Leveraging Complexity in Software for Cybersecurity

[Extended Abstract]

Robert C. Armstrong
Sandia National Laboratories
Livermore, California 94551
rob@sandia.gov

Jackson R. Mayo
Sandia National Laboratories
Livermore, California 94551
jmayo@sandia.gov

ABSTRACT

A method for assessing statistically quantifiable improvements in security for software vulnerabilities is presented. Drawing on concepts in complexity theory, undecidability, and previous work in high-reliability systems, we show that ensembles of similar implementations have statistical value even though each by itself is inscrutable. Research questions are identified that may allow practical application of these concepts.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*

General Terms

Security, Reliability, Design, Measurement

Keywords

complexity, cybersecurity

1. INTRODUCTION

Complexity of software is an artifact of the complex things we require computers to do [9]. Their capacity for computation is inextricably connected to the fact that they are also unpredictable, or rather capable of unforeseen emergent behavior [5]. Vulnerabilities are one of those behaviors.

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:

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 [1]). 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. Complex systems' emergent behavior cannot be predicted 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 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. 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.

2. THE BURDEN AND OPPORTUNITY OF COMPLEXITY IN CYBERSECURITY

Except for supply-chain exploits calculated to compromise downstream consumers, vulnerabilities are unintentional artifacts incidental to an implementation of an intentional design. Here we will focus on software systems, but most of the concepts transfer, in modified form, to hardware and networks. It is useful as an object of study to separate the

[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.

CSIIRW '09, April 13–15, Oak Ridge, Tennessee, USA

feature set (i.e., the designed inputs, outputs, and state that a program produces) from the complete set of things the program can be caused to do. This feature set can be defined concretely as a set of tests that verify a program's adherence to it—fitting neatly within the philosophy of Test Driven Development [2]. Under normal conditions these tests cannot be made exhaustive pragmatically. The test suite ensures that the program performs in the foreseen ways to the foreseen input history. But we know that it will almost certainly perform in other ways to *unforeseen* input histories; some of these will be benign, some faults, and others vulnerabilities.

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 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 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). 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 in the conclusion.

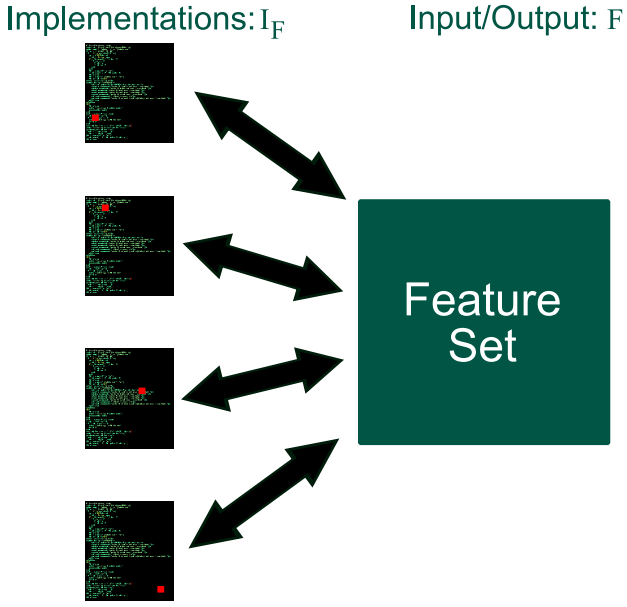

Figure 1: Here the "feature set" F is the collection of designed inputs and outputs that a program produces. There is a large set I_F of possible implementations of the given feature set, each different in their unrelated implementation particulars. Assuming that the design of the feature set is sound, security "holes" (red dots) in programs happen as a result of the implementation only, and not the feature set.

3. N-VERSION TECHNIQUE ADAPTED TO CYBERSECURITY

An obvious application of ensembles is closely related to N-version software techniques [6] for high-reliability, fault-tolerant systems used in aerospace and other time-critical systems and recently applied to cybersecurity [7, 8]. 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 implemen-

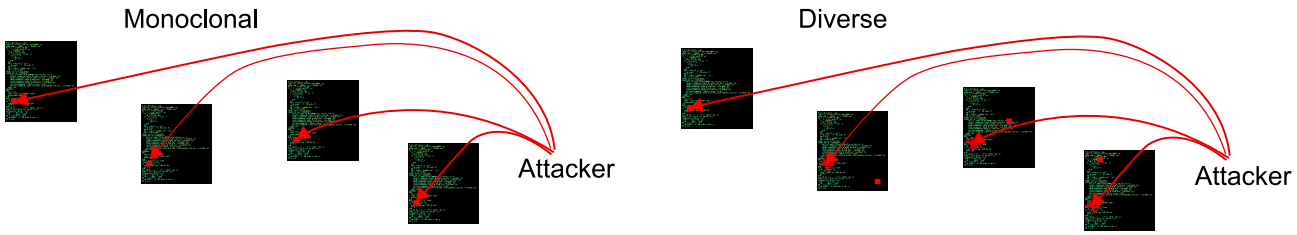

Figure 2: Turing completeness and undecidability cause security holes to be unknowable in the general case to programmers, users, and attackers. Implementations that are identical (monoclonal) are equally susceptible to the discovery of a security hole by an adversary. Diverse implementations are less so, depending on the number of variants.

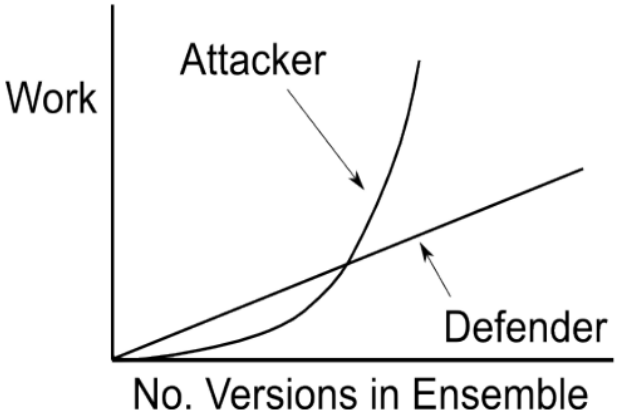

Figure 3: An idealized graph compares the linear scaling of defender effort with the exponential scaling of attacker effort for a system that votes diverse software implementations.

tations can be very substantial (Figure 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.

4. EXAMPLE SYSTEM FOR GENERATING DIVERSE IMPLEMENTATIONS

4.1 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.

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 behav-

ior (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.

4.2 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.

5. 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.

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.

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.

6. ACKNOWLEDGMENTS

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Department of Energy under contract DE-AC04-94AL85000.

7. REFERENCES

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

[2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[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] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

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

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

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

[9] S. Wolfram. *A New Kind of Science*. Wolfram Media, Champaign, IL, 2002.