM Computing Surveys*, 33:177–208, 2001.

[9] National Institute of Standards and Technology. Vulnerability summary for CVE-2009-4131. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4131`.

[10] J. Oberheide, E. Cooke, and F. Janhanian. CloudAV: $N$-version antivirus in the network cloud. In *Proc. 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[11] B. Salamat, T. Jackson, A. Gal, and M. Franz. Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. EuroSys'09*, Nürnberg, Germany, April 2009.

[12] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security*, Washington, DC, October 2004.

[13] Wikipedia. f00f. `http://en.wikipedia.org/wiki/F00f`.

[14] Wikipedia. Rice's theorem. `http://en.wikipedia.org/wiki/Rice's_theorem`.

## DISTRIBUTION:

| | | | |
|---|---|---|---|
| 1 | MS | 0899 | Technical Library, 9536 (electronic copy) |
| 1 | MS | 0359 | D. Chavez, LDRD Office, 1911 |

Sandia National Laboratories