

Defining an Adaptive Software Security Metric from a Dynamic Software Failure Tolerance Measure

J. Voas, A. Ghosh, G. McGraw, F. Charron
Reliable Software Technologies
21515 Ridgetop Circle
Sterling, VA 20166
{jmvoas,anup,gem,fhchar}@RSTcorp.com

K. Miller
Dept. of Computer Science
U. of Illinois at Springfield
Springfield, IL 62794
miller@uis.edu

Abstract

This paper describes a software assessment method that is being implemented to quantitatively assess information system security and survivability. Our approach — which we call Adaptive Vulnerability Analysis — exercises software (in source-code form) by simulating incoming malicious and non-malicious attacks that fall under various threat classes. A quantitative metric is computed by determining whether the simulated threats undermine the security of the system as defined by the user according to the application program. This approach stands in contrast to common security assurance methods that rely on black-box techniques for testing completely-installed systems. AVA does not provide an absolute metric, such as mean-time-to-failure, but instead provides a relative metric, allowing a user to compare the security of different versions of the same system, or to compare non-related systems with similar functionality.

1 Introduction

The original computer security defense strategy, circa 1970, was appropriately termed “penetrate and patch.” At that time, defense was entirely *reactive* — something that happened only after an attack was detected and some damage had already been inflicted. Penetrate and patch was followed by a series of more advanced defensive techniques (*e.g.*, real-time intrusion detection and auditing tools). Unfortunately, a recent proliferation of sophisticated threats has caused defensive security schemes to come full circle, back to where they began twenty-some years ago. Penetrate and patch has once again become the status quo.

This work applies technologies originally developed for software engineering (specifically, for software test-

ing and software assessment) to perform software security assessment. Related work includes [3, 24, 4], though the fault-injection-based approach taken here is novel. The premise underlying this work is that tools used in software certification/assurance disciplines are ripe for application to the computer security domain as well.

Problems in information security are more difficult to understand than those of other certification/assurance disciplines such as software safety or failure tolerance. The fact that a security threat is *malicious* adds subtleties and challenges that are different from those usually encountered in software quality assurance. For example, it is obvious that an “unsafe” event has occurred after an aviation disaster; but security intrusions are far less observable, and are often virtually undetectable. In a recent keynote address at the Eleventh Annual Computer Security Applications Conference (ACSAC’95) held in New Orleans, LA [27], Paul Strassman stated that only between 1 in 400 and 1 in 1,000 attacks are detected. Because security violations are so hard to detect, there is a shortage of good data about them.

The detection of malicious threats is one complication. The nature of those threats is another. Unlike real military intrusions, software intrusions are “virtual”. Counterintuitive though it may seem, an unsuccessful software offensive almost always strengthens the attacker (by way of gained knowledge), and does *not* strengthen the site attacked (by way of weakening the attacker). In traditional military intrusions, an unsuccessful offensive usually weakens the attacker at least as much as the site attacked. Furthermore, in traditional war strategies, the potential for retaliation provides an important deterrent to attack. On the information battlefield, however, the fear of retaliation is minimal at most, and does not affect the balance of power. Most information security techniques used to-

day are either based on the outdated tactics of twenty years ago, or are based on tactics that apply only to conventional warfare, not to information warfare. As a result of these shortcomings, the information infrastructure is left weakened and ill-prepared for defense.

This paper describes a new quantitative (though relative) measure for computer security. We seek to measure a program’s security weaknesses in terms of both known threats and unknown threats that may occur in the future. Security defenses can be more sharply honed and security strategies more objectively compared and contrasted if the vulnerabilities of a system can be scientifically measured. Our aim in this work is to develop such a measure.

Throughout this paper, we define a *security attack* as a dynamic event that occurs during the execution of a piece of software. An attack is made possible by a combination of two things. Firstly, weaknesses (or defects) must exist in the system under attack. Secondly, some particular sequence of weakness-exploiting input signals to the system is required. An attack is a particular event at a particular time. Like an attack, a *vulnerability* comprises two parts: a potential defect or weakness in an information system together with the knowledge required to exploit the defect. An attack is an active attempt to exploit a system vulnerability. A *threat* is an agent outside of a software system that works to exploit a vulnerability through one or more attacks [28].¹ For an attack to be successful, the sequence of input signals used to exploit a weakness often must be carefully timed. It is often the case that both a particular sequence of inputs as well as the timing of the inputs are critical features. To thwart an attack, a system can attempt any combination of three strategies: 1) compensate for its weakness with monitors or wrappers, 2) detect and deflect malicious input, and 3) disrupt the timing of an attack. When devising such strategies, it clearly pays to know as much about security vulnerabilities as possible.

This paper details our adaption of a software failure tolerance metric to security measurement. The aim of this paper is to provide the theory that underlies our new security assessment methodology which we shall call *Adaptive Vulnerability Analysis* (AVA) and to present a high-level prototype design for a vulnerability assessment prototype tool. AVA is intended to provide a relative measure of software security. Though AVA may fail to account for especially clever intruders who create *new* malicious threats from scratch, it will certainly be capable of simulating many important *pre-*

vious security threats and (with some luck) will also be able to detect some unknown vulnerabilities. Our approach will allow information system vendors to know *a priori* whether their systems are secure against a pre-defined set of threats, $T = \{t_1, t_2, \dots, t_n\}$, where T includes recurrent threats that are commonly encountered. T will be open-ended in the sense that when novel intrusion schemes do surface and are analyzed, T can be augmented so that they are included during security assessment. Since the evaluated metrics will vary with different sets T , we label the method “adaptive”. We attempt to simulate novel threats as well as known threats during the application of AVA. The success we will achieve through the injection of novel threats is less certain than the success we anticipate we will achieve in uncovering known vulnerabilities. However, the knowledge to be gained through the discovery of even one vulnerability using fault injection techniques is potentially significant.

2 The state of security assessment

As we stated in the Introduction, the current state-of-the-art in security assessment amounts to a “penetrate and patch” approach. Usual security-defense methods involve testing a site by applying common security attacks against it and determining if such attacks are successful. (This has come to be known as a *tiger-team penetration* approach to security.) If the result of any attack is an intrusion, then an appropriate patch for the software is devised and applied to the vulnerable site. Such patches may prevent further exploitation of a particular known vulnerability, but they provide no assurance that other vulnerabilities in software will not be exploited in the future. Furthermore, the patches are usually applied only to one site, leaving other sites open to the same attack until they, too, are patched.

Computer Emergency Response Teams (or CERTs) have been set up to try to coordinate the patching of multiple sites throughout the Internet. Each time a vulnerability is discovered and a patch is devised, CERT broadcasts this information publicly throughout the Net. The assumption is that patches will be applied large-scale throughout the Net. Unfortunately, many sites choose to ignore CERT Alerts until they have been successfully attacked. (This is a result of the common “don’t fix it if it ain’t broke” phenomenon that is pervasive in Computer Science disciplines.) Often, sites that have been invaded patch only the security hole that allowed the attack to succeed, leaving themselves open to other attacks.

Empirical vulnerability methods generally center

¹Note that “outside the software system” does not imply that the agent is necessarily outside of the organization being attacked.

around dynamic software testing that takes place *after* a system has been installed. One such method (briefly alluded to above) is *tiger team penetration testing*. In tiger-team testing, a group of experts in the design and weaknesses of operating systems attempts to break into an installed system. Tiger team penetration testing is aimed more at uncovering the presence of anomalies than at showing the absence of anomalies. Thus it could be argued that this form of testing is actually a specialized form of *stress testing*. One major disadvantage of tiger team penetration testing is that deciding exactly what the results of this testing mean is a highly subjective undertaking. Tiger teams all have different strengths and weaknesses, and often use dissimilar approaches. In other words, there is no standard in place for tiger teams.² Furthermore, vulnerabilities that actually exist in a system may not be discovered since they are so novel that they are nearly impossible to detect through standard penetration techniques. Tiger team testing is the traditional procedure that is used to demonstrate resistance to intrusion. However *quantifying* security via tiger team penetration testing is far from statistically re-demonstratable, since there are many factors in a manual process that may not be reproducible at a later time. In addition, as we mentioned above, tiger teams vary in their levels of expertise and their approaches. Another disadvantage of tiger team testing is that it is often difficult to put together a tiger team.

The Orange Book and similar evaluation schemes provide a rough measure of *trusted computing base* (TCB) security. Unfortunately, this criteria is an evaluation of the *process* that some piece of software undergoes during development, and does *not* involve any evaluation of the software itself. Therefore it is possible for a software-developing organization to “jump through all of the right hoops” but still end-up with a system having inadequate security. Models such as the TCSEC and the Systems Security Engineering Capability Maturity Model (SSECMM) [1] all suffer from this weakness — a weakness that comes as a result of *process versus product* verification. AVA is not geared toward any of TCSEC’s certification levels in particular; however, our metric could aid this process by empirically substantiating that an implementation is as sound as its design and specification.

A number of security tools that have automated the process for tiger-team testing are currently available in the public domain. Among these tools are the TAMU

²It is worth mentioning that some programs like `crackmapexec` could be construed as tiger-team approaches as well. Such programs obviously present some sort of “standard” approach that can be applied across systems, though our other complaints regarding tiger team testing still apply to these programs.

system [21], COPS [8], Tripwire [12], ISS [13], SATAN [9], and Crack [20]. The defensive security tools listed above (as well as related tools that are on the market or in the public domain) attempt to assess vulnerability using the tiger team approach. These tools allow users to attack a site through well-documented and common security holes. One major disadvantage of using these tools is that deciding exactly what the results of this testing mean is a highly subjective undertaking.

In addition to the penetrate and patch method of improving system security *after* systems have been installed, there are two other security-defense measures that are widely implemented: firewalls and data-communication encryption. Firewalls are gateway machines interposed between a local site (often a LAN) and the larger network outside the site (*e.g.*, the Internet). Often, a user must first log into the firewall itself in order to access a machine on the other side of the firewall. Firewalls serve to filter data packets based on their source and destination addresses. While firewalls are effective at thwarting some security attacks, they provide almost no protection against problems in higher-level protocol applications [5]. Consider the following trivial, but illustrative, example: if the software that a firewall uses is buggy, then vulnerabilities in the firewall software can be exploited by potential intruders. Firewalls can also be powerless in preventing attacks that aim to exploit vulnerabilities in applications executing at a *higher level* than that of the network protocol stack. For example, bugs in `sendmail` once permitted unauthorized remote users to execute programs on local machines protected by firewalls by placing actual commands in mail headers [2]. At the time, most firewalls did not pay attention to mail headers since this flaw was not anticipated (mostly due to the fact that the `sendmail` application executed at a higher layer of protocol than most firewalls). Even patching this flaw in `sendmail` does not solve the problem of intruders using other non-standard mail headers or exploiting other application-layer program vulnerabilities.

Another example of firewall security vulnerability was recently introduced along with the advent of Java and its hooks in the Netscape web browser. Over the past several months, many vulnerabilities in Sun Microsystems’s Java and Netscape’s JavaScript have been reported [10]. Although both Java and JavaScript were developed with security concerns in mind, unanticipated security vulnerabilities have come to light. In February of this year, three researchers from Princeton discovered a way to exploit a Domain Name Server (DNS) vulnerability through Java applets. More recently, the same researchers discovered (but have not released details about) a method for exploiting a vul-

nerability in Java that allows a malicious Java applet from a remote site to generate and execute raw machine code on a victim's local machine [7]. JavaScript security vulnerabilities have been employed to read a user's URL history list and transmit it to a remote site, to read a user's disk cache and send the information to a Web server, to steal the e-mail address of a Web user and forge an e-mail message with it, and to obtain a recursive listing of local disk directories of a website. We have little doubt that the importance of security assessment in web-based software development will continue to surge with the growth of the World Wide Web [16].

Encryption is generally used to ensure confidentiality in data communications and to authenticate a message's author. The primary drawback with relying heavily on encryption algorithms is (as in the firewall case) the strong possibility that the implementation of the encryption algorithm will be flawed. In a panel session at the 1995 National Information Systems Security Conference (NISSC'95) entitled, "Will Encryption Keep Hackers Out?", the panelists agreed that encryption does nothing to solve the problem of buggy software (among other problems) [25].

The on-going work summarized in this paper tackles the problem of assessing security from a completely different angle. Instead of trying to "break" existing software that has already been installed and is currently in use, our method is applied during the software development phase *before* a program is released in order to predict how strong a system would be if an attack were to occur. Our approach rests on the (very reasonable) premise that all software has errors. In the case of security-related software, any such errors may engender significant security problems.

As Cheswick and Bellovin state [5] (page 7):

... any program, no matter how innocuous it seems, can harbor security holes. (Who would have guessed that on some machines integer divide exceptions could lead to system penetrations?) We thus have a firm belief that everything is guilty until proven innocent.

The realization that bugs in software can lead to security intrusions is a prime motivation of our approach. In this sense, our approach is designed to find security-related bugs in software *before* the software is put into use. The method by which security vulnerabilities are found is based on fault injection techniques previously devised for use in analyzing software failure tolerance [11, 19]. Historically speaking, buggy software is responsible for most security vulnerabilities. As an example, consider that the Internet worm spread by sending unexpected input to a buggy *finger* daemon

[23]. The worm was programmed to send some program code to the daemon. The daemon program then executed a `gets` call that did not specify the maximum buffer length. To exploit this weakness, the worm filled the read buffer until it overwrote the return address in `gets`'s stack frame. As a result, when the call returned, it branched to the address in the overwritten buffer and executed the code sent by the worm. We concur with Cheswick and Bellovin who point out that the most effective safeguards in the Orange Book would have done no good at all against this kind of attack [5]. Unfortunately, all of the good software-engineering processes mandated by the Orange Book (and similar process-oriented guidelines) are ineffective against these sorts of attacks. This is because even the most assiduously developed code will still have bugs that can be exploited by malicious intruders. The Internet worm illustrates a case where a single component failure can lead to system-wide shutdowns in service. It also clearly illustrates the critical need for using unexpected or malicious inputs to programs during program testing. Not too surprisingly, many vulnerabilities are a direct result of what happens when a program processes "garbage" input that the program's author never anticipated [5].

As we have shown, the state-of-the-art in software security assessment is lacking in three fundamental ways. Firstly, there seem to be few (or possibly no) metrics. Secondly, most methods in common use take the "process versus product" approach. Thirdly, vulnerability assessment for the most part relies on post-installation tiger-team testing of known vulnerabilities. As we shall see, AVA addresses each of these three concerns.

3 AVA theoretical model

The software vulnerability metric that we propose is based on observing the impact of incoming *simulated* threats on an executing system. *Adaptive Vulnerability Analysis* (AVA) is a dynamic software analysis algorithm adapted from the extended propagation analysis technique (EPA) used in assessing safety-critical software [11, 19, 17]. Threat simulation in AVA will be accomplished through a combination of fault injection and test case generation. Intrusion detection will be accomplished using a predicate-based intrusion specification language.

3.1 To measure or not to measure

Some researchers in the software security assurance community may oppose the introduction of a metric to

quantify software security. We believe, however, that some way of measuring software security is required before programs with demonstrably better security can be developed. Just as some in the security field warn against using metrics, there is a vocal group of researchers in software engineering who crusade against placing numbers on software quality. These researchers hold that any such numbers either cannot be believed or are inaccurate. It is our contention that the only thing intrinsically wrong with numbers is that their meaning is often over-sold. If we remain realistic about the true worth of metrics, they provide an important scientific tool. We contend that numbers are the only way to assess anything meaningful about software quality, provided that any such numbers are used as *relative measures* of quality and not as absolute measures of quality.

Many software professionals rely on the theory that if the right processes are followed during development, then a secure product will result. We have termed this the “process versus product” problem. As we spelled out above, we find this exclusive emphasis on process misleading and even dangerous. Nowhere is the problem more evident than in the SSECMM [1]. The SSECMM comprises a set of development processes that, in theory, should improve security. While good software engineering processes may result in higher quality software, it is our opinion that good processes *alone* are not sufficient. Knight [14] warns of the complacency brought on by an overestimation of the impact of employing standard software processes in developing safety-critical software. Our suggestion is to enhance the “process” approach with a good “product” approach. This is entirely feasible since quantitative product-based assurance and process-based assurance can play complementary roles. Consider the fact that the metric we are proposing could be used as a tool to demonstrate whether a system developed according to the SSECMM *actually does* decrease vulnerability when compared with a system developed that does not adhere to SSECMM. With metrics come the possibility of objective comparisons of approaches.

In this paper we define a *vulnerability* as a weakness in a system that together with some exploitation knowledge allows an *intrusion* to occur. Weaknesses include logic flaws, coding bugs, particular input events, and combinations of these events. We define “level of vulnerability” as the degree to which unauthorized access is allowed in a system. For example, if access to inner layers of an operating system is supposed to be protected from users without system authorization, and if such users can frequently gain access, then the operating system will be classified as having a high level of

vulnerability.

We propose AVA as a concrete way to rate the vulnerability of a software system. Because it is an adaptive measure, the AVA vulnerability measuring technique can be specialized to assess different kinds of threats. The AVA environment can be tuned to better assess a particular piece of software based on vulnerabilities that similar pieces of software revealed in the past. For example, if `httpd` programs have proven to be particularly open to specific attacks, new versions of `httpd` should be tested using such attacks. AVA does not implement a traditional source code-based *static* measure (such as cyclomatic complexity, Halstead’s program volume, SLOC, etc. [29]). Instead, AVA measures how software behaves when it is forced into anomalous situations. The method by which we analyze how software behaves under malicious threats is closely based on the extended propagation analysis (EPA) algorithm described below. Our main objective is to determine whether a piece of software has weaknesses that can be leveraged into security vulnerabilities. Like its EPA ancestor, AVA measures a dynamic characteristic of software.

3.2 Extended propagation analysis

In order to set the stage for a discussion of AVA, we will first describe the EPA algorithm. EPA was designed and commercialized as a result of the urgent needs of a customer base concerned with safety-critical systems. The original EPA algorithm [17] has proven successful in assessing software safety. EPA employs fault injection and source-code instrumentation to evaluate a dynamic failure tolerance metric for software. Software is dynamically analyzed over multiple runs to assess its safety.

Under EPA, hazardous output conditions are specified through assertions of program variable states (*e.g.*, an assertion might state that if $\{a > 5 \text{ and } b > 100\}$ then a safety violation is considered to have taken place). Security vulnerabilities are usually not quite so simple. Hence, the assessment of software security requires modification of the original EPA algorithms. In addition, the original EPA methodology generally considers a fixed set of simulated infections (*i.e.*, fault injections) regardless of the kind of application under analysis. In terms of security assessment, this makes no sense. Security assessment requires the ability to adapt an analysis technique to a specific application. These fundamental differences have forced us to rethink some of the underlying assumptions behind the EPA algorithm, resulting in a series of major modifications and additions being implemented in the security prototype

tool. We call the resulting new technology AVA to distinguish it from the original EPA.

Our efforts to assess basic software failure tolerance began in 1995 with the EPA assessment paradigm. EPA provides a prediction of the “level of failure tolerance” for anomalies such as incoming hardware failures and programmer faults. Our metric reports a level of failure tolerance as a numerical estimate between 0.0 and 1.0, where 1.0 represents complete failure tolerance and 0.0 indicates no failure tolerance. The EPA metric is calculated for a program executing under a specific input profile. As one step in the process, the program under analysis is instrumented for fault injection and has hazardous condition assertions associated with it. The instrumented code is then run multiple times using an input profile. As a result of the metric’s dependence on the input profile, when the input profile changes, the measured failure tolerance will most likely change as well.

In order to improve software’s failure tolerance, there are two classes of “problems” that should be protected against: *software faults* and *hardware failures*. In the past, the EPA failure tolerance assessment method has been used to simulate both of these fault classes — providing observations of the impact to computations when a hardware sensor sends corrupt data or when the software itself computes corrupted intermediate data (with respect to its specification). The EPA technique simulates fault classes by inserting instrumentation into the code that forces changes (termed “simulated infections”) into the state of the program as it executes. A *simulated infection* is simply a modified data value assigned to some variable or set of variables during execution.³ We have successfully applied the EPA technique to many safety-critical applications. One such application involved measuring the failure tolerance of a safety-critical medical application, called the Magneto Stereotaxis System (MSS). We will now summarize the results of applying EPA to MSS in order to demonstrate EPA’s effectiveness.

3.2.1 Applying EPA to the MSS

MSS is an experimental device for performing human neurosurgery that was being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa, but to our knowledge no ongoing development is occurring. The system was designed to manipulate a small permanent metal seed that is

³Since true infections are incorrect program state values, our injected infections are termed “simulated” since we cannot know if our modifications in the state are forcing the program into an incorrect or possibly correct state.

moved throughout the brain using an externally applied magnetic field. By varying the magnitude and gradient of the magnetic field, the seed can be moved, positioned at a site requiring therapy (*e.g.*, a tumor location), and used for ablation and/or cauterization of the diseased tissue. The magnetic field required to move the seed is extraordinarily powerful. The field is produced by six superconducting magnets that are located in a helmet that is placed on the patient’s head. Dr. John Knight of UVA provided us with access to the source code for a version of the control software for the MSS. (Further information concerning the operation of the MSS is provided in a paper by Elder and Knight [15].)

The EPA failure tolerance assessment technique was applied to a portion of the MSS responsible for charging the coils in the helmet. Seed movement requests from the surgeon are converted into coil commands that charge or discharge the coils in order to create a varying magnetic field that will move the seed to a specified location in the brain. The coil controller code, which is written in C++, contains functions that interface with the hardware by reading and setting coil parameters. Several potential hazards were found to exist if errors in the software routines are executed or if hardware failures occur. For each of the hazards identified before EPA was applied, source code was instrumented to check for those hazardous conditions during execution. As a part of the analysis, the source code was instrumented with perturbation functions to corrupt program variable states randomly during execution. The three potential hazards that were checked for are summarized below:

Coil status fault: The status variable is used to determine if the coil is operational or in a fault state. If the coil subsystem detects an error during execution, (*e.g.*, an out-of-bounds parameter is set), the status variable is set to fault. A hazard results if the coil starts charging when the status variable is in the fault state.

Current update limit: The coil is charged according to a mathematical ramp model. When current requests are made by the control program, the current is increased or decreased to the target current according to the ramp function. The goal of the ramp function is to prevent large steps in current. A hazard results if a step in current exceeds 15.0 amps. This hazard is checked during EPA analysis.

Current Bounds: The current flowing through the coils at any point in time must always be in the

range of -100.0 to 100.0 amps. Current exceeding these bounds can be hazardous for the patient.

The hazardous conditions described above were instrumented in the coil subsystem code as hazardous assertions to be checked during EPA’s execution. Test cases were generated and then supplied to a coil control driver program. The program was executed over all test cases and analyzed using the EPA algorithm. The failure tolerance results from this analysis are shown in Table 1.

Hazard	Failure tolerance score
Coil Status	0.497
Current Update	0.245
Current Bounds	0.643

Table 1: MSS Simulation Results

The results from these simulations show the percentage of simulations that did not violate the particular hazard condition asserted during execution. This metric is the *software’s failure tolerance*. For safety-critical applications, the magnitude of the results are not as important as the existence of violations of the hazardous conditions. Thorough examination of the simulation results and causes can improve the failure tolerance of the code. For example, the maximum current step is checked each time the current is updated according to the ramp model. However, this function calls another function that sends the target current to a hardware device. Perturbations in the called function resulted in the maximum current bound being exceeded. The lesson learned is that the maximum current bound should be checked in the called function just prior to sending the target current to the hardware device. Furthermore, the actual current on the coil should be read after it is sent to the hardware device and compared against the sent target current to confirm correct operation.

Simulated infections of the type we used in MSS have provided a plausible and relatively inexpensive means for measuring software failure tolerance, software safety, and software testability (all of which are desirable attributes for software). In order to successfully employ simulated infections to the task of analyzing software vulnerability, we will require a way to simulate combinations of both code defects and specifically-targeted input sequences. In addition, the notion of a vulnerability must be made concrete enough so that intrusion detection is possible.

3.3 Moving from EPA to AVA

EPA’s success as a safety-assurance technique suggests its application to other closely-related fields. One such field where dynamic assurance methods and metrics are sorely lacking is software security. A natural question that follows is: *how might EPA be adapted for use in security assurance?* This question motivated our original research proposal. Basically, adapting the standard EPA methodology into a security assurance tool requires three major activities: 1) strengthening the ability to simulate code weaknesses through fault injection, 2) implementing automatic test case generation, and 3) creating a language for specifying intrusion predicates. We call the resulting methodology Adaptive Vulnerability Analysis. Next we describe each component of AVA in turn.

3.3.1 Simulating code weaknesses

Step one in adapting EPA to AVA is enhancing EPA’s primitive means for simulating code weaknesses (flaws) via fault-injection mechanisms. EPA employs “perturbation functions” that force simulated infections into program states. AVA will likewise use these constructs. *Perturbation functions* are the source-instrumentation mechanisms used to inject simulated anomalies such as logic faults, hardware failures, or threats. Such infections are equivalent to program states of the sort caused by syntactic code mutations. Besides making use of perturbation functions, we will also occasionally employ syntactic strong mutation [22] to simulate classes of code weaknesses for which it is especially difficult to derive a perturbation function. Such weaknesses include:

- out-of-sequence events,
- event failures,
- deadlocking, and
- incorrect events.

To ensure that our technique will have the capacity to simulate defects that might have security implications, we have analyzed those vulnerabilities archived at CERT (as well as those discussed in [6] and [3]). A concrete example of the theory and the implementation of fault-injection-based simulation techniques is provided by the perturbation function, `flipBit`. `flipBit` allows a user to flip any memory bit that the program has access to, from 0 to 1, or vice-versa.

Fault injection with flipBit

The first argument to function `flipBit` is the original variable or program state that we wish to corrupt. The second argument is the bit to be flipped (we assume little-endian notation). The function `flipBit` can be written in C as shown below and then linked with the executable. NOTE: The `^` represents the XOR operation in C and the `<<` operator represents a SHIFT-LEFT of `y` positions.

```
void flipBit(int *var, int y)
{
    *var = *var ^ (1 << y);
}
```

`flipBit` can be used to model various kinds of program state corruptions, including:

- all bits high,
- all bits low,
- random bit patterns,
- off-by-one, and
- n random bits flipped, ($n \geq 1$).

In summary, the corruptions generated by `flipBit` simulate programmer flaws, human factor errors, design flaws, and corruptions coming in to a component from external sources. `flipBit` provides the lowest-level atomic operator possible to corrupt digital information.

3.3.2 Test case generation

In order to exploit code weaknesses, a means for defining expected input profiles as well as unusual input sequences is necessary. Input sequences come in many variations. We denote the set of all inputs to a system by Δ . In the vulnerability analysis algorithm to follow, the user will define either a file of input sequences or a distribution of input values. We define Q to represent the normal operational profile of inputs that some system is expected to receive and \bar{Q} to represent the inverse operational profile.

Realize that during testing according to some distribution Q , the majority of the test cases that are chosen at random from this subset of Δ is made up of cases that are the most likely to be selected. This is true even if testing an enormous number of times. As more and more samples are picked according to the assumed operational profile and run, the potential for repeating particular test cases (or at least running test cases that do not exercise any novel functionality or forge new

paths) becomes greater. Running more and more test cases according to the same distribution does improve the chances of selecting an “infrequent test case” (from an area of the input space which we term the “ultra-rare” region), but only slightly. In some cases (*e.g.*, if the distribution has a small variance), selecting members of the ultra-rare region, \bar{Q} , may require sampling an intractable number of test cases. When an operational profile is inverted in order to sample “rare” test cases, the “rareness” of these cases stems solely from the infrequency with which they are sampled during operational use. This does not mean that such cases are necessarily a small subset of Δ . In fact, it is possible that the “rare” test cases represent a substantial portion of Δ .

Sampling from the input space rarely used in practical operation may prove to be useful for detecting security vulnerabilities. That is, processing unexpected, rarely used but legal inputs may result in a program state that compromises the security of the system. These ideas suggest that it will be meaningful in this effort to combine the “modified-for-security”EPA algorithm with test cases that represent the ultra-rare region of an input space. Such inputs, when employed, will help predict how secure the software will be during unusual operational events. When the software does not perform securely during experiments with unusual inputs, chances are that it will not perform securely if unusual events occur *after* deployment. Likewise, if the software does behave securely, this suggests that the software may be sufficiently hardened against anomalous events and threats. Such indicators are not guarantees of future secure behavior of the code, but they clearly suggest that such behavior will occur.

Note that there are certain operational distributions that when inverted produce the original distribution. Hence this idea of inverting a distribution will only be applicable to test distributions with spikes (*i.e.*, large variance in the probabilities). For brevity, we will not describe the formulae and algorithms for actually doing the inversion (see [18, 19]).

3.3.3 Specifying intrusions using predicates

The third major requirement for our security assurance tool is a clear definition of what kinds of events represent security breaches. Security intrusions are the events that our tool must detect. A security intrusion can include: unauthorized read access, unauthorized write access, denial of service to the system under attack, and so on. The vulnerability assessment tool will simulate threats by injecting faults into an executing program and then asking the question: *has a particu-*

lar type of intrusion occurred? Intrusions will be defined via logical predicates using a predicate specification language that is powerful enough to specify a range of different intrusions.

We envision the development of a predicate specification language having C-like syntax. We plan to incorporate the ability to call `perl` scripts using the predicate language in order to write sophisticated detection filters. We intend to provide some “default” sets of predicates for use in common situations. As pointed out above, intrusion detection predicates are context sensitive and will need to be specialized for use with certain kinds of programs.

We call the predicate expression that will permit specification of a security intrusion for a particular application, *PRED*. In analyzing an `ftp` daemon program, *PRED* may specify, for example, that user *anonymous* is not allowed to own any files or directories within the `ftp` file partition. Depending on the layer of the protocol stack at which the program under attack operates, the intrusion classes will vary. At the application layer, having a program create a shell with *SUID* may constitute an intrusion, while at the operating system kernel layer, creating root shells will most likely be acceptable.

Care is required in the definition of intrusions. The more ambiguous the description, the more potentially insecure events will be marked. The more specific the description, the fewer potentially vulnerable events will be generated and the more precise our ability to localize regions of vulnerability will be. These two characteristics of intrusion descriptions are not necessarily at odds, but both should be considered in the development of *PRED*.

3.4 The AVA algorithm

Based on the modeling of weaknesses, inputs, and intrusions, the following algorithm determines the proportion of intrusions that occur as a result of injecting a program with simulated threats. The number resulting from this algorithm can be used to calculate the AVA security metrics discussed below.

Let *P* denote our program, *x* denote a program input value, *Q* denote the normal usage probability distribution of Δ , \bar{Q} denote the inverse usage probability distribution, and *l* denote a program location in *P*.

Algorithm 1:

1. For each location *l* in *P* that is appropriate, perform Steps 2–7.
2. Set **count** to 0.

3. Randomly select an input *x* or input sequence from *Q* or \bar{Q} , and if *P* halts on *x* in a fixed period of time, find the corresponding data state created by *x* immediately after the execution of *l*. Call this data state *Z*.
4. Alter the sampled value of variable *a* found in *Z* creating \check{Z} , and execute the succeeding code on \check{Z} . The manner by which *a* is altered will be representative of the threat class from *T* that is desired.
5. If the output from *P* satisfies *PRED*, increment **count**.
6. Repeat steps 3–5 *n* times, where *n* is the number of input test cases.
7. Divide **count** by *n* yielding $\hat{\psi}_{alPQ}$, a vulnerability assessment, for each line *l*. This means that $1 - \hat{\psi}_{alPQ}$ is the *security assessment* that was observed, given *P*, *Q*, and *T*.

We will now dissect this algorithm and explain what it is doing and why.

The first two steps of the algorithm are basic. AVA is a source-code based methodology, meaning that instrumentation is placed between particular statements (which we call “locations” in the code). Either an automated system that implements the algorithm (if it is intelligent enough) or the user must tell the system which locations are relevant for fault-injection. Thus, the first step is to localize where injection is to occur. Next, a counter is initialized to zero, since we wish to observe how many security intrusions occur due to the simulated threats that the prototype attempted for a particular location *l*.

Unlike most software metrics currently in use, our AVA software assessment measure is not looking at software structure, but rather at software *behavior*. Hence the algorithm selects test cases upon which the program will run, since we will need to run the software in order to statistically quantify its behavior. This sampling of inputs is performed in Step 3. The inputs can come from different testing schemes that are more likely to trigger a successful intrusion: rare events (with respect to the operational profile), known input sequences that are unusual or likely to be threatening, totally random inputs, or even the operational profile of the system. The fourth step performs the actual program state corruption or syntactic mutation of the code (*i.e.*, it is the fault injection step). Once the fault that is injected by Step 4 is executed during the analysis phase, the program has been “tripped-up” in some

way. Step 5 then determines if the problem forced during Step 4 causes the program to produce an output event that satisfies our definition of what constitutes a security violation. If so, the counter is bumped by one. In order to provide a statistical estimate of vulnerability, Steps 3, 4, and 5 are repeated (Step 6). This estimate is calculated in Step 7.

For the metric that we are building, the class of all potential threats is infinite. Clearly, Algorithm 1 cannot simulate all members of the set. Furthermore, for any particular P , it is likely that most members of T are irrelevant. For these reasons, our prototype tool has two different means for defining the members of T that are relevant:

1. default perturbation functions, and
2. a perturbation function template that will allow the user to define the idiosyncrasies of specialized threats that are only relevant to P . This template will be tuned to specific input signals, source-code-based defect classes, and timing.

As shown above, our approach requires that the implementation has certain default threat classes built into it, as well as a capability for the user to define application-specific threat classes. Without this capability, commercial prospects for the tool would be poor.

Because the class of potential, future threats is unknown, any set of default threat classes may not adequately reflect how future threats will affect internal program states. In addition, the user of the perturbation function template may not have the foresight to envision certain classes of threats. To address this weakness in the technique, our prototype will have a set of default perturbation functions that do not necessarily simulate threats, but simulate random corruptions in the state of the executing program. These random corruptions will be forced into the software after which program behavior will be analyzed to see whether $PRED$ is ever satisfied.

Since future threats may be so novel that they are unknown, we will simulate as many anomalous events as possible. It is plausible that some of the future threats will have signatures on the states that are at least similar to the random corruptions that we employ. We recognize the theoretical weakness of this argument, but we know of at least one anecdotal example of where EPA was employed without any application-specific perturbation functions, and was still able to find many weaknesses in a safety-critical system even though *only* random state corruptions were employed [11]. That is, the perturbation function template was not employed (to our knowledge), yet the EPA algorithm, when only random corruptions were used, found

dozens of locations in the software where potential hazards could have originated. This information allowed the developers to add appropriate cleansing assertions to the code in order to ensure that certain classes of corruption were never allowed in the program states. Although these classes of corruptions were random, when the programmers sat down and analyzed the results, they realized that these were events that were so undesirable that even though they did not know the chances of such state corruptions actually occurring in the future, they decided to take the corrective actions to avert potential future disasters anyway.

A recent, significant research finding in support of software fault-injection showed that the behavior of simple, “simulated” faults nicely mirrors the behavior of actual, complex fault classes [26]. Although this study employed student programs, it was a highly controlled experiment that was in no way biased in favor of the findings that it revealed. For years, many researchers in the formal methods community have expressed skepticism regarding the utility of fault-injection methods. Their arguments cited a lack of real evidence that simulated faults behaved in a similar manner to real faults. It is our contention that software fault-injection is one of the most under-utilized technologies in software assessment today. We expect that once AVA is implemented and has been shown to work on systems such as the `httpd` daemon and similar utilities, we will have similar empirical results to report. Results from using EPA support this claim [11].

3.5 Global security metrics from AVA

Our measure of information system security is not an *absolute* metric, such as mean-time-to-failure. Instead, it is a *relative* metric that allows a user to compare different versions of the same system, or to compare different (but similar) systems that have the same purpose. It is clear that particular classes of threats may be extremely powerful when applied to system A, yet impotent when applied to system B. Obviously if A fails to survive some given threat, yet B can survive the threat (only because the chosen class of threat is inherently impotent against B), then we cannot necessarily say that B is more secure than A. However, for two different systems that are susceptible to the exact same set of threats, it will clearly be possible to compare their relative strengths.

Our metric, *minimum-time-to-intrusion* (MinTTI), is based on the information produced by Algorithm 1. Two notes here: (1) the metric can be formalized for any unit of time that is desired, (2) the larger the assessed value, the more secure we predict the system

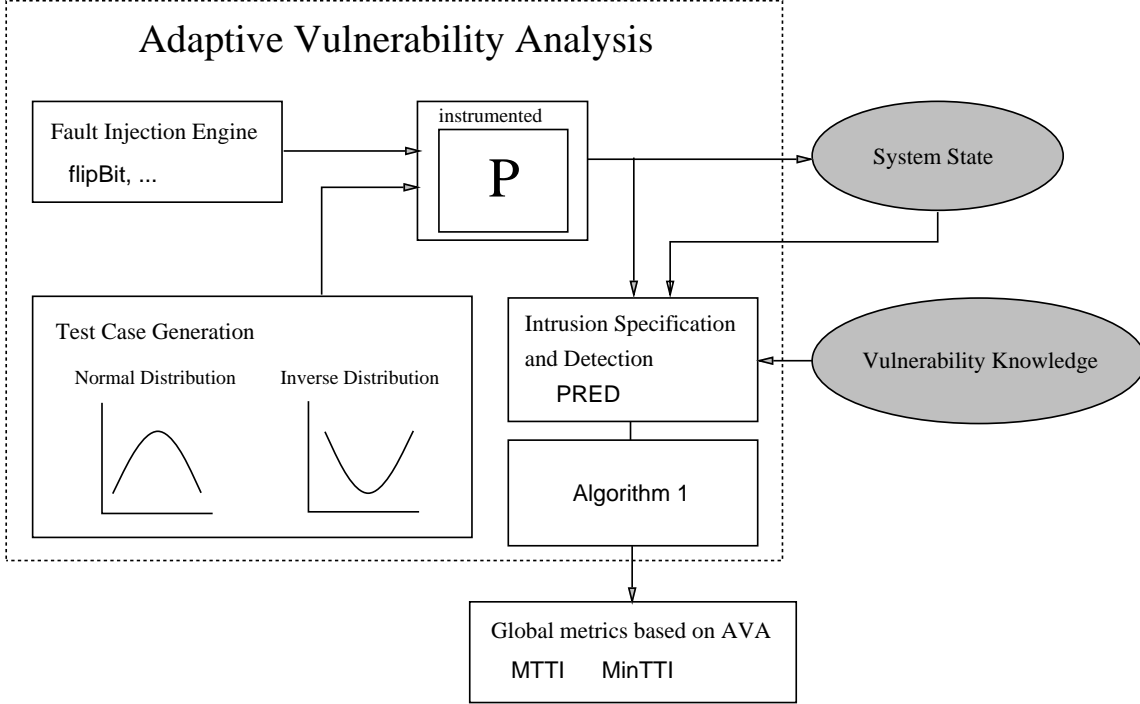


Figure 1: Overview of the AVA prototype. The three main components of the system — fault injection, test case generation, and intrusion specification — provide a system for determining global security assessment metrics.

will be.

Mean-time-to-intrusion (MTTI) is defined as the average time interval before an intrusion will occur based on three things: input cases in Q (and its inverse), the classes of fault injections that are used, and the classes of intrusions defined in $PRED$. The *minimum-time-to-intrusion* (MinTTI) is the shortest predicted period of time before any intrusion defined in $PRED$ will occur. Note that in the equations below the *execution probability* $\hat{\epsilon}_{lPQ}$ of location l of program P is simply the probability that a randomly selected input x selected according to Q will execute location l . We define MTTI next according to the vulnerability assessment metric calculated in Algorithm 1 and the execution probability $\hat{\epsilon}_{lPQ}$:

$$\text{MTTI} = \frac{\sum_{l=1}^M [\hat{\psi}_{aIPQ} \cdot \hat{\epsilon}_{lPQ}] \cdot \left(\frac{\text{program executions}}{\text{unit of time}} \right)}{M}]^{-1} \quad (1)$$

where M is the number of locations where AVA was applied. Our equation for MinTTI follows:

$$\text{MinTTI} = \left[\max_l [\hat{\psi}_{aIPQ} \cdot \hat{\epsilon}_{lPQ}] \cdot \left(\frac{\text{number of program executions}}{\text{unit of time}} \right) \right]^{-1} \quad (2)$$

The MinTTI measure is based on the location in the code that demonstrated the greatest weakness (*i.e.*, the location(s) that satisfied $PRED$ the most often). $\hat{\psi}_{aIPQ}$ does not account for the frequency with which any particular location is executed according to Q , and so we must account for the frequency with which l is reached in our calculation, which is given by $\hat{\epsilon}_{lPQ}$.

We have described the theoretical framework on which we are basing our innovation. This framework has included a discussion of AVA’s predecessor, EPA, the components of AVA, the AVA algorithm, and a relative metric for comparing relative security assessment measures. Such assessments can be used to compare the security of different systems, the security of different versions of the same system, or the security of the same system under different operational conditions.

4 The AVA prototype tool

Currently, we are building a prototype tool that implements the AVA methodology for assessing security of information systems. A design overview of the AVA prototype tool is shown in Figure 1. The AVA prototype tool executes an instrumented program P with inputs from the normal operating condition distribution,

or the inverse operating condition distribution. Faults (*i.e.*, simulated threats) are injected into the program source code during execution. Algorithm 1 is used in conjunction with the defined intrusions in order to determine where vulnerabilities are most likely to exist. Using the information recorded during runs with simulated threat injections together with a set of definitions for intrusions in the predicate language, the global security metrics for P are calculated and recorded.

The AVA prototype tool will be used to estimate the vulnerability of C/C++ programs with respect to simulated attacks that cause predicate-detectable intrusions in the output of the program under attack. The prototype tool will work by instrumenting the source code of the program under attack such that each time P executes, attacks are simulated using default and user-defined perturbation threats. As the program executes, Algorithm 1 is evaluated and the resulting $\hat{\psi}_{alPQ}$ data for instrumented source code lines are used to calculate the global security metrics.

5 Conclusions

Our security research project began as an attempt to modify a successful failure-tolerance assessment tool, EPA, to work in the security domain. RST's safety tool is currently one of the most sophisticated software fault-injection tools in the marketplace. However, it was designed to simulate code defects and hardware failures, not security threats. In order to create a tool that can simulate security threat classes we are forced to develop a more robust, adaptive fault-injection capability. As a result, we are developing a new tool, AVA, that is capable of simulating anomalous events representing combinations of input sequences and code weaknesses. An alpha-release AVA tool with these capabilities is scheduled for limited distribution in the Fall of 1996.

AVA is not a foolproof assurance metric. We do not believe that any security metric will ever be comprehensive. However, AVA, if it proves successful once implemented and benchmarked, will provide a paradigm shift in approaches to software security research. Previous security measures have focused more on manual effort and human judgment. By contrast, AVA concentrates software experimentation and computer power on the problem of assessing security.

AVA has several clear advantages over other approaches to software security:

1. It can be continually modified to simulate additional classes of threats. Once the algorithms are implemented as described above, new threat

classes can be quickly linked into the system without disturbing the processing of old threats.

2. It can be customized (by the user) for application-specific classes of intrusions without requiring any changes in changing the assessment measures.
3. It measures dynamic information (as opposed to a static information) in order to determine behavior.
4. It can provide information from which to develop improved design-for-security heuristics,
5. It yields reproducible results — meaning that the same analysis parameters will produce the same results at a later time. Tiger team testing is too subjective to be considered reproducible in this way.
6. It can be used *in conjunction with* tiger team penetration testing and other established techniques to better assess software security.

The disadvantages of AVA are:

1. It is based on fault injection models, and thus any results are given only with respect to those threat classes being simulated. A new threat class, unseen before, with a lesser MinTTI than any previous threat, will result in an over-estimated level of confidence in security provided by our metric.
2. It requires that what constitutes a reasonable or unreasonable output event be clearly defined. This can be difficult for complex systems such as an operating systems, thus requiring an expert user.
3. The technique is applied late in the development life-cycle.
4. The technique could be used *against* a system being analyzed if certain internal information that exists during the execution of Algorithm 1 is revealed to potential attackers.⁴

In summary, we have provided an overview of the key issues that we have faced during the transformation of a software failure-tolerance assessment technique into a security assurance technique. Our methodology can isolate regions within the source code where further scrutiny should be applied. We believe that

⁴Although AVA is designed to produce a numerical prediction of MinTTI, to do so requires that it evaluate the source code on a region-by-region basis to determine where the weak regions are. This information reveals where offensive threats are most likely to succeed. This "disadvantage" could be seen instead as an opportunity for developing offensive capabilities.

our novel approach to software security will shift attention towards the root causes of information system vulnerability (including programming errors), and thus stimulate a greater interest in secure system design and security assessment.

Acknowledgments

This work is sponsored under the Advanced Research Projects Agency (ARPA) Contract F30602-95-C-0282.

References

- [1] Systems Security Engineering Capability Maturity Model Report October 2, 1995.
- [2] CERT Advisory. Sendmail vulnerability, November 1993. CERT Advisory CA-93:16.
- [3] T. Aslam. *A Taxonomy of Security Faults in the Unix Operating System*. Master's thesis, Purdue University, West Lafayette, Indiana., 1995.
- [4] B. Beizer. *Software Testing Techniques*. Electrical Engineering/Computer Science and Engineering. Van Nostrand Reinhold, 1983.
- [5] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.
- [6] C. LANDWEHR, A. BULL, J. McDERMOTT AND W. CHOI. A Taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, September 1994.
- [7] D. Dean, E. Felton, and D. Wallach. Java security: From hotjava to netscape and beyond. Web documents at URL: <http://www.cs.princeton.edu/~dDean/java>. Also see the related CERT Alert document CA-96.05., March 1996.
- [8] D. Farmer and E.H. Spafford. The security checker system. In *USENIX Conference Proceedings*, pages 165–170, Anaheim, CA, Summer 1990.
- [9] D. Farmer and W. Venema. Improving the security of your site by breaking into it. Available by ftp from <ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking.101.Z>, 1993.
- [10] J. Fisher. Java and Javascript vulnerabilities, March 1996. CIAC Notes 96-01.
- [11] J. VOAS, F. CHARRON, G. MCGRAW, K. MILLER AND M. FRIEDMAN. Predicting How Badly 'Good' Software can Behave. *Submitted to IEEE Software*.
- [12] G. Kim and E.H. Spafford. The design and implementation of Tripwire: A file system integrity checker. Technical Report CSD-TR-93-071, Purdue University, 1993.
- [13] C. Klaus. Internet security scanner. Available by ftp from <ftp://ftp.iss.net/pub/iss>, 1995.
- [14] J.C. Knight and K.G. Wika. Software safety in a medical application. Technical report, University of Virginia, Department of Computer Science, April 1995.
- [15] M. ELDER AND J. KNIGHT. Specifying user interfaces for safety-critical medical systems. In *Proc. of the 2nd Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 148–155, November 1995.
- [16] G. McGraw and D. Hovemeyer. Untangling the woven web: Testing web-based software. In *Proceedings of the Thirteenth International Conference on Testing Computer Software*, Washington, D.C., June 1996.
- [17] J. VOAS AND K. MILLER. Dynamic testability analysis for assessing fault tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994.
- [18] J. VOAS AND K. MILLER. Examining software quality (fault-tolerance) using unlikely inputs: Turning the test distribution up-side down. In *Proc. of Eighth Annual Conference on Computer Assurance*, pages 3–11, National Institute of Standards and Technology, Gaithersburg, MD, June 1995.
- [19] J. VOAS AND K. MILLER. Predicting software's minimum-time-to-hazard and mean-time-to-hazard for rare input events. In *Proc. of the Int'l. Symp. on Software Reliability Eng.*, pages 229–238, Toulouse, France, October 1995.
- [20] A.D.E. Muffett. Crack version 4.1, a sensible password checker for Unix. Available by ftp from <ftp.cert.org/pub/tools/crack/readme.txt>., 1992.
- [21] D.R. Safford, D.L. Schales, and D.K. Hess. The TAMU security package: An ongoing response to Internet intruders in an academic environment. In

Proceedings of the Fourth Usenix UNIX Security Symposium, pages 91–118, Santa Clara, CA, October 1993.

- [22] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [23] E.H. Spafford. The Internet worm program: An analysis. *Computer Communications Review*, 19(1):17–57, January 1989.
- [24] E.H. Spafford. Extending mutation testing to find environmental bugs. *Software Practice and Principle*, 20(2):181–189, February 1990.
- [25] E.H. Spafford. Will encryption keep hackers out? In *COAST Watch Digest*, March 1996. Summary of Panel Discussion from NISSC '95.
- [26] M. DARAN AND P. THEVENOD-FOSSE. Software error analysis: A real case study involving real faults and mutations. *Proc. of the ACM SIGSOFT ISTA '95*, pages 158–171, 1995.
- [27] B. Werner, editor. *Eleventh Annual Computer Security Applications Conference (ACSAC'95)*, Los Alamitos, CA, December 1995. IEEE Computer Society Press.
- [28] G.B. White, E.A. Fisch, and U.W. Pooch. *Computer System and Network Security*. CRC Press, New York, 1996.
- [29] H. Zuse. *Software Complexity: Measures and Methods*. DeGruyter, Berlin, 1990.