Computation in the Wild*

Stephanie Forrest
Justin Balthrop
Matthew Glickman
David Ackley

Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

July 18, 2002

1 Introduction

The explosive growth of the Internet has created new opportunities and risks by increasing the number of contacts between separately administered computing resources. Widespread networking and mobility has blurred many traditional computer system distinctions, including those between operating system and application, network and computer, user and administrator, and program and data. An increasing number of executable codes, including applets, agents, viruses, email attachments, and downloadable software, are escaping the confines of their original systems and spreading through communications networks. These programs coexist and coevolve with us in our world, not always to good effect. Our computers are routinely disabled by network-borne

* To appear in K. Park and W. Willinger (Eds.), The Internet as a Large-Scale Complex System. Oxford University Press.

infections, our browsers crash due to unforeseen interactions between an applet and a language implementation, and applications are broken by operating system upgrades. We refer to this situation as *computation in the wild*, by which we mean to convey the fact that software is developed, distributed, stored, and executed in rich and dynamic environments populated by other programs and computers, which collectively form a *software ecosystem*. The thesis of this chapter is that networked computer systems can be better understood, controlled, and developed when viewed from the perspective of living systems.

Taking seriously the analogy between computer systems and living systems requires us to rethink several aspects of the computing infrastructure—developing design strategies from biology, constructing software that can survive in the wild, understanding the current software ecosystem, and recognizing that all nontrivial software must evolve. There are deep connections between computation and life, so much so that in some important ways, "living computer systems" are already around us, and moreover, such systems are spreading rapidly and will have major impact on our lives and society in the future.

In this paper we outline the biological principles we believe to be most relevant to understanding and designing the computational networks of the future. Among the principles of living systems we see as most important to the development of robust software systems are: Modularity, autonomy, redundancy, adaptability, distribution, diversity, and use of disposable components. These are not exhaustive, simply the ones that we have found most useful in our own research. We then describe a prototype network intrusion-detection system, known as LISYS, which illustrates many of these principles. Finally, we present experimental data on LISYS' performance in a live network environment.

2 Computation in the Wild

In this section we highlight current challenges facing computer science and suggest that they have arisen because existing technological approaches are inadequate. Today's computers have significant and rapidly increasing commonalities with living systems, those commonalities have predictive power, and the computational principles underlying living systems offer a solid basis for secure, robust, and trustworthy operation in digital ecosystems.

2.1 Complexity, modularity, and linearity

Over the past fifty years, the manufactured computer has evolved into a highly complex machine. This complexity is managed by deterministic digital logic which performs extraordinarily complicated operations with high reliability, using modularity to decompose large elements into smaller components. Methods for decomposing functions and data dominate system design methods, ranging from object-oriented languages to parallel computing. An effective decomposition requires that the components have only limited interactions, which is to say that overall system complexity is reduced when the components are nearly independent [24]. Such well-modularized systems are "linear" in the sense that they obey an analog of the superposition principle in physics, which states that for affine differential equations, the sum of any two solutions is itself a solution (see [10] for a detailed statement of the connection). We use the term "linear" (and "nonlinear") in this sense—linear systems are decomposable into independent modules with minimal interactions and nonlinear systems are not. In the traditional view, nearly independent components are composed to create or execute a program, to construct a model, or to solve a problem. With the emergence of large-scale networks and distributed computing, and the concomitant increase in aggregate com-

plexity, largely the same design principles have been applied. Although most single computers are designed to avoid component failures, nontrivial computer networks must also survive them, and that difference has important consequences for design, as a much greater burden of autonomy and self reliance is placed on individual components within a system.

The Achilles heel of any modularization lies in the interactions between the elements, where the independence of the components, and thus the linearity of the overall system, typically breaks down. Such interactions range from the use of public interfaces in object-oriented systems, to symbol tables in compilers, to synchronization methods in parallel processes. Although traditional computing design practice strives to minimize such interactions, those component interactions that are not eliminated are usually assumed to be deterministic, reliable, and trustworthy: A compiler assumes its symbol table is correct; an object in a traditional object-oriented system doesn't ask where a message came from; a closely-coupled parallel process simply waits, indefinitely, for the required response. Thus, even though a traditionally engineered system may possess a highly decoupled and beautifully linear design, at execution time its modules are critically dependent on each other and the overall computation is effectively a monolithic, highly nonlinear entity.

As a result, although software systems today are much larger, perform more functions, and execute in more complex environments compared to a decade ago, they also tend to be less reliable, less predictable, and less secure, because they critically depend on deterministic, reliable and trustworthy interactions while operating in increasingly complex, dynamic, error prone, and threatening environments.

2.2 Autonomy, living software, and reproductive success

Eliminating such critical dependencies is difficult. If, for example, a component is designed to receive some numbers and add them up, what else can it possibly do but wait until they arrive? A robust software component, however, could have multiple simultaneous goals, of which 'performing an externally assigned task' is only one. For example, while waiting for task input a component might perform garbage collection or other internal housekeeping activities. Designing components that routinely handle multiple, and even conflicting, goals leads to robustness.

This view represents a shift in emphasis about what constitutes a practical computation, away from traditional algorithms which have the goal of finding an answer quickly and stopping, and towards processes such as operating systems that are designed to run indefinitely. In client-server computing, for example, although client programs may start and stop, the server program is designed to run forever, handling requests on demand. And increasingly, many clients are designed for open-ended execution as well, as with web browsers that handle an indefinite stream of page display requests from a user. Peer-to-peer architectures such as the ccr system [1] or Napster [18], which eliminate or blur the client/server distinction, move even farther toward the view of computation as interaction among long-lived and relatively independent processes. Traditional algorithmic computations, such as sorting, commonly exist not as stand-alone terminating computations performed for a human user, but as tools used by higher-level nonterminating computational processes. This change, away from individual terminating computations and toward loosely-coupled ecological interactions among open-ended computations, has many consequences for architecture, software design, communication protocols, error recovery, and security.

Error handling provides a clear example of the tension between these two approaches. Viewed

algorithmically, once an error occurs there is no point in continuing, because any resulting output would be unreliable. From a living software perspective, however, process termination is the last response to an error, to be avoided at nearly any cost.

As in natural living systems, successful computational systems have a variety of lifestyles other than just the 'run once and terminate' algorithms on the one hand and would be immortal operating systems and processes on the other. The highly successful open-source Apache web server provides an example of such a strategy. Following the pattern of many Unix daemons, Apache employs a form of queen & workers organization, in which a single long-lived process does nothing except spawn and manage a set of server subprocesses, each of which handles a given number of web page requests and then dies. This hybrid strategy allows Apache to amortize the time required to start each new worker process over a productive lifetime serving many page requests, while at the same time ensuring that the colony as a whole can survive unexpected fatal errors in server subprocesses. Further, by including programmed subprocess death in the architecture, analogous to cellular apoptosis (programmed cell death), Apache deals with many nonfatal diseases as well, such as memory leaks. A related idea from the software fault tolerance community is that of software rejuvenation [21], in which running software is periodically stopped and restarted again after "cleaning" of the internal state. This proactive technique is designed to counteract software aging problems arising from memory allocation, fragmentation, or accumulated state.

In systems such as the Internet, which are large, open, and spontaneously evolving, individual components must be increasingly self-reliant and autonomous, able to function without depending on a consistent global design to guarantee safe interactions. Individual components must act autonomously, protect themselves, repair themselves, and increasingly, as in the example of Apache,

decide when to replicate and when to die. Natural biological systems exhibit organizational hierarchies (e.g., cells, multi-cellular organisms, ecosystems) similar to computers, but each individual component takes much more responsibility for its own interactions than what we hitherto have required of our computations.

The properties of autonomy, robustness, and security are often grouped together under the term "survivability," the ability to keep functioning, at least to some degree, in the face of extreme systemic insult. To date, improving application robustness is largely the incremental task of growing the set of events the program is expecting—typically in response to failures as they occur. For the future of large-scale networked computation, we need to take a more basic lesson from natural living systems, that survival is more important than the conventional strict definitions of correctness. As in Apache's strategy, there may be leaks, or even outright terminal programming errors, but they won't cause the entire edifice to crash.

Moving beyond basic survival strategies, there is a longer term survivability achieved through the reproductive success of high-fitness individuals in a biological ecosystem. Successful individuals reproduce more frequently, passing on their genes to future generations. Similarly, in our software ecosystem, reproductive success is achieved when software is copied, and evolutionary dead-ends occur when a piece of software fails to be copied—as in components that are replaced (or "patched") to correct errors or extend functionality.

2.3 Disposability, adaptation, and homeostasis

As computers become more autonomous, what happens when a computation makes a mistake, or is attacked in a manner that it cannot handle? We advocate building autonomous systems from

disposable components, analogous to cells of a body that are replicating and dying continually. This is the engineering goal of avoiding single points of failure, taken to an extreme. Of course, it is difficult to imagine that individual computers in a network represent disposable components; after all, nobody is happy when it's their computer that 'dies', for whatever reason, especially in the case of a false alarm. However, if the disposability is at a fine enough grain-size, an occasional inappropriate response is less likely to be lethal. Such lower-level disposability is at the heart of Apache's survival strategy, and it is much like building reliable network protocols on top of unreliable packet transfer, as in TCP/IP. We will also see an example of disposability in the following section, when we discuss the continual turnover of detectors in LISYS, known as *rolling coverage*.

Computation today takes place in highly dynamic environments. Nearly every aspect of a software system is likely to change during its normal life cycle, including: Who uses the system and for what purpose, the specification of what the system is supposed to do, certain parts of its implementation, the implementors of the system, the system software and libraries on which the computation depends, and the hardware on which the system is deployed. These changes are routine and continual.

Adaptation and homeostasis are two important components of a robust computing system for dynamic environments. Adaptation involves a component modifying its internal rules (often called "learning") to better exploit the current environment. Homeostasis, by contrast, involves a component dynamically adjusting its interactions with the environment to preserve a constant internal state. As a textbook says, homeostasis is "the maintenance of a relatively stable internal physiological environment or internal equilibrium in an organism" [5, p. G-11].

Given the inherent difficulty of predicting the behavior of even static nonlinear systems, and

adding the complication of continuously evolving computational environments, the once plausible notion of finding *a priori* proofs of program correctness is increasingly problematic. Adaptive methods, particularly those used in biological systems, appear to be the most promising near-term approach for modeling inherent nonlinearity and tracking environmental change. The phenomenon of "bit rot" is widespread. When a small piece of a program's context changes, all too often either the user or a system administrator is required to change pathnames, apply patches, install new libraries (or reinstall old ones), etc. Emerging software packaging and installation systems such as RPM [22] or Debian's `apt-get` utility, which provide common formats for 'sporifying' software systems to facilitate mobility and reproduction, and provide corresponding developmental pathways for adapting the software to new environments; we believe this sort of adaptability should take place not just at software installation but as an ongoing process of accommodation, occurring automatically and autonomously at many levels, and that programs should have the ability to evolve their functionality over time.

Turning to homeostasis, we see that current computers already have many mechanisms that can be thought of as homeostatic, for example, temperature and power regulation at the hardware level, and virtual memory management and process scheduling at the operating systems level. Although far from perfect, they are essential to the proper functioning of almost any application program. However, mechanisms such as virtual memory management and process scheduling cannot help a machine survive when it is truly stressed. This limitation stems in part from the policy that a kernel should be fair. When processor time is shared among all processes, and memory requests are immediately granted if they can be satisfied, stability cannot be maintained when resource demands become extreme. Fairness necessitates poor performance for all processes, either

through "thrashing," when the virtual memory system becomes stressed, or through unresponsiveness, failed program invocations, and entire system crashes, when there are too many processes to be scheduled. One fair response to extreme resource contention, used by AIX [4], is random killing of processes. This strategy, however, is too likely to kill processes that are critical to the functioning of the system. Self-stabilizing algorithms [23] are similar in spirit, leading to systems in which an execution can begin in any arbitrary initial state and be guaranteed to converge to a "legitimate" state in a finite number of steps.

A more direct use of homeostasis in computing is Anil Somayaji's pH system (for process homeostasis) [25, 26]. pH is a Linux kernel extension which monitors every executing process on a computer and responds to anomalies in process behavior, either by delaying or aborting subsequent system calls. Process-level monitoring at the system-call level can detect a wide range of attacks aimed at individual processes, especially server processes such as sendmail and apache [11]. Such attacks often leave a system functional, but give the intruder privileged access to the system and its protected data. In the human body, the immune system can kill thousands of cells without causing any outward sign of illness; similarly, pH uses execution delays for individual system calls as a way to restore balance. By having the delays be proportional to the number of recent anomalous system calls, a single process with gross abnormalities will effectively be killed (e.g., delayed to the point that native time-out mechanisms are triggered), without interfering with other normally behaving processes. Likewise, small deviations from normal behavior are tolerated in the sense that short delays are transient phenomena which are imperceptible at the user level. pH can successfully detect and stop many intrusions before the target system is compromised. In addition to coping with malicious attacks, pH detects a wide range of internal system-level problems, slowing down

the offending program to prevent damage to the rest of the system.

2.4 Redundancy, diversity, and the evolutionary mess

Biological systems use a wide variety of redundant mechanisms to improve reliability and robustness. For example, important cellular processes are governed by molecular cascades with large amounts of redundancy, such that if one pathway is disrupted an alternative one is available to preserve function. Similarly, redundant encodings occur throughout genetics, the triplet coding for amino acids at the nucleotide level providing one example. These kinds of redundancy involve more than the simple strategy of making identical replicas that we typically see in redundant computer systems. Identical replicas can protect against single component failure but not against design flaws. The variations among molecular pathways or genetic encodings thus provide additional protection through the use of diverse solutions.

This diversity is an important source of robustness in biology. As one example, a stable ecosystem contains many different species which occur in highly conserved frequency distributions. If this diversity is lost and a few species become dominant, the ecosystem becomes unstable and susceptible to perturbations such as catastrophic fires, infestations, and disease. Other examples include the variations among individual immune systems in a population and other forms of genetic diversity within a single species. Computers and other engineered artifacts, by contrast, are notable for their lack of diversity. Software and hardware components are replicated for several reasons: Economic leverage, consistency of behavior, portability, simplified debugging, and cost of distribution and maintenance. All these advantages of uniformity, however, become potential weaknesses when they replicate errors or can be exploited by an attacker. Once a method is created

for penetrating the security of one computer, all computers with the same configuration become similarly vulnerable [13]. Unfortunately, lack of diversity also pervades the software that is intended to defend against attacks, be it a firewall, an encryption algorithm, or a computer virus detector. The potential danger grows with the population of interconnected and homogeneous computers.

Turning to evolution, we see that the history of manufactured computers is a truly evolutionary history, and evolution does not anticipate, it reacts. To the degree that a system is large enough and distributed enough that there is no effective single point of control for the whole system, we must expect evolutionary forces—or 'market forces'—to be significant. In the case of computing, this happens both at the technical level through unanticipated uses and interactions of components as technology develops and at the social level from cultural and economic pressures. Having humans in the loop of an evolutionary process, with all their marvelous cognitive and predictive abilities, with all their philosophical ability to frame intentions, does not necessarily change the nature of the evolutionary process. There is much to be gained by recognizing and accepting that our computational systems resemble naturally evolving systems much more closely than they resemble engineered artifacts such as bridges or buildings. Specifically, the strategies that we adopt to understand, control, interact with, and influence the design of computational systems will be different once we understand them as ongoing evolutionary processes.

In this section we described several biological design principles and how they are relevant to computer and software systems. These include modularity, redundancy, distribution, autonomy, adaptability, diversity, and use of disposable components. As computers have become more complex and interconnected, these principles have become more relevant, due to an increasing variety

of interactions between software components and a rising number of degrees of freedom for variation in computational environments. Software components are now confronted with challenges increasingly similar to those faced by organic entities situated in a biological ecosystem. An example of how biological organisms have evolved to cope with their environments is the immune system. In the next section we explore a software framework that is specifically patterned after the immune system. In addition to exhibiting many of the general biological design principles advocated above, this framework illustrates a specific set of mechanisms derived from a particular biological system and applied to a real computational problem.

3 An Illustration: LISYS

In the previous section, we outlined an approach to building and designing computer systems that is quite different from that used today. In this section we illustrate how some of these ideas could be implemented using an artificial immune system framework known as ARTIS and its application to the problem of network intrusion detection in a system known as LISYS. ARTIS and LISYS were originally developed by Steven Hofmeyr in his dissertation [16, 15].

The immune system processes peptide patterns using mechanisms that in some cases correspond closely to existing algorithms for processing information (e.g., the genetic algorithm), and it is capable of exquisitely selective and well-coordinated responses, in some cases responding to fewer than ten molecules. Some of the techniques used by the immune system include learning (affinity maturation of B-cells, negative selection of B- and T-cells, and evolved biases in the germline), memory (cross-reactivity and the secondary response), massively parallel and distributed computations with highly dynamic components (on the order of 10^8 different varieties of

receptors [27] and 10^7 new lymphocytes produced each day [19]), and the use of combinatorics to address the problem of scarce genetic resources (V-region libraries). Not all of these features are included in ARTIS, although they could be (affinity maturation and V-region libraries are the most notable lacunae).

ARTIS is intended to be a somewhat general artificial immune system architecture, which can be applied to multiple application domains. In the interest of brevity and concreteness, we will describe the instantiation of ARTIS in LISYS, focusing primarily on how it illustrates our ideas about computation in the wild. For more details about LISYS as a network intrusion-detection system, the reader is referred to [16, 3, 2].

3.1 The Network Intrusion Problem

LISYS is situated in a local-area broadcast network (LAN) and used to protect the LAN from network-based attacks. In contrast with switched networks, broadcast LANs have the convenient property that every location (computer) sees every packet passing through the LAN.[1] In this domain, we are interested in building a model of normal behavior (known as *self* through analogy to the normally occurring peptides in living animals) and detecting deviations from normal (known as *nonself*). Self is defined to be the set of normal pairwise TCP/IP connections between computers, and non-self is the set of connections, potentially an enormous number, which are not normally observed on the LAN. Such connections may represent either illicit attacks or significant changes in network configuration or use. A connection can occur between any two computers in the LAN

[1]There are several ways in which the architecture could be trivially modified to run in switched networks. For example, some processing could be performed on the switch or router, SYN packets could be distributed from the switch to the individual nodes, or a combination could be tried.

Figure 1: The 49-bit compression scheme used by LISYS to represent TCP SYN packets.

as well as between a computer in the LAN and an external computer. It is defined in terms of its "data-path triple"—the source IP address, the destination IP address, and the port by which the computers communicate [17, 14].

LISYS examines only the headers of IP packets. Moreover, at this point we restrict LISYS to examining only a restricted set of features contained in the headers of TCP SYN packets. The connection information is compressed to a single 49-bit string that unambiguously defines the connection. This string is compressed in two ways [16]. First, it is assumed that one of the IP addresses is always internal, so only the final byte of this address needs to be stored. Secondly, the port number is also compressed from 16 bits to 8 bits. This is done by re-mapping the ports into several different classes. Figure 1 shows the 49-bit representation.

3.2 LISYS Architecture

LISYS consists of a distributed set of *detectors*, where each detector is a 49-bit string (with the above interpretation) and a small amount of local state. The detectors are distributed in the sense that they reside on multiple hosts; there exists one *detector set* for each computer in the network. Detector sets perform their function independently, with virtually no communication. Distributing detectors makes the system more robust by eliminating the single point of failure associated with centralized monitoring systems. It also makes the system more scalable as the size of the network being protected increases.

A perfect *match* between a detector and a compressed SYN packet means that at each location in the 49-bit string, the symbols are identical. However, in the immune system matching is implemented as binding affinities between molecules, where there are only stronger and weaker affinities, and the concept of perfect matching is ill defined. To capture this LISYS uses a partial matching rule known as r-contiguous bits matching [20]. Under this rule, two strings match if they are identical in at least r contiguous locations. This means there are $l - r + 1$ windows where a match can occur, where l is the string length. The value r is a threshold which determines the specificity of detectors, in that it controls how many strings can potentially be matched by a single detector. For example, if $r = l$ (49 in the case of LISYS' compressed representation), the match is maximally specific, and a detector can match only a single string—itself. As shown in [9], the number of strings a detector matches increases exponentially as the value of r decreases.

LISYS uses *negative detection* in the sense that valid detectors are those that fail to match the normally occurring behavior patterns in the network. Detectors are generated randomly and are initially *immature*. Detectors that match connections observed in the network during the *tolerization period* are eliminated. This procedure is known as *negative selection* [12]. The tolerization period lasts for a few days, and detectors that survive this period become *mature*. For the r-contiguous bits matching rule and fixed self sets which don't change over time, the random detector generation algorithm is inefficient—the number of random strings that must be generated and tested is approximately exponential in the size of self. More efficient algorithms based on dynamic programming methods allow us to generate detectors in linear time [6, 7, 29, 28, 30] . However, when generating detectors asynchronously for a dynamic self set, such as the current setting, we have found that random generation works sufficiently well. Negative detection allows LISYS to be distributed,

because detectors can make local decisions independently with no communication to other nodes. Negative selection of randomly generated detectors allows LISYS to be adaptive to the current network condition and configuration.

LISYS also uses *activation thresholds*. Each detector must match multiple connections before it is activated. Each detector records the number of times it matches (the *match count*) and raises an alarm only when the match count exceeds the activation threshold. Once a detector has raised an alarm, it returns its match count to zero. This mechanism has a time horizon: Over time the match count slowly returns to zero. Thus, only repeated occurrences of structurally similar and temporally clumped strings will trigger the detection system. The activation threshold mechanism contributes to LISYS's adaptability and autonomy, because it provides a way for LISYS to tolerate small errors that are likely to be false positives.

LISYS uses a "second signal," analogous to costimulatory mechanisms in the immune system. Once a detector is activated, it must be costimulated by a human operator or it will die after a certain time period. Detectors which are costimulated become *memory detectors*. These are long-lived detectors which are more easily activated than non-memory detectors. This secondary response improves detection of true positives, while costimulation helps control false positives.

Detectors in LISYS have a finite lifetime. Detectors can die in several ways. As mentioned before, immature detectors die when they match self, and activated detectors die if they do not receive a costimulatory signal. In addition, detectors have a fixed probability of dying randomly on each time step, with the average lifespan of a single detector being roughly one week. Memory detectors are not subject to random death and may thus survive indefinitely. However, once the number of memory detectors exceeds a specified fraction of the total detector population, some are