Computer Immunology

Stephanie Forrest[†]

Steven A. Hofmeyr[†]

Anil Somayaji[†]

Dept. of Computer Science

University of New Mexico

Albuquerque, NM 87131-1386

forrest@cs.unm.edu

Communications of the ACM (to appear)

March 21, 1996

Natural immune systems protect animals from dangerous foreign pathogens, including bacteria, viruses, parasites, and toxins. Their role in the body is analogous to that of computer security systems in computing. Although there are many differences between living organisms and computer systems, this article argues that the similarities are compelling and could point the way to improved computer security. Improvements can be achieved by designing computer immune systems that have some of the important properties illustrated by natural immune systems. These include multi-layered protection, highly distributed detection and memory systems, diversity of detection ability across individuals, inexact matching strategies, and sensitivity to most new foreign patterns. We first give an overview of how the immune system relates to computer security. We then illustrate these ideas with two examples.

The immune system is comprised of cells and molecules.[1] Recognition of foreign protein, called antigen, occurs when immune system detectors, including T cells, B cells, and anti-

[†]Currently on leave at the MIT Artificial Intelligence Laboratory.

[1]A good source for basic immunology is [6] and a computer scientist's overview of immunology is given in http://www.cs.unm.edu/~steveah/imm-html/immune-system.html.

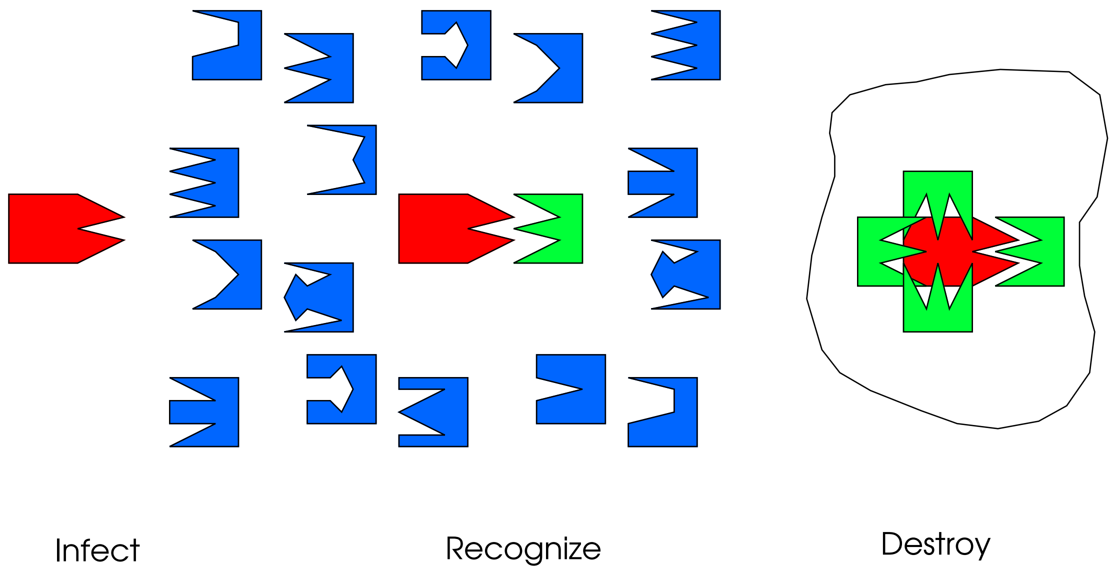

Figure 1: An Overview of the Immune System. Infections, shown in red, (bacteria, viruses, or parasites) are recognized by immune system detectors (T cells, B cells, and antibodies), shown in blue and green, when molecular bonds form between them. Infections are eliminated by general-purpose scavenger cells (macrophages), indicated by the thin line surrounding the detector/antigen complex.

bodies, bind to antigen. Binding between detector and antigen is determined by the physical and chemical properties of their binding regions. Binding is highly specific, such that each detector recognizes only a limited set of structurally related antigen. When a detector and antigen bind, a complex set of events takes place, usually resulting in the elimination of the antigen by scavenger cells called macrophages (the details of how antigen is bound and cleared depend on the type of detectors involved). Figure 1 illustrates this highly simplified view of the immune system. A striking feature of the immune system is that the processes by which it generates detectors, identifies and eliminates foreign material, and remembers the patterns of previous infections are all highly parallel and distributed. This is one reason that the immune system mechanisms are so complicated, but it also makes them highly robust, both to failure of individual components and to attacks on the immune system itself.

The analogy between computer security problems and biological processes was recognized as early as 1987, when the term "computer virus" was introduced by Adelman [1]. Later, Spafford argued that computer viruses are a form of artificial life [12], and several authors investigated the analogy between epidemiology and the spread of computer viruses across networks [10, 7]. However, current methods for protecting computers, both against viruses and many other kinds of intrusions, have largely failed to take advantage of what is known about how natural biological systems protect themselves from infection. Some initial work in this direction included a virus-detection method based on T cell censoring in the thymus [5] and an integrated approach to virus detection incorporating ideas from several different biological systems [8]. However, these early efforts are regarded largely as novelties, and the principles they illustrate have yet to be widely adopted.

Immunologists have traditionally described the problem solved by the immune system as the problem of distinguishing "self" from dangerous "other" (or "nonself") and eliminating other.[2] Self is taken to be the internal cells and molecules of the body, and nonself is any foreign material, particularly bacteria, parasites, and viruses. The problem of protecting computer systems from malicious intrusions can similarly be viewed as the problem of distinguishing self from nonself. In this case nonself might be an unauthorized user, foreign code in the form of a computer virus or worm, unanticipated code in the form of a Trojan horse, or corrupted data.

Distinguishing between self and nonself in natural immune systems is difficult for several reasons. First, the components of the body are constructed from the same basic building blocks as nonself, particularly proteins. Proteins are an important constituent of all cells, and the immune system processes them in various ways, including in fragments called peptides, which are short sequences of amino acids. Second, the size of the problem to be solved is large with respect to the available resources. For example, it has been estimated that the vertebrate immune system needs to be able to detect as many as 10^{16} different patterns, yet it only has about 10^5 different genes, out of which it must construct the entire immune

[2]The modern view emphasizes the immune system's role in eliminating infection in addition to its tolerance of self, an emphasis that is similarly important in the computer-security problem.

system (as well as everything else in the body). The difficulty of this discrimination task is shown by the fact that the immune system occasionally makes mistakes. Autoimmune diseases provide many examples of the immune system confusing self with other.

The computer security problem is also difficult. There are many legitimate changes to self (new users, new programs, etc.), many paths of intrusion, and the periphery of a networked computer is not as clearly defined as the periphery of an individual animal. Firewalls attempt to construct such a periphery, often with limited success.

The natural immune system has several distinguishing features that we believe provide important clues about how to construct robust computer security systems. These features include multi-layered protection, distributed detection, diversity across different systems, and inexact detection:

- Multi-layered protection. The body provides many layers of protection against foreign material, including passive barriers such as the skin and mucus membranes, physiological conditions such as pH and temperature, generalized inflammatory responses, and adaptive responses, including both the humoral (B cell) and cellular (T cell) mechanisms. Many computer security systems are monolithic in the sense that they define a periphery inside which all activity is trusted. Once the basic defense mechanism is violated there is rarely any backup mechanism to detect the violation. A good example is a computer security system that relies on encryption to protect data but has no mechanism for noticing if the encryption system has been broken.

- Detection is distributed. The immune system's detection and memory systems are highly distributed, and there is no centralized control that initiates or manages a response. Its success arises from highly localized interactions among individual detectors and effectors, variable cell division and death rates that allow the immune system to allcoate resources (cells) where they are most needed, and the ability to tolerate many kinds of failures, including the deletion of entire organs such as the spleen.

- Each copy of the detection system is unique. Each individual in a population has a unique set of protective cells and molecules. Computer security often involves protecting multiple sites (multiple copies of software, multiple computers on a network, etc.). In these environments, once a way is found to avoid detection at one site, then all sites become vulnerable. A better approach would be to provide each protected location with a unique set of detectors or even a unique version of software. Thus, if one site were compromised, other sites would likely remain secure.

- Detection of previously unseen foreign material. An immune system that protected us only from those diseases against which we had been vaccinated would be much less effective than one which noticed new forms of infection. Immune systems remember previous infections and mount a more agressive response against those that have been seen before. Immunologists call this a secondary response. However, in the case of a novel infection, the immune system initiates a primary response, evolving new detectors that are specialized for the infection. This process is slower than a secondary response, yet it provides an essential capability lacking in many computer security systems. Many virus and intrusion detection methods scan only for known patterns (e.g., virus signatures), which leaves systems vulnerable to attack by novel means. Some exceptions include anomaly intrusion detection systems [2] and cryptographic checksums.

- Detection is imperfect, i.e., not all antigen are well matched by a preexisting detector. The immune system uses two strategies to confront this problem—learning (during the primary response) and distributed detection (both within a single individual and across populations of individuals). Thus, high system-wide reliability is achieved at relatively low cost (in time and space) and with minimal communication among components.

What would it take to build a computer immune system with some or all of the above features? Such a system would have much more sophisticated notions of identity and protection than those afforded by current operating systems, and it would provide a general-purpose protection system to augment current computer security systems. It would have at least the following basic components: a stable definition of self, prevention or detection and subsequent elimination of dangerous foreign activities (infections), memory of previous infections, a method of recognizing new infections, and a method of protecting the immune system itself from attack.

If we want to cast the problem of computer security into the framework of distinguishing self from nonself, then the first task is to define what we mean by self and what we mean by nonself. Do we want to define self in terms of memory access patterns on a single host, TCP/IP packets entering and leaving a single host, the collective behavior of a local network of computers, network traffic through a router, instruction sequences in an executing or stored program, user behavior patterns, or even keyboard typing patterns? The immune system has evolved its recognition machinery to focus on peptides (protein fragments). Yet, it must consider many different paths of intrusion. For example, there are two quite different recognition systems in the immune system, the cell-mediated response aimed at viruses and other intra-cellular infections and the humoral response which is primarily directed at bacteria and other extra-cellular material. For computers, it is likely that self will also need to be presented in multiple ways to provide comprehensive protection.

We want our definition of self to be tolerant of many legitimate changes, including editing files, new software, new users, changes in user habits, and routine activities of system administrators. At the same time, we want it to notice unauthorized changes to files, viral software, unauthorized users, and insider attacks. In computer security parlance, we desire a system with low false positive rates and few false negatives. It is generally not possible to get perfect discrimination between legitimate and illegitimate activities. Given our bias towards multi-layered protection and adaptive responses, we are more willing to tolerate false negatives than false positives, because false negatives for one layer might be true positives for another.

Two examples of how we are applying ideas from immunology to today's computer security problems are an intrusion-detection method and a distributable change-detection algorithm. The examples highlight an important question about how analogies between biology and computer science can be applied. In one case the analogy is much more direct than in the other. Yet, both examples incorporate the basic principles elucidated earlier and support the overall vision that guides our work. The analogy between immunology and computer security is a rich one and goes well beyond the two examples presented here. For example, Kephart et al. [8] exploit the same analogy in completely different ways. The analogy with immunology contributes an important point of view about how to achieve computer security, one that can potentially lead to systems built with quite different sets of assumptions and biases than in the past.

Intrusion Detection for System Processes

As an initial step towards defining self in a realistic computing environment, we are developing an intrusion-detection system for networked computers [4]. Discrimination must be based on some characteristic structure that is both compact and universal in the protected system. The immune system's "choice" to base discrimination on patterns of peptides limits its effectiveness. For example, it cannot protect the body against radiation. However, proteins are a component of all living matter and generally differ between self and nonself, so they provide a good distinguishing characteristic.

What is the most appropriate way to define self in a computer? Most earlier work on intrusion detection monitors the behavior of individual users, but we have decided to concentrate on system processes [9]. Our "peptide" for a computer system is defined in terms of short sequences of system calls executed by privileged processes in a networked operating system. Preliminary experiments on a limited testbed of intrusions and other anomalous behavior show that short sequences of system calls (currently sequences of length 6) provide a compact signature for self that distinguishes normal from abnormal behavior.

The strategy for our intrusion-detection system is to build up a database of normal behavior for each program of interest. Each database is specific to a particular architecture, software version and configuration, local administrative policies, and usage patterns. Once a stable database is constructed for a given program in a particular environment, the database can be used to monitor the program's behavior. The sequences of system calls form the set of normal patterns for the database, and sequences not found in the database indicate anomalies. In terms of the immune system, one host (or small network of hosts) would have many different databases defining self. This is analogous to the many types of tissue in the body, each of which expresses a somewhat different set of proteins. That is, the patterns comprising self are not uniformly distributed throughout the protected system.

There are two stages to the proposed system. In the first stage, we scan traces of normal behavior and build up a database of characteristic normal patterns (observed sequences of system calls). [SHOWN IN SIDEBAR.] Parameters to system calls are ignored, and we trace forked subprocesses individually. In the second stage, we scan traces that might contain abnormal behavior, matching the trace against the patterns stored in the database. If a pattern is seen that does not occur in the normal database, it is recorded as a mismatch (see SIDEBAR). In our current implementation, tracing and analysis are performed off-line. Mismatches are the only observable that we use to distinguish normal from abnormal. We observe the number of mismatches encountered during a test trace and aggregate the information in several ways. For example, Figure 2 shows the mismatch rate through time during a successful exploit.

Do the normal databases discriminate between normal and abnormal behavior? To date, we have constructed databases of normal behavior for three different UNIX programs: `sendmail`, `wu.ftpd` (a Linux version of `ftpd`), and `lpr`. When comparing the normal database for one program (e.g., `sendmail`) against traces of normal behavior of a different program (e.g., `ls`), we observe 40 - 80 % mismatches over the length of the foreign (e.g., `ls`) trace. We also observe clear detection of several common intrusions for the three programs (mismatch rates are generally in the range of 1 to 20 % over the length of the trace). These results suggest that short sequences of system calls do provide a compact signature for normal behavior and that the signature has a high probability of being perturbed during intrusions.

Although this method does not provide a cryptographically strong or completely reliable discriminator between normal and abnormal behavior, it is much simpler than other proposed

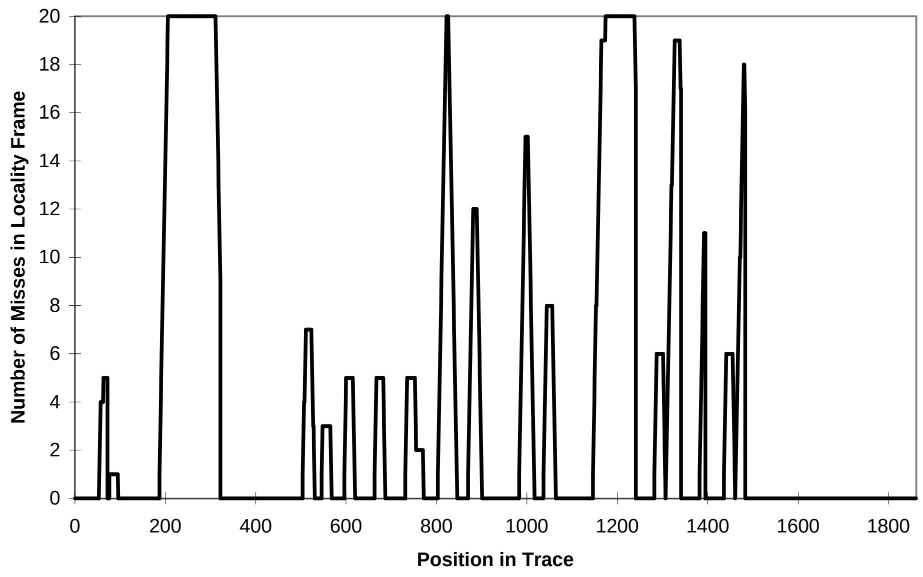

Figure 2: Anomalous signature for successful `syslog` exploit of `sendmail` under SunOS4.1.4. The normal database was generated with sequences of length 6. The x-axis measures the position in the anomalous trace in units of system calls. The y-axis shows how many mismatches were recorded when the anomalous trace was compared with the normal database. The y-axis unit of measure is total number of mismatches over the past 20 system calls in the trace (called the locality frame). That is, for position i in the trace, the locality frame records how many mismatches were observed in positions $i - 19$ through i.

methods and could potentially provide a lightweight, real-time tool for continuously checking executing code. Another appealing feature is that code that runs frequently will be checked frequently, and code that is seldom executed will be infrequently checked. Thus, system resources are devoted to protecting the most relevant code segments. Finally, given the large variability in how individual systems are currently configured, patched, and used, we conjecture that databases at different sites would likely differ enough to meet the principle of uniqueness stated earlier. This is important for another reason—it could provide a behavioral signature, or identity, for a computer that is much harder to falsify than, for example, an IP address. Our results, however, are quite preliminary, and a great deal of additional testing and development are needed before a system built on these ideas could be deployed.

Distributed Change Detection

The second example borrows more closely from mechanisms used in the immune system. T cells are an important class of detector cells in the immune system. There are several different kinds of T cells, each of which plays its own role in the immune response. All T cells, however, have binding regions that can detect antigen fragments (peptides). These binding regions are created through a pseudo-random genetic process, which we can think of analogously to generating random strings. Given that the binding regions, called receptors, are created randomly, there is a high probability that some T cells will detect self peptides. The immune system solves this problem by sending nonfunctional T cells to an organ called the thymus to mature. There are several stages of T-cell maturation, one of which is a censoring process in which T cells that bind with self proteins circulating through the thymus are destroyed. T cells that fail to bind to self are allowed to mature, leave the thymus, and become part of the active immune system. This process, called negative selection, is illustrated in Figure 3. Once in circulation, if a T cell binds to antigen in sufficient concentration, a recognition event can be said to have occurred, triggering the complex set of events that leads to elimination of the antigen.

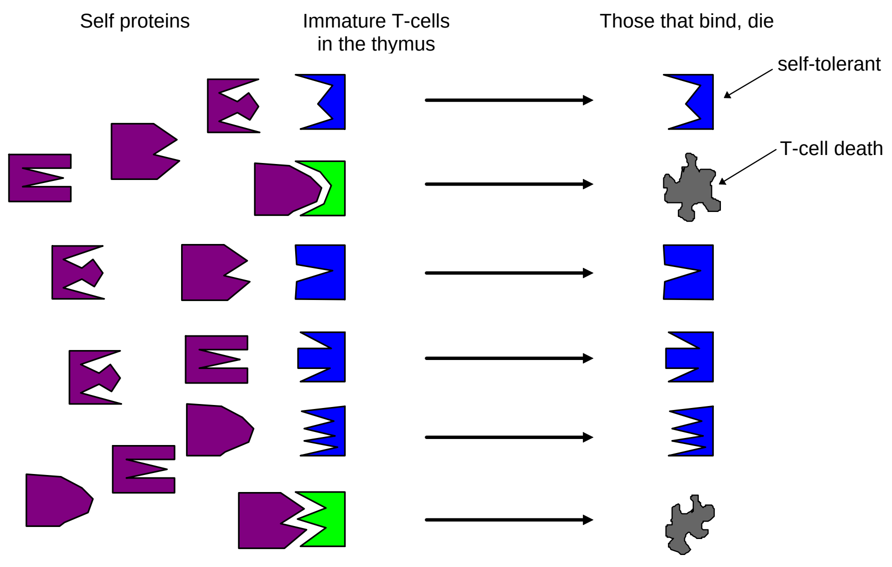

Figure 3: Censoring T cells in the thymus.

The T cell censoring process can be thought of as defining a protected collection of data (the self proteins) in terms of its complementary patterns (the nonself proteins). We can use this principle to design a distributable change-detection algorithm with interesting properties. Suppose we have some collection of digital data, which we will call self, that we wish to monitor for changes. This might be an activity pattern, as in the intrusion-detection algorithm described above, a compiled program, or a file of data. The algorithm works as follows:

1. Generate a set of detectors that fail to match self.
2. Use the detectors to monitor the protected data.
3. Whenever a detector is activated, a change must have occurred, and the location of the change is known.

There are several details that must be specified before we have an implementable algorithm: (1) how are the detectors represented? (2) How is a match defined? (3) how are detectors generated? (4) How efficient is the algorithm? These questions are explored in detail in [5, 3], but we give some highlights here.

In our computer immune system, binding between detectors and foreign patterns is modeled as string matching between pairs of strings. Self is defined as a set of equal-length strings (e.g., by logically segmenting the protected data into equal-size substrings), and each detector is defined to be a string of the same length as the substring. A perfect *match* between two strings of equal length means that at each location in the string, the symbols are identical. However, perfect matching is rare in the immune system. Partial matching in symbol strings can be defined using Hamming distance, edit distance, or a more immunologically plausible rule called r-contiguous bits [11]. This rule is based on regions of contiguous matches. It looks for r contiguous matches between symbols in corresponding positions. Thus, for any two strings x and y, we say that $match(x, y)$ is true if x and y agree at at least r contiguous locations.

Detectors can be generated in several ways. A general method (one that works for any matching rule) is also the one apparently used by the immune system. Simply generate detectors at random, compare them against self, eliminating those that match self. For the "r-contiguous bits" rule defined above, the random generating procedure is quite inefficient—roughly exponential in the size of self.[3] However, more efficient algorithms based on dynamic programming methods allow us to generate detectors in linear time for certain matching rules [3]. The total number of detectors that are required to detect nonself (using the r-contiguous bits matching rule) is the same order of magnitude as the size of self.[4]

The algorithm has several interesting properties. First, it can be easily distributed because each detector covers a small part of nonself. A set of negative detectors can be split up over multiple sites, which will reduce the coverage at any given site but provide good system-wide coverage. To achieve similar system-wide coverage with positive detection is

[3]It is interesting to note that only 2% of the immature T cells entering the thymus complete the maturation process and become functioning T cells. It is not known how much of this deletion can be attributed to negative selection, but it is thought to be sizable.

[4]This is a gross simplification. The actual number is heavily dependent on how the self set is organized, what false-negative rate we are willing to tolerate, and choice of matching rule. See [5, 3] for a more careful analysis.

much more expensive: either a nearly complete set of positive detectors will be needed at every site, resulting in multiple copies of the detection system, or the sites must communicate frequently to coordinate their results. A second point about this algorithm is that it can tolerate noise, depending on the details of how the matching function is defined. Consequently, the algorithm is likely to be more applicable to dynamic or noisy data like the intrusion-detection example than, for instance, in cryptographic applications where efficient change-detection methods already exist. The algorithm's feasibility was originally shown on the problem of computer virus detection in DOS environments [5]. In this work, the protected data were DOS system files, the self set was generated by logically segmenting .com files into equal-size substrings of 32 (binary) characters, detectors (32-bit strings) were generated randomly, the r-contiguous bits matching rule was used with thresholds ranging from 8 to 13, and infections were generated by various file-infector viruses. For example, one self set consisted of 655 self strings and was protected with essentially 100% reliability by as few as 10 detectors. Similar results were later obtained with boot-sector viruses.

Conclusion

Returning to the four principles that we discussed earlier, the intrusion-detection system could be part of a multi-layered system, for example, sitting behind cryptographic and user authentication systems. It could be distributed among sites, for example, using the negative-selection algorithm. Because the databases of normal behavior are generated empirically, based on local operating conditions, each different site will have unique protection. Finally, by focusing on anomaly intrusion detection, it trivially meets the requirement to be sensitive to new forms of attack.

We have given two concrete illustrations of how principles of immunology can be incorporated into a computer security framework. These examples represent some initial steps towards the larger intellectual vision of robust and distributed protection systems for computers. We have ignored many important complexities of the immune system, some of which will have to be incorporated before we achieve our goal. For example, it is hard to imagine how we could implement truly distributed protection without adopting the immune system strategy of self-replicating components or some of the complex molecular signaling mechanisms (e.g., interleukins) used to control the immune response. Another aspect of the analogy that has not yet been specified involves the circulation pathways by which immune cells migrate through the body. More generally, many other biological mechanisms have been incorporated into computational systems, including evolution, neural models, viruses, and parasites, many of which might be relevant to the computer security problem. In the near future, we hope to integrate the negative-selection algorithm with our intrusion-detection work, and then begin augmenting the system with other immune system features.

Although this article has stressed the similarities, there are also many important differences between computers and living systems. The success of the analogy will ultimately rest on our ability to identify the correct level of abstraction, preserving what is essential from an information-processing perspective and discarding what is not. This is complicated by the fact that natural immune systems process cells and molecules, but computer immune systems would be handling other kinds of data. In the case of a computer-vision or speech-recognition system, the input data to the system is in principle the same as that processed by the natural system—photons or sound waves. Thus, deciding exactly how to draw the analogy is a difficult task, and there are certainly many different strategies that could be tried. We have chosen to model peptides as sequences of system calls and to model binding as string matching. There are many other possible choices, some of which we hope to explore in future work.

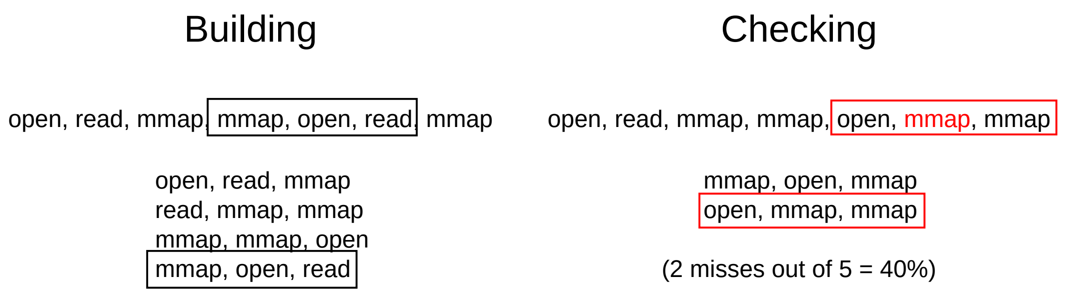

Figure 4: Sequence databases for system calls in UNIX processes. This trace of seven system calls produces a database of normal patterns that contains four unique sequences. An example anomalous trace is constructed by replacing one `read` with a `mmap` (shown in read). This is detected in the checking phase because the anomalous trace contains two subsequences that do not appear in the normal database.

Sidebar—Database of Normal Patterns

To build a database of normal patterns, we first collect a trace of system calls emitted by a normally running process. We then slide a window of size k across the trace, recording each unique k-symbol sequence. This technique goes by various names, including "time-delay embedding" and "n-gram analysis." It is illustrated in Figure 4 on a trace of seven system calls and a window size of three, resulting in a database of four unique sequences. This method differs slightly from that described in [4] and gives significantly better results.

Once the database of normal behavior has been constructed, new behavior can be monitored for anomalies, by tracing the system calls and checking them against the existing database (Figure 4). Here, a one-symbol change (read to mmap) in one position causes two mismatches in the checking procedure.

An important consideration is how to choose the normal behavior that is used to define the normal database. We have experimented with two methods, synthetic and actual. In the first case, we trace a running process while we exercise it via a set of synthetic commands. For example, we have a suite of 112 artificial email messages that we use to exercise `sendmail`, which results in a highly compact database of 891 different sequences, each of length 6.

This test suite is very useful for replicating results, comparing performance in different settings and other kinds of controlled experiments. We are also actively collecting traces of normal behavior in live user environments. These data are difficult to collect and hard to evaluate, but they provide important information about how our system is likely to perform in realistic settings, including data on false-positive rates. For example, in initial studies of `lpr` in a "live" environment, we observed some growth in database size but less than we anticipated (over a four month trial the database roughly doubled in size, from 171 distinct sequences after one month to 354 after four, mostly due to network access errors under loaded conditions).

Acknowledgments

The authors gratefully acknowledge support from the National Science Foundation (grant IRI-9157644), the Office of Naval Research (grant N00014-95-1-0364), Defense Advanced Research Projects Agency N00014-96-1-0680), Interval Research Corp., the MIT Artificial Intelligence Laboratory, and the Santa Fe Institute. Many people have contributed important ideas and suggestions for this paper, including D. Ackley, R. Brooks, P. D'haeseleer, A. Kosoresow, N. Minar, M. Opera, A. Perelson, B. Sanchez, H. Shrobe, and D. Smith. Figure 1 was prepared by R. Hightower. The data presented in Figure 2 were generated with the help of L. Rogers and T. Longstaff of the Computer Emergency Response Team (CERT).

References

[1] F. Cohen. Computer viruses. *Computers & Security*, 6:22–35, 1987.

[2] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, (2):222, February 1987.

[3] P. D'haeseleer, S. Forrest, and P. Helman. An immunological approach to change detection: algorithms, analysis and implications. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[4] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[5] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[6] C. A. Janeway and P. Travers. *Immunobiology: the immune system in health and disease*. Current Biology Ltd., London, 2nd edition, 1996.

[7] J. O. Kephart, S. R. White, and D. M. Chess. Computers and epidemiology. *IEEE Spectrum*, 30(5):20–26, 1993.

[8] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White. Biologically inspired defenses against computer viruses. In *IJCAI '95*. International Joint Conference on Artificial Intelligence, 1995.

[9] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in priviledged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 5–9 1994.

[10] W. H. Murray. The application of epidemiology to computer viruses. *Computers & Security*, 7:139–150, 1988.

[11] J. K. Percus, O. Percus, and A. S. Perelson. Predicting the size of the antibody combining region from consideration of efficient self/non-self discrimination. *Proceedings of the National Academy of Science*, 90:1691–1695, 1993.

[12] E. H. Spafford. Computer viruses—a form of artificial life? In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 727–745. Addison-Wesley, Redwood City, CA, 1992.