Computer Vulnerability Analysis
Thesis Proposal

Ivan Krsul
The *COAST* Laboratory
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
krsul@cs.purdue.edu

Technical Report CSD-TR-97-026

April 15, 1997

Abstract

Computer security professionals and researchers do not have a history of sharing and analyzing computer vulnerability information. Scientists and engineers from older or more established fields have long understood that publicizing, analyzing, and learning from other people's mistakes is essential to the stepwise refinement of complex systems. Computer scientists, however, have not followed suit. Programmers reinvent classical programming mistakes, contributing to the reappearance of known vulnerabilities.

In the recent past, computer systems have come to be a part of critical systems that have a direct effect on the safety and well-being of human beings and hence we must have lower tolerance for software failures.

In the dissertation I will attempt to show that computer vulnerability information presents important regularities and these can be detected, and possibly visualized, providing important insight about the reason of their prevalence and existence. The information derived from these observations could be used to improve on all phases of the development of software systems, as could be in the design, development, debugging, testing and maintenance of complex computer systems that must implement a set of policies defined by security analysis.

A significant portion of the work that must be performed will concentrate on the development of classifications and taxonomies that will permit the visualization and analysis of computer vulnerability information. I hope that these classifications and taxonomies applied to a collection of vulnerabilities will provide a set of features whose analysis will show that there are clear statistical clusterings and patterns caused because developers and programmers are not learning from each others mistakes. This analysis may be performed by applying statistical analysis and knowledge discovery tools.

1 Introduction

Computer vulnerability analysis, or the process of collecting vulnerability information, classifying this information, storing it in some consistent and logical format, and processing it using analysis tools, is not a well understood process. In spite of the fact that software developers have been repairing vulnerabilities in their products for many years, only a few researchers are beginning to understand that vulnerability analysis is an essential process in the step-wise refinement of software products.

Traditional sciences and engineering have a long history in the analysis of vulnerabilities of complex systems [Per84, Sch94, And94, Pet85, LS92]. For example, the aerospace industry has a long record of recovering vulnerability information from incidents, storing this information and finding regularities or patterns and incorporating their findings into production systems, making each subsequent product safer than the preceding one. Making sense of apparent chaos by finding regularities is an essential characteristic of human beings. The Austrian-born British philosopher of natural and social science Karl Popper even argues that "we are born with expectations... one of the most important of these expectations is the expectation of finding a regularity. It is connected with an inborn propensity to lookout for regularities, or with a *need* to *find* regularities..." [Pop69]

There are a few instances of researchers who have indeed attempted to find such regularities in computer vulnerabilities, including the vulnerability analysis project at the University of California at Davis [BD96] and the vulnerability classification efforts by Matt Bishop at the University of California at Davis [Bis95], both of which are used to develop generic vulnerability detection mechanisms; the intrusion detection work done by Kumar et al. at the COAST laboratory, that concludes that vulnerabilities can be grouped together in generic classes described by patterns [KS94, KS95b, Kum95, KS95a, $\mathrm{CDE^{+}96}$]; and the vulnerability classification work done by Aslam and Krsul that provides a taxonomy of security faults [Asl95, AKS96a]; Ross Anderson collected and analyzed vulnerability information for automatic teller machine (ATM) cryptosystems [And94] and concluded that designers of automatic teller machines (ATM) cryptosystems failed to understand the real threat model because vulnerability information is not collected, published and analyzed.

We see the same vulnerabilities appear again and again (for example a sendmail bug, a finger bug, a syslog bug, and a Netscape bug all share the buffer overrun problem). This should be a powerful incentive for the development of a freely available vulnerability database that could be used to learn from other people's mistakes. Many articles in the literature make similar arguments for the need of publicly available data collections and analysis tools [And94, Asl95, AKS96a, Pol]: computer programmers are not learning from their mistakes and we do not have tools that allow us to understand why this is the case.

The last few years have seen a surge in interest for the design and maintenance of vulnerability databases. These are closely guarded commodities that are typically oriented towards the storage of vulnerability related information, with system administrators that are capable of interpreting them in mind. These databases are not being widely published and analyzed.

Groups in industry that market intrusion detection systems require vulnerability information for the generation of their misuse signatures or intrusion patterns and benefit from proprietary databases because it gives them a competitive advantage. In many cases, industry, government, and academia resist the publication of these databases for fear that it would make them liable if hacker groups use this information to break into other systems. Intelligence agencies may not reveal vulnerability information because they may be using these against systems as part of their information warfare efforts. Finally, some systems administrators in academia, government, and industry fear that public availability of vulnerability information may trigger a greater number of intrusions or intrusion attempts by students or employees that otherwise would not be able to obtain vulnerability information.

2 Definition of Vulnerability

In the development of my dissertation, and all throught this document I will write about the development of vulnerability classifications and the analysis of computer vulnerabilities. Unfortunately, there is widespread disagreement on the precise definition of the term "computer vulnerabilitiy." Consider, for example, the following three widely accepted definitions:

1. In [BB96] Matt Bishop and Dave Bailey give the following definition of computer vulnerability:

> “A computer system is composed of *states* describing the current configuration of the entities that make up the computer system. The system computes through the application of *state transitions* that change the state of the system. All states reachable from a given initial state using a set of state transitions fall into the class of *authorized* or *unauthorized*, as defined by a security policy. In this paper, the definitions of these classes and transitions is considered axiomatic.”
>
> “A *vulnerable* state is an authorized state from which an unauthorized state can be reached using authorized state transitions. A *compromised state* is the state so reached. An *attack* is a sequence of authorized state transitions which end in a compromised state. By definition, an attack begins in a vulnerable state.”
>
> “A *vulnerability* is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states. If generic, the vulnerability may characterize many vulnerable states; if specific, it may characterize only one...”

2. The Data & Computer Security Dictionary of Standards, Concepts, and Terms [LS90] provides the following definition of computer vulnerability:

   > “1) In computer security, a weakness in automated systems security procedures, administrative controls, Internet controls, etc., that could be exploited by a threat to gain unauthorized access to information of to disrupt critical processing. 2) In computer security, a weakness in the physical layout, organization, procedures, personnel, management, administration, hardware or software that may be exploited to cause harm to the ADP system or activity. The presence of a vulnerability does not itself cause harm. A vulnerability is merely a condition or set of conditions that may allow the ADP system or activity to be harmed by an attack. 3) In computer security, any weakness or flaw existing in a system. The attack or harmful event, or the opportunity available to a threat agent to mount that attack.”

3. The Handbook of INFOSEC Terms [han96] provides the following definiton of computer vulnerabilty:

   > “1) A weakness in automated system security procedures, administrative controls, internal controls, and so forth, that could be exploited by a threat to gain unauthorized access to information or disrupt critical processing. 2) A weakness in system security procedures, hardware design, internal controls, etc., which could be exploited to gain unauthorized access to classified or sensitive information. 3) A weakness in the physical layout, organization, procedures, personnel, management, administration, hardware, or software that may be exploited to cause harm to the ADP system or activity. The presence of a vulnerability does not in itself cause harm; a vulnerability is merely a condition or set of conditions that may allow the ADP system or activity to be harmed by an attack. 4) An assertion primarily concerning entities of the internal environment (assets); we say that an asset (or class of assets) is vulnerable (in some way, possibly involving an agent or collection of agents); we write: V(i,e) where: e may be an empty set. 5) Susceptibility to various threats. 6) A set of properties of a specific internal entity that, in union with a set of properties of a specific external entity, implies a risk. 7) The characteristics of a system which cause it to suffer a definite degradation (incapability to perform the designated mission) as a result of having been subjected to a certain level of effects in an unnatural (manmade) hostile environment. 8) Weakness in an information system, or cryptographic system, or components (e.g., system security procedures, hardware design, internal controls) that could be exploited.”

From all these definitions we can at least infer that a computer vulnerability is the characterization of actions taken by a software system in violation to policy. It is unfortunate that policies are often not stated explicitly during the development of software systems and when they are specified they are often ambiguous.

Consider the first definition given above and it's application to a well known and documented vulnerability– the **xterm** log vulnerability in UNIX systems [KS95a, KS95b, CDE+96, Bis95]. The **xterm** program runs as root and is tricked into changing the ownership of an arbitrary file to that of the current user. We can model the program execution as a three step process where the program starts execution, asks for a file name, and changes the ownership of the file so as to give control of file to the user.

As Figure 1 shows, every filename entered by the user will cause the system to take a different path in the state machine describing the execution of the system. The number of states in the machine are finite because in all actual computing systems there is a limited amount of memory. In practice, however, the number of such states will be extremely large. In this case we are worried about those states that result from the application of the **chown** system call (in the figure S'_3, S'_4, S'_q, S'_{q+1}, S'_k, S'_n, and S'_{n+1}). A security policy will need to identify those states that can be considered to be compromised and a characterization of these states will provide the vulnerability information desired.

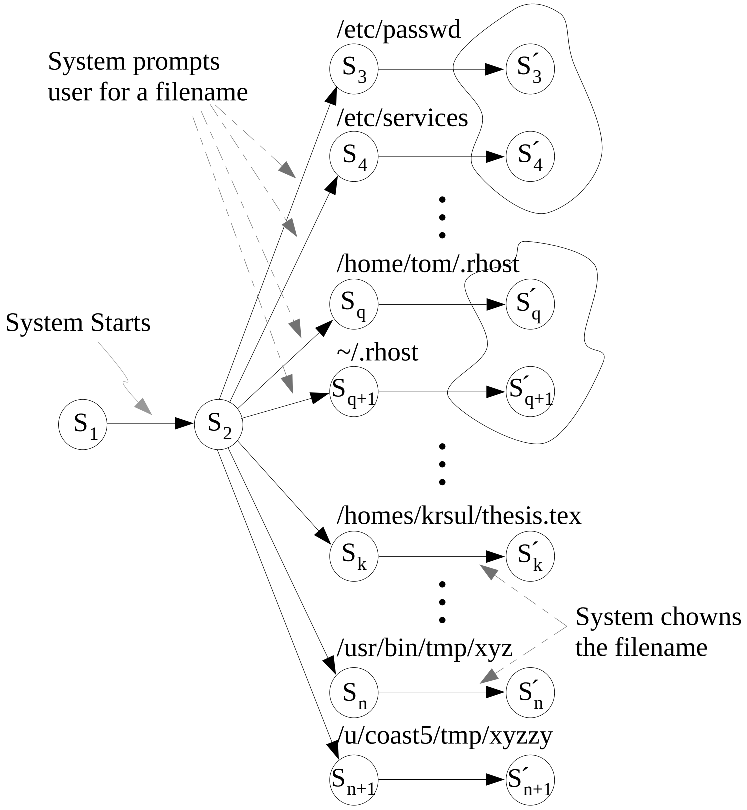

Figure 1: Sample characterization of vulnerable states for a simple version of the **xterm** log bug. It is not possible to determine all possible vulnerable states without a significant amount of semantic information.

Users familiar with the UNIX operating system would readily agree that states S'_3 and S'_4 are clear violations of the security policy explicitly stated by the UNIX access control mechanisms; it may be argued that states S'_q and S'_{q+1} are violations of policy because it is widely accepted that modifications to **.rhost** files should always be viewed with suspicion; whether states S'_k, S'_n, and S'_{n+1} are violations of policy is debatable because a fair amount of semantic information about the content and purpose of these files must be known a-priory.

And indeed, there are many cases where the explicit definition of policy depends heavily on perspective. Take, for example the actual case of Netscape running on a Windows 95 machine: How can a policy be specified for the access of files by Netscape? There is no access control mechanism provided by the operating system and hence Netscape itself must voluntarily enforce a security policy. Developers might, for example, specify that Netscape has complete control of all files in its temporary and cache directories. In at least one machine at the COAST Lab, however, the default for this temporary directory (as configured by the Netscape installer) is `C:\DOS`. From the perspective of the developer the policy might have been sufficient. From the perspective of the user this policy is certainly not specific enough and must be refined because this directory also contains the systems files and should not be tampered with.

This same problem exists in other operating systems with access control mechanisms like UNIX. In these operating systems users can tell Netscape what temporary directory to use, and the fact that a user might choose to have all his applications run with the same temporary directory does not mean that Netscape should modify files it has not created, regardless of what is explicitly stated by the UNIX access control mechanism.

Another example can be found with the Frame Maker application running under Solaris 2.5. The default behavior of this application, as installed in the Computer Science department at Purdue University, is to create a file called `fmdictionary` in the home directory of the person running the application whenever a word is added to the user dictionary. If there already exists such a file, Frame Maker will replace it with a new file containing the personal dictionary of the user.

The policy for what is allowed and not allowed with this software has not been explicitly stated. It's expected behavior, and hence the perceived policy, will be different from the point of view of the programmer (the policy is that the user may never have a file called `fmdictionary` in his home directory) and from the user (that may have a file or a symbolic link called `fmdictionary` that may have important information regarding, say, a dictionary of FM radio stations).

In her book "Cryptography and Data Security", Dorothy Denning states that an access control policy specifies the authorized accesses of a system and gives the following definitions of system states and policies [Den83]:

> The state of a system is defined by a triple (S, O, A), where:
>
> 1. S is a set of subjects, which are the active entities of the model. Subjects are also considered to be objects; thus $S \subseteq O$.
> 2. O is a set of objects, which are the protected entities of the system. Each object is uniquely identified with a name.
> 3. A is an access matrix, with rows corresponding to subjects and columns to objects. An entry $A[s, o]$ lists the access rights (or privileges) of subject s over object o.
>
> Changes to the state of a system are modeled by a set of commands, specified be a sequence of primitive operations that change the access matrix.
>
> A configuration of the access matrix describes what subjects can do – not necessarily what they are authorized to do. A protection policy (or security policy) partitions the set of all possible states into authorized versus unauthorized states.

As shown by the above examples, actual systems lack such detailed and explicit protection policies. Hence, it is difficult to provide a precise working definition of what a vulnerability is because this depends on a precise security policy for any system being analyzed. Also, a working definition of a vulnerability based on explicit policies is particularly difficult in technologies such as Microsoft's ActiveX [Gar96]. Systems running ActiveX, either as part of Microsoft's proposed Active Desktop [act96] or as part of the Microsoft Explorer WWW browser, dynamically download binary executables from the Internet without verifying that their execution satisfies the policies for the system in which it will run. Microsoft argues that it's Authenticode Technology guarantees the authenticity of software as [aut96] explains:

> "Today's web sites provide not only a rich experience for users but also the possibility of unwittingly downloading malicious code. With increasingly active content on the Internet, end users often must decide whether or not to download code over the Internet. However, end users cannot really tell what a piece of software will do, until they've downloaded to their PCs."
>
> "Unlike the retail environment, software on the Internet is not labeled or "shrink-wrapped." Therefore, end users don't know for sure who published a piece of software on the Internet. They also don't know if the code has been tampered with. As a result, end users take on a certain amount of risk when downloading, Java applets, plug-ins, ActiveX Controls, and other executables over the Internet."
>
> "Internet Explorer 3.0 uses Authenticode technology to help address this problem for end users. Authenticode identifies the publisher of signed software and verifies that it hasn't been tampered with, before users download software to their PCs. As a result, end users can make a more informed decision as to whether or not to download code."

The problem with this model is that developers, administrators, and users may not have the same view as to what policies should be in effect when the binary code has been deployed. An ActiveX control that scans a hard disk and reports its contents to a particular server, for example, may be appropriate for a user that is being aided by a consultant but may not be appropriate for users in the military.

At this point, the quest for finding a precise definition of the term "computer vulnerability" that everyone can agree on may seem hopless. There are, however, a subset of vulnerabilities that we may be able to clearly identify. The vulnerabilities that result from the violation of clear access control mechanisms (such as protection bits in UNIX) are included in this subset, and these will be used as a starting point for my dissertation.

At this point it is not clear that these are the only vulnerabilities we can identify unambiguously. Part of the work left during this dissertation is to develop a precise working definition of vulnerability that will encompass a large but manageable subset of the vulnerabilities encompassed by the definitions given in this section.

3 Related Work

Several groups have constructed vulnerability databases with various degrees of sophistication. Examples of such databases are[1]: the database at the COAST Laboratory; the CMET database at the AFIW (Air-force Information Warfare); the database maintained by Mike Neuman; the database at the University of California at Davis; the database at CERT; the internal databases at Netscape, SUN, and NCSA; etc.

The schemas for the databases I have seen, the CMET database, the AFIW database, the COAST database, the database at the University of California at Davis, the CERN database, and the database maintaned by Mike Neuman, share a common feature: they are designed for the storage of information, and not for its automatic manipulation. They are designed to store and classify the textual information that is normally available in the Internet, if at all. In any of these databases it is possible to search for vulnerabilities present in a particular version of an operating system, display the patches that have been released for a particular vulnerability, an display the scripts used to exploit a vulnerability or class of vulnerabilities.

However, in none of these databases can we perform even the most basic analysis on the information stored within. None of the databases mentioned above have enough information or an adequate representation to perform

[1] Information about these databases was provided through personal exchanges at workshops or using e-mail messages.

operations such as determining if a vulnerability is present in a particular system[2], whether finding a vulnerability in an operating system implies that the same vulnerability is present in another, or if the vulnerability information in the database seems to imply that there is a direct correlation between software metrics and the existence of vulnerabilities (correlation and cluster analysis). The ability to perform these operations is of importance in the prevention of the exploitation of unknown (but familiar) vulnerabilities.

In [Asl95, AKS96a] Aslam and Krsul explore the development of a classification scheme that can aid in the understanding of software faults that can subvert security mechanisms. Their classification scheme divides software faults into two broad categories: Coding Faults that result from errors in programming logic, missing requirements, or design errors; and Emergent Faults resulting from improper installation or administration of software so that faults are present even if the software is functioning according to specifications.

Several projects have dealt with the issue of identifying and classifying software faults, including the Protection Analysis (PA) Project which conducted research on protection errors in operating systems [CBP75, BPC75]; the RISOS project that was aimed at understanding security problems in existing operating systems and to suggest ways to enhance their security[A$^+$76]; the paper published by Carl Landwehr (et al.) in which he published a collection of security flaws in different operating systems and classified each flaw according to its genesis, or the time it was introduced into the system, or the section of code where each flaw was introduced [L$^+$93]; and the survey published by Brian Marick of software fault studies from the software engineering literature (the studies reported faults that were discovered in production quality software) [Mar90].

Finally, there are several products that incorporate vulnerability information for detection of their presence in computer systems. The more widely known are the COPS security checker [FS91], SATAN, Tiger, ISS, and the TAMU security checker.

4 Proposed Solution

In the ideal case it would be desirable to perform sophisticated computer analysis on existing vulnerability information including, but not limited to, the detection of hidden and significant, and possibly non-obvious, relationships in vulnerabilities; the application of correlation and clustering analysis to existing vulnerability information to aid in the understanding of the relationships between vulnerabilities and the systems they are found on; and the generation of software engineering tools that will find vulnerabilities on computer systems before these are deployed.

Data analysis tools work best when the data is well defined and when features are available. I propose to organize vulnerability information as points or vectors in n dimensional space. The term "vulnerability database" is so overloaded with meaning that we will speak of a *vulnerability state space* consisting of tuples of the form $<\alpha, \beta, \gamma, \ldots, \omega>$ where α, β, γ, and ω are enumerable characteristics of vulnerabilities such as cause, exploit, locale, and result. Each fully specified tuple represents a point in an n dimensional space.

This notation allows for the explicit manipulation of data in the vulnerability state space according to meaningful operators. One such operator is, for example, the projection operator. Consider, as illustrated by Figure 2, a three dimensional state space where there the only features analyzed are α, β, and γ. It should be possible to apply a projection operator to eliminate one of the dimensions and obtain from the state space a two dimensional bit pattern that can be used for cluster analysis or pattern recognition. The argument generalizes to n dimensions where projections can create hyper-planes where analysis can be performed.

There are serveral tools and mechanisms that we might use in the search for useful and insightful patterns in

[2] Determining if a vulnerability is present in a system is not the same as listing the operating systems that are known to have been affected by the vulnerability. Determining if a vulnerability is present involves examining the system of interest and determining if the conditions necessary for the system to be vulnerable are satisfied.

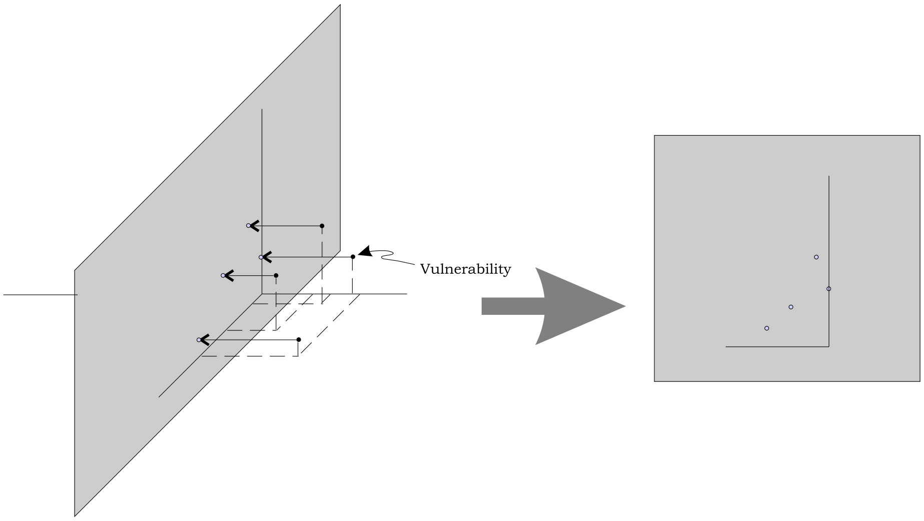

Figure 2: Detection of patterns on projections of the database. The figure illustrates how projections of a three dimensional state space can generate a two dimensional grid that could be used to detect patterns using cluster analysis.

the data of a good vulnerability state space including tools for discovery statistics, visualization, neural networks, applied perception, and machine learning [Wat95, Hed95, kdd, HS94].

The database model that this vulnerability state space is implemented on is not of critical importance. The COAST group, for example, has structured its database as a relational hierarchy implemented as flat text files. Other institutions have structured theirs as relational databases administered by commercial database engines. Others have structured their databases as simple flat files.

5 Contributions

Virtually every field where failure can be catastrophic has recognized that accumulation of information about failures is critical to the stepwise refinement of technology, particularly when the systems that fail are highly complex. In “The logic of failure,” Dietrich Dorner [Dor96] points out that complex systems that fail often have four characteristics that make them specially prone to failure: complexity, intransparence[3], internal dynamics, and incomplete or incorrect understanding of the system. Although this book analyzes highly dynamic systems such as nuclear power plants, management of entire cities or countries, etc., we find that many of these ideas can be applied to computer science in the search for explanations on why computer systems fail (or have vulnerabilities that can be exploited).

[3] Intransparence is defined as the fact that an observer of the operation of a system cannot see the inner workings of a program. It contributes to the development of faulty software because developers can not *see* the execution of the program without the help of sophisticated monitoring tools, and sometimes even these tools are not useful because their presence alters the behavior of the system.

Complexity is certainly a factor in the appearance of vulnerabilities as has been demonstrated by a particularly failure-prone system: sendmail. It has been argued that this system is so complex that the only truly intelligent action regarding this system would be to replace it with a simpler, more manageable one. That software development is a complex business is not a ground-breaking realization. Already in 1975 Frederick Brooks wrote in his book *The Mythical Man-Month* that "Since software construction is inherently a systems effort–an exercise in complex interrelationships–communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning." [Jr.95]

In "Why Cryptosystems Fail" [And94], Ross Anderson writes the following about this topic:

> *"When an aircraft crashes, it is front page news. Teams of investigators rush to the scene, and the subsequent enquiries are conducted by experts from organisations with a wide range of interests - the carrier, the insurer, the manufacturer, the airline pilots' union, and the local aviation authority. Their findings are examined by journalists and politicians, discussed in pilots' messes, and passed on by flying instructors. In short, the flying community has a very strong and institutionalised learning mechanism. This is the main reason why, despite the inherent hazards of flying in large aircraft, which are maintained and piloted by fallible human beings, at hundreds of miles an hour through congested airspace, in bad weather and at night, the risk of being killed on an air journey is only about one in a million."*

Several other sources including "When Technology Fails" [Sch94] and "Normal Accidents" [Per84] make it clear that prompt and complete information dissemination is critical if we want to learn from past mistakes. More often than not, it is not the designers of the systems that find and debug complex systems but observers who find patterns that lead to the cause of failures.

Scientists and engineers that are responsible for the development of critical systems are used to the idea of learning from past mistakes. In "Why Buildings Fall Down" [LS92], Levy and Salvadori describe in great detail some of the more spectacular structural failures in history and provide evidence that structural failures are likely to become less common because of the application of the knowledge gathered in the examination of past failures to modern designs. The same argument can be made in the design of airplanes, bridges, nuclear power plants, or any complex system where it is virtually impossible to design a correct system from scratch [Pet85, Jr.95, Sch94, Per84, Dor96, And94]. As Confucius said: "If you wish to control the future, study the past" [bri95].

Accordingly, a fundamental contribution from my work will be the organization of past vulnerability data into a state space where the classification of vulnerabilities (i.e. for each vulnerability assigning values to each of the dimensions of the space) is repeatable. Given the dimensions of the space and the vulnerability information, any person should be able to re-create the state space.

Some of the dimensions of this space will be classifications that will be self sustained, consistent, objective, and capable of distinguishing important features that can be used to find patterns of and dependencies that might help us better understand the nature of computer vulnerabilities.

The complexity, intransparence, and sheer volume of code in computer systems systems make it difficult to find such patterns and dependencies. Existing data mining techniques and other feature-extracting tools widely used in such areas as computer vision and pattern recognition could be used to extract information that would allow a much better understanding of why and how vulnerabilities get introduced in computer systems.

An example will illustrate the type of knowledge that can be obtained: Assume that the vulnerability database to be used would contain software metrics and system call counts for the systems that have known vulnerabilities. Assume further that the knowledge discovery tools found a high correlation between systems that have buffer overrun vulnerabilities, systems that have high decision counts, and systems that use the `gets` system call.

Software designers could then have information that could identify potential buffer overrun vulnerabilities of their systems automatically by simply computing the necessary metrics and comparing the results with those obtained in the example.

At this point, the example presented is complete speculation. However, those readers that have followed-up on the discovery of computer vulnerabilities in the past few years might agree that it is not entirely unreasonable. Even this result, however trivial it may seem at first, would be a considerable advancement in the field towards the development of tools that will help detect vulnerabilities during the development phase of a project.

For my dissertation I will attempt to find such relationships in vulnerability data and a prerequisite for this work is coming up with a list of features that will allow statistical analysis and data mining in the vulnerability state space from many sources that can be used as a starting point for finding these [SCS86, Tas78, KP78, RN93, KS94, KS96, OC91, Eva84, LR92, SSTG92, Tou94, OC90, Tas78]. The collection classification of vulnerabilities using these features should be an objective, reliable and repeatable process.

6 Completion Conditions

As with most scientific theories, confidence in our theory must be built by repeated observations that will validate our predictions. The philosopher of Science Karl Popper has emphasized that a good theory makes a series of predictions that can be proved false, or falsified, by observations. In his book *Conjectures and Refutations* he states that "...every genuine *test* of a theory is an attempt to falsify it, or to refute it. Testability is falsifiability." [Pop69]. As the physicist Stephen Hawking remarked: "Any physical theory is always provisional, in the sense that it is only a hypothesis: you can never prove it. No matter how many times the results of experiments agree with some theory, you can never be sure that the next time the result will not contradict the theory. On the other hand, you can disprove a theory by finding a single observation that disagrees with the predictions of the theory... Each time new experiments are observed to agree with the predictions the theory survives, and our confidence in it is increased; but if ever a new observation is found to disagree, we have to abandon or modify the theory." [Haw88]

And indeed I make a significant prediction: Computer vulnerability information presents important regularities and these can be detected, and possibly visualized, providing important insight about the reason of their prevalence and existence. I expect to provide enough supporting evidence that will show the usefulness of the knowledge acquired and techniques developed throughout the development of the dissertation.

It is conceivable, however, that during my research I may find that the result of experimentation will show that falsity of our theory, in which case a substantial contribution to the field would still be made because the prediction I have verbalized is implicitly assumed by a portion of the computer security community.

6.1 State Space Size

As stated in section 4, my work requires a collection of vulnerabilities for the state space, each with as many characteristics (also known as features or dimensions) filled in as possible from the possible set (i.e. the number of dimensions in the state space). Fully qualified features represent points in the multi-dimensional space and partially qualified vulnerabilities represent hyper-planes.

How many points or hyper-planes (i.e. samples) do we need to be able to apply the proposed data analysis tools? Without detailed knowledge of the distribution and characteristics of the data in question, we cannot know a-priori the number of samples required. The machine learning community, however, can provide us with some guidelines that will help us estimate the number of samples required for learning and classification algorithms in general. It is important to note that at this point in time, these guidelines set only very tentative limits to the initial size of the state space.

In [WK91], Weiss Kulikowsk claim that that for classifiers and learning systems, a surprisingly small number of test cases are needed for accurate error rate estimations: at 50 test cases there is a good chance that, even though the test sample error rate is 0%, the true error rate is as large as 10% (See Appendix A for a definition of these errors). At 1000 samples, the true error is almost certainly below 1%. Traditionally, a small statistical sample size is around 30 samples.

Many simplifying assumptions were made for this particular approximation, including some assumptions about the distribution of the test cases. These are assumed to be a good representation of the true population.

Another useful insight regarding the size of the state space is the rule that learning systems need samples that must exceed two to three times the number of features [DH73]. If we have 30 features in the state space, we must have at least 60-90 samples.

After reviewing some of the machine learning literature [WK91, DH73, Qui86, QCJ, Bre94, FS96] we can provide an initial estimation of the number of samples and number of features that will be needed: approximately 100-200 samples and 20-30 features.

6.2 Feature Indentification

A feature for a computer vulnerability is characteristic or metric of the vulnerability, it's exploitation, and the software it manifests itself on. The number of possible features for a vulnerability, according to the rather limited definition in section 2, is infinitely enumerable and one of the challenging aspects of this dissertation is the identification of a small set of useful features for the state space.

The following is an enumeration of some of the possible features we have identified to date. At this point there is no guarantee that any or all of these features will be useful for my analysis, or that we will be able to collect them automatically and objectively.

Impact. The impact and consequences of the exploitation of the vulnerability by a threat agent.

Immediate Impact. Rather than the ultimate or eventual impact of the vulnerability (which in Unix tends to be *root access*), the first or immediate impact of the vulnerability.

Threat. Threat based on Don Parker's threat classification scheme [Pow96].

System Vendor.

System Version.

Application.

Application Version.

Advisories. Institutions or groups that have issued an advisory about the vulnerability.

Analysis. Do we have a detailed analysis about the vulnerability?

Detection. Does a mechanism for detecting the exploitation of the vulnerability exist?

Test. Does a mechanism exist for detecting a vulnerability in a particular system?

Patch. Does a patch exist for the vulnerability?

Exploit. Does an exploit script exist for the vulnerability?

Aslam Classification. The Aslam et al. classification [AKS96b].

System Category. To what system component does the vulnerability belong to?

Software Metrics. A number of software metrics collected from the vulnerable systems might be useful including:

> **Program Complexity** .

System Call Usage. The use of several notorious system calls in programs that contain vulnerabilities.

Programming Language. Programming language used in the system that contains the vulnerability.

Access Required for Exploitation. What previous access is required for the exploitation of the vulnerability?

Ease of Exploit. How easy is it to exploit the vulnerability?

Origins and Causes. What are the origins and causes of the vulnerability?

7 Current State

In the past months I have been working on the development of a data set that will be used to test my theories. A byproduct of my research is a vulnerability database that is substantially better than the previous collection of vulnerabilities that resulted from the work done by Taimur Aslam for his masters thesis [Asl95]. As of this writing, the database consists of several fields that can be of types text, list, or hierarchical classifier. Text fields are free format and can include any information. Values for list fields are limited to codes defined in a field definition file. Similarly, values for hierarchical classifiers are limited to codes defined in hierarchical field definition file.

The vulnerability database can be found in `/homes/krsul/security/vulner`. This directory must be mounted as a Cryptographic File System[Bla93] before it can be used. Applications and tables related to the database can be found in `/homes/krsul/security/vdbase`. We will refer to this directory as the `$VAPP` directory. Library files including field definitions and lists and classifiers are in the `$VAPP/lib` directory. I will call this last directory the `$VLIB` directory.

The fields acceptable in the database as well as their types are defined in the `$VLIB/field_list` directory. As of this writing the list of acceptable fields allowed in the database are:

(Identification)			
Field ID	**Title**	**Description**	**Type**
vid	Identification	Vulnerability ID or identification string	Text
title	Title	Title of vulnerability	Text

(Description and impact)			
Field ID	**Title**	**Description**	**Type**
desc	Description	Vulnerability description	Text
u_impact	Ultimate Impact	Ultimate consequences of the exploitation of the vulnerability by a threat agent	H. classifier
i_impact	Immediate Impact	Immediate consequences of the exploitation of the vulnerability by a threat agent	H. classifier
impact_verbatim	Impact	Textual description of impact	Text
threat	Threat	Threat based in Don Parker's threat classification	H. classifier

(System Identification)			
Field ID	**Title**	**Description**	**Type**
system	System(s)	System(s) vulnerable	List
system_version	System version	System Version	Text
system_vendor	System vendor	System vendor	List
system_verbatim	Misc. system	Additional system information	

(Application)			
Field ID	**Title**	**Description**	**Type**
app	Application(s)	Application(s) vulnerable	List
app_version	Application version	Application Version	Text
app_verbatim	Misc. application	Additional application information	Text

(References)			
Field ID	**Title**	**Description**	**Type**
advisory	Advisory/ies	Advisory/ies that warn/describe about the vulnerability	Text
reference	References	References to the vulnerability in literature or on the net	Text

(Detailed analysis, detection techniques and fixes)			
Field ID	**Title**	**Description**	**Type**
analysis	Analysis	A detailed analysis of the vulnerability	Text
detection	Detection	Method of detecting that the vulnerability is being exploited	Text
fix	Fix	A fix that can be used to eliminate the vulnerability.	Text
test	Test	Method/s that can be used to detect whether the vulnerability is present in a system	Text
workaround	Workaround	A temporary workaround for the vulnerability. Used until a patch can be applied.	Text
patch	Patch(es)	A patch or a series of patches that can be used to eliminate the vulnerability.	Text

(Detailed information about exploitaition)			
Field ID	**Title**	**Description**	**Type**
exploit	Exploit Scripts	Reference to exploit scripts or programs	Text
idiot	IDIOT Pattern	IDIOT Pattern used to detect the exploitation of the vulnerability.	Text
access_required	Access Required	Access required for the exploitation of the vulnerability.	List
eo_exploit	Ease of Exploit	How easy is it to exploit the vulnerability?	List
comp_exploit	Complexity of Exploit	How complex is the exploitation of the vulnerability?	List

(Classifications and features)			
Field ID	**Title**	**Description**	**Type**
class	Classification	Classification	H. classifier
category	System/Category	To what system or component does the vulnerability belong to	List
origin_causes	Origin and Causes	Origin and causes of the vulnerability	List

(Verification of vulnerability)			
Field ID	**Title**	**Description**	**Type**
verif	Verified by	Person or entity that verified the vulnerability	Text

The following lists and hierarchical classifiers are currently defined in the vulnerability database:

Classifier: *Access required*

This classifier was originally defined from a talk given by Tom Longstaff [Lon97] and defines the kind of access that is required to exploit the vulnerability.

- Remote using a common service
- Trusted system
- User account
- Physical access
- Privileged access

Classifier: *Application*

This classifier defines the application that has the vulnerability. This classifier is relevant for those vulnerabilities that are present in user-level programs, daemons, servers, etc. that are not a part of the operating system istelf (kernel?)

- Netscape WWW Browser
- SUN's HotJava WWW Browser
- Java Developer Kit's appler viewer
- Oracle PowerBrowser
- CD digital audio player utility for X11/Motif
- Network Information System
- Apache WWW httpd
- Microsoft FrontPage
- Microsoft Internet Explorer
- Netscape's News Server

Classifier: *Category*

This classifier attempts to identify the system component that a vulnerability belong to.

- General system software
- General system utilities
- Logging software
- Software that deals with electronic mail
- Software that deals with networking
- Cryptographic software

Classifier: *Classification*

This classifier defines the Aslam classification for the vulnerability. The Aslam classification was designed by Taimur Aslam and later refined by Ivan Krsul at the COAST laboratory [AKS96a].

- Coding Faults
  - Synchronization errors
    - Timing window/Race condition: A fault can be exploited because of a timing window between two operations.
    - Improper serialization: A fault results from improper serialization of operations.
  - Atomicity: Did the error occur when partially-modified data structures were observed by another process? Did the error occur because the code terminated with data only partially modified as part of some operation that should have been atomic?
  - Condition validation errors
    - Boundary Condition Error
      - Condition is missing
      - Condition is incorrectly specified
      - Predicate in condition is missing
    - Access Validation Error
      - Condition is missing
      - Condition is incorrectly specified
      - Predicate in condition is missing
    - Origin Validation Error
      - Condition is missing
      - Condition is incorrectly specified
      - Predicate in condition is missing
    - Input Validation Error
      - Condition is missing
      - Condition is incorrectly specified
      - Predicate in condition is missing
    - Failure to Handle Exceptional Conditions
      - Condition is missing
      - Condition is incorrectly specified
      - Predicate in condition is missing
  - Program design errors
    - Program uses relative path names
    - The use of relative path names to specify dynamically linked libraries.
  - Failure of software to authenticate that it is really communicating with the desired software or hardware module it wants to be accessing.
    - Implicit trust: For example, routine B assumes routine A's parameters are correct because routine A is a system process.
  - Emergent Faults
    - Configuration errors.
      - Program/utility installed in the wrong place.
      - Program/utility installed with incorrect setup parameters.
        - SUID/SGID shell
        - SUID/SGID programs written in PERL that don't use the "taintperl" program.
        - SUID/SGID routines that use the system(), popen(), execlp(), or execvp() calls to run

        something else.
    - Secondary storage object or program is installed with incorrect permissions.
- Environment faults are introduced when specifications are translated to code but sufficient attention is not paid to the run-time environment. Environmental faults can also occur when different modules interact in an unanticipated manner. Independently the modules may function according to specifications but an error occurs when they are subjected to a specific set of inputs in a particular configuration environment.

Classifier: *Ease of Exploit*

This classifier was originally defined from a talk given by Tom Longstaff [Lon97] and attempts to identify how easy (or how hard) it is to exploit the vulnerability.

- Simple command
- Toolkit available
- Expertise required
- Must convince a user to take an action
- Must convince an administrator to take an action

Classifier: *Impact*

This classifier attempts to identify the impact of the vulnerability. This classifier is used to define both direct and indirect impacts. Direct impacts are those that are felt immediately after the vulnerability is exploited and indirect impact are those that ultimately result from the exploitation of the vulnerability.

- Access to data
    - Access to administrative or system data
    - Access to user level data
    - Loss of data
        - System data is lost or corrupted by the exploitation of a vulnerability
        - User data is lost or corrupted by the exploitation of a vulnerability
- Execution of commands
    - Execution of administrative or system commands
        - Generalized root access
            - Internal users can obtain generalized root access
            - External users can obtain generalized root access
        - Execution of specific system commands
            - Internal users can execute specific system commands
            - External users can execute specific system commands
    - Execution of user level commands
        - A software that is running in behalf of the user can execute a user level command in violation of access controls set by administrators
        - Internal users can execute user level commands in violation of access controls set by administrators
        - External users can execute user level commands in violation of access controls set by administrators
- Execution of code
    - Execution of machine language code with system privileges
        - Internal users can execute machine language code with privileges
        - External users can execute machine language code with privileges
    - Execution of machine language code with user privileges
        - Internal users can execute machine language code
        - External users can execute machine language code

  - Execution of scripts with system privileges
    - Internal users can execute scripts with privileges
    - External users can execute scripts with privileges
  - Execution of scripts with user privileges
    - Internal users can execute scripts
    - External users can execute scripts
- Denial of service
  - System resources are exhausted
  - System resources are eliminated

Classifier: *Origin and causes*

This classifier was originally defined from a talk given by Tom Longstaff [Lon97] and attempts to identify the origins of the vulnerability.

- Lack of training
- Procedures not followed
- Problem re-introduced
- Bug fix not propagated
- Inconsistent specifications
- Debug code not removed
- From [Eis97]: Faulty assumption/model or misdirected blame.

Classifier: *Sources*

This classifier is used to identify the source of the vulnerability (i.e. where does the information we have comes from).

- Information posted to the Internet (Public!)
  - Information posted to a newsgroup
  - Information posted to a public mailing list
  - Information posted to a private mailing list
  - Information obtained through personal or private e-mail
- Information published in the literature
- Information exchanged
  - Information from Haystack Labs.

Classifier: *System*

This classifier is used to indicate the systems that are known (to us!) to have the vulnerability.

- SUN Solaris
- SUN OS
- Microsoft DOS
- Microsoft Windows 95
- Microsoft Windows NT
- BSDI Unix
- Linux Unix
- Novel UnixWare
- NetBSD Unix
- FreeBSD Unix

- MIT-distributed Athena
- Cygnus Network Security
- openVision
- SGI IRIX
- Digital OSF/1
- NEC XX-UX
- Hewlett-Packard Unix
- IBM's AIX
- OpenStep
- OSF
- Caldera
- NEC's Goah

Classifier: *Threat*

This hierarchical classifier attempts to classify the threat that vulnerabilities create and was extracted from "Current and Future Danger: A CSI Primer on Computer Crime & Information Warfare" by Richard Power [Pow96]. The classification is attributed to Don Parker of SRI International as a classification of hostile actions that your adversary could take against you.

- Threats to availability and usefulness
  - Destroy, damage or contaminate
  - Deny, prolong or delay use of access
- Threats to integrity and authenticity
  - Enter, use of produce false data
  - Modify, replace or reorder
  - Misrepresent
  - Repudiate
  - Misuse or fail to use as required
- Threats to confidentiality and possesions
  - Access
  - Disclose
  - Observe or monitor
  - Copy
  - Steal
- Exposure to threats
  - Endanger by exposure to any of the other threats

Classifier: *Vendor*

This classifier is used to identify the vendor of the systems or that the vulnerability is present on.

- Sun Microsystems, Inc.
- Microsoft
- Sillicon Graphics Inc.
- Netscape Corporation
- Berkeley Software Design, Inc.
- Data General Corporation
- FreeBSD, Inc

- Hewlett-Packard Company
- IBM Corporation
- NEC Corporation
- The Santa Cruz Operation, Inc.
- NeXT Software, Inc.
- The Open Group
- The Santa Cruz Operation (SCO)
- Caldera

Classifier: *Complexity of Exploit*

This classifier attempts to identify the complexity of the exploitation of a vulnerability, regardless of whether a script or toolkit exists for the exploitation of the vulnerability.

- Exploitation is a simple sequence of commands or instructions
- Exploitation requires a complex set or large number of commands or instructions.
- The exploitation requires timing and synchronization. Tipically requires a script that tries several times and may require slowing down the system.

8 Future Work

Although I do not plan to address these issues fully in my dissertation, my work could be expanded to the design of vulnerability databases that contain information useful for at least the following categories:

1. It is desirable to express the vulnerabilities so that tools can use these descriptions to detect the presence of vulnerabilities in a particular system.
2. It is desirable to express the vulnerabilities so that intrusion detection patterns can be generated automatically.

In the first case, it is important to discern between being able to detect vulnerabilities and being able to exploit them. Presumably, a vulnerability database oriented towards the detection of vulnerabilities would not include enough information to allow the exploitation of these vulnerabilities. The computer security community seems to agree that it is difficult to reveal any information about the vulnerability without revealing enough information that could be used to exploit it. Some have gone as far as claiming that the sole announcement of a vulnerability is enough of a hint to allow the more sophisticated hackers to produce exploit scripts. However, in practice it may be possible to structure the information to make the generation of exploit scripts as difficult as possible.

The generation of intrusion detection signatures or patterns, and in particular patterns for the IDIOT intrusion detection system[KS94, KS95b, Kum95], is a difficult and time consuming task. IDIOT patterns are encodings of vulnerabilities as Colored Petri Nets that encode the transitions that a typical vulnerability (from a class of similar vulnerabilities) takes. Development of a pattern implies detailed working knowledge of the problem and of the structure of the system being analyzed.

For example, [CDE+96, KS95a, KS94, Kum95] all describe in detail a pattern to detect the exploitation of a famous `xterm` log bug: The problem is that a user could create only one end of a named pipe and run `xterm` requesting that it create a log with the same filename as the named pipe just created. `Xterm` would attempt to comply and the `creat()` system call (attempting to create the log file) would block on the named pipe. The exploit script would then rename the named pipe to some arbitrary value, create a symbolic link from the old name of the pipe to the password file, and open the other end of the pipe (by `cat`ing the pipe). The `creat()` system call in `xterm` would then succeed and the program would continue. Unfortunately, the next system call called by the program was a `chmod()` that would change the ownership of the log file (now the password file) and because `xterm` was running setuid root, the password file would be compromised.

The IDIOT pattern that detects the exploitation of this particular vulnerability, as well as others, looks for a setuid root program that executes a `creat()` followed by a `chmod()` system call with a different file name. Unfortunately, setuid root programs that have a valid reason to create a file and change the ownership of a different file, in that order, would trigger the pattern and a false alarm would sound. If one wishes to reduce the number of false positives, a more complicated pattern, perhaps, is required.

What is interesting about this particular pattern is that it will detect the exploitation of any bug that shares the same primary characteristics. Hence, it seems logical to assume that all possible vulnerabilities of this kind require only one entry on the vulnerability database (and hence only one pattern needs to be written). In a vulnerability as simple as this, it may be feasible to comb the existing databases to search for similar problems that have been seen before. However, for more complex vulnerabilities this may be a time consuming and error prone task.

What is really needed is a classification mechanism that allows us to unequivocally group similar vulnerabilities together. In trying to address this particular issue, Taimur Aslam [Asl95] worked on a taxonomy of security faults in the Unix operating system. This work includes a classification scheme, based on a binary decision tree, that allows such grouping of vulnerabilities. Many of these classifications are rough indications of the kinds of problems that make intrusions possible. However, they are by no means detailed enough to allow the automatic generation of IDIOT patterns.

My dissertation could be expanded to the generation of more detailed classifications that could be used both for the development of pattern generators and as additional dimensions to the state space providing more detailed descriptions of vulnerabilities so feature extraction and data mining tools could be refined further.

9 Appendix A - Machine learning definitons and terminology

Test Sample Error is the perceived error rate of a classifier or learning system on the test data.

True Error is the actual error rate of a classifier or learning system when presented data it has not seen durin the training stages.

References

[A$^+$76] R.P. Abbott et al. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[act96] The Microsoft Active Platform Overview. http://www.microsoft.com/ActiveX/, October 1996.

[AKS96a] Taimur Aslam, Ivan Krsul, and Eugene Spafford. Use of A Taxonomy of Security Faults. In *19th National Information Systems Security Conference Proceedings*, Baltimore, MD, October 1996.

[AKS96b] Taimur Aslam, Ivan Krsul, and Eugene Spafford. Use of a taxonomy of security faults. Technical Report TR-96-051, Purdue University, Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, September 1996.

[And94] Ross Anderson. Why Cryptosystems Fail. Technical report, University Computer Laboratory, Cambridge, January 1994.

[Asl95] Taimur Aslam. A Taxonomy of Security Faults in the Unix Operating System. Master's thesis, Purdue University, 1995.

[aut96] Microsoft Authenticode Technology. http://www.microsoft.com/intdev/security/misf8-f.htm, 1996.

[BB96] Matt Bishop and Dave Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis, September 1996.

[BD96] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, 1996.

[Bis95] Matt Bishop. A Taxonomy of UNIX System and Network Vulnerabilities. Technical Report CSE-95-10, Department of Computer Science at the University of California at Davis, 1995.

[Bla93] Matt Blaze. A Cryptographic File System for Unix. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 9 – 16, Fairfax, VA, November 1993.

[BPC75] Richard Bibsey, Gerald Popek, and Jim Carlstead. Inconsistency of a Single Data Value over time. Technical report, Information Sciences Institute,University of Southern California, December 1975.

[Bre94] Leo Breiman. Bagging predictors. Technical Report TR 421, Department of Statistics, University of California, September 1994.

[bri95] Encyclopedia Britannica, Fifteenth Edition. Encyclopedia Britannica Online version 1.2, 1995.

[CBP75] Jim Carlstead, Richard Bibsey II, and Gerald Popek. Pattern-directed protection evaluation. Technical report, Information Sciences Institute,University of Southern California, June 1975.

[CDE$^+$96] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT - Users Guide. Technical Report TR-96-050, Purdue University, September 1996.

[Den83] Dorothy Denning. *Cryptography and Data Security.* Addison-Wesley Publishing Company, 1983.

[DH73] Richard Duda and Peter Hart. *Pattern classification and scene analysis.* Wiley, New York, 1973.

[Dor96] Dietrich Dorner. *The Logic of Failure.* Metropolitan Books, 1996.

[Eis97] Marc Eisenstadt. My hariest bug war stories. *Communications of the ACM*, 40(4), April 1997.

[Eva84] M. Evangelist. Program complexity and programming style. In *Proceedings of the International Conference of Data Engineering*, pages 534–541. IEEE, 1984.

[FS91] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Software Engineering Research Center, Purdue University, September 1991.

[FS96] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. 1996.

[Gar96] Simson Garfinkel. Could ActiveX pose a threat to national security? http://www.packet.com/packet/, Nov 1996.

[han96] Handbook of INFOSEC Terms Version 2.0. CDROM, 1996.

[Haw88] Stephen W. Hawking. *A Brief History of Time: From the Big Bang to Black Holes.* Bantam Books, 1988.

[Hed95] Sara Reese Hedberg. The Data Gold Rush. *BYTE*, October 1995.

[HS94] Marcel Holsheimer and Arno Siebes. Data Mining: The Search for Knowledge in Databases. Technical Report CS-R9406, Centrum voor Wiskunde en Informatica, 1994.

[Jr.95] Frederick P. Brooks Jr. *The Mythical Man-Month.* Addison-Wesley, 1995.

[kdd] S*i*ftware: Tools for Data Mining and Knowledge Discovery. WWW URL http://info.gte.com/~kdd/siftware.html.

[KP78] B. Kernighan and P. Plauger. *The Elements of Programming Style.* McGraw-Hill Book Company, second edition, 1978.

[KS94] Sandeep Kumar and Eugene Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *17th National Computer Security Conference*, 1994.

[KS95a] Sandeep Kumar and Eugene Spafford. A Taxonomy of Common Computer Security Vulnerabilities based on their Method of Detection. Technical report, Purdue University, 1995.

[KS95b] Sandeep Kumar and Eugene H. Spafford. A Software Architecture to Support Misuse Intrusion Detection. Technical Report CSD-TR-95-009, Purdue University, 1995.

[KS96] Ivan Krsul and Eugene Spafford. Authorship analysis: Identifying the author of a program. Technical Report TR-96-052, Purdue University, Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, September 1996.

[Kum95] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue University, August 1995.

[L$^+$93] Carl Landwher et al. A Taxonomy of Computer Program Security Flaws. Technical report, Naval Research Laboratory, November 1993.

[Lon97] Tom Longstaff. Update: Cert/cc vulnerability knowledgebase. Technical presentation at a DARPA workshop in Savannah, Georgia, February 1997.

[LR92] D. Longley and S. Rigby. An Automatic Search for Security Flaws in Key Management Schemes. *Computers & Security*, 11(1):75–89, March 1992.

[LS90] Dennis Longley and Michael Shain. The Data and Computer Security Dictionary of Standards, Concepts, and Terms, 1990.

[LS92] Matthys Levy and Mario Salvadori. *Why Buildings Fall Down.* W. W. Norton & Company, 1992.

[Mar90] Brian Marick. A survey of software fault surveys. Technical Report UIUCDCS-R-90-1651, University of Illinois at Urbana-Champaign, December 1990.

[OC90] P. Oman and C. Cook. A Taxonomy for Programming Style. In *Eighteenth Annual ACM Computer Science Conference Proceedings*, pages 244–247. ACM, 1990.

[OC91] P. Oman and C. Cook. A Programming Style Taxonomy. *Journal of Systems Software*, 15(4):287–301, 1991.

[Per84] Charles Perrow. *Normal Accidents: Living With High-Risk Technologies.* Basic Books, 1984.

[Pet85] Henry Petrosky. *To engineer is human : the role of failure in successful design.* St. Martin's Press, 1985.

[Pol] W. Timothy Polk. Automated Tools for Testing Computer System Vulnerability. Unknown if a published version of the paper exists.

[Pop69] Karl R. Popper. *Conjections and Refutations*. Routledge and Kegan Paul, 1969.

[Pow96] Richar Power. Current And Future Danger: A CSI Primer of Computer Crime & Information Warfare. CSI Bulletin, 1996.

[QCJ] J. R. Quinlan and R. M. Cemeron-Jones. Oversearching and Layered Search in Empirical Learning.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, pages 81–106, 1986.

[RN93] J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill Inc., 1993.

[Sch94] Neil Schlager. *When Technology Fails: Significant Technological Disasters, Accidents, and Failures of the Twentieth Century*. Gale Research Inc., 1994.

[SCS86] H. Dunsmore S. Conte and V. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, 1986.

[SSTG92] S. R. Snapp, S. E. Smaha, D. M. Teal, and T. Grance. The DIDS (distributed intrusion detection system) prototype. In *USENIX Association. Proceedings of the Summer 1992 USENIX Conference*, pages 227–33, Berkeley, CA, USA, June 1992. USENIX Association, USENIX Association.

[Tas78] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging, and Testing*. Prentice Hall, 1978.

[Tou94] M. Toure. An interdisciplinary approach for adding knowledge to computer security systems. In *Proceedings. The Institute of Electrical and Electronics Engineers 28th Annual 1994 International Carnahan Conference on Security Technology*, pages 158–68, New York, NY, USA, October 1994. The Institute of Electrical and Electronics Engineers, IEEE.

[Wat95] Karen Watterson. A Data Miner's Tools. *BYTE*, October 1995.

[WK91] Sholom Weiss and Casimir Kulikowski. *Computer systems that learn : classification and prediction methods from statistics, neural nets, machine learning, and expert systems*. M. Kaufmann Publishers, 1991.