

AFRL-IF-RS-TR-2006-105
Final Technical Report
March 2006



COMPUTATION IN THE WILD: MOVING BEYOND THE METAPHOR

University of New Mexico

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. K541

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-105 has been reviewed and is approved for publication.

APPROVED: /s/

FRANK H. BORN
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE MARCH 2006	3. REPORT TYPE AND DATES COVERED Final Sep 00 – Sep 05	
4. TITLE AND SUBTITLE COMPUTATION IN THE WILD: MOVING BEYOND THE METAPHOR			5. FUNDING NUMBERS C - F30602-00-2-0584 PE - 62301E PR - TASK TA - 00 WU - 06	
6. AUTHOR(S) Stephanie Forrest, David H. Ackley				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of New Mexico 1 University of New Mexico Albuquerque New Mexico 87131-0001			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive Arlington Virginia 22203-1714			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-105	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Frank J. Born/IFGA/(315) 330-4726/ Frank.Born@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) This report presents an overview of the research performed and technical results of the "Computation in the Wild: Moving Beyond the Metaphor" project, which was part of the DARPA TASK program. The work was performed between September 2000 and September 2005, at the Department of Computer Science at the University of New Mexico. The goal of the work was to demonstrate concrete examples and develop theoretical models of the metaphor of biology in computation. During the course of the project, significant progress was made in several principal areas, including: Developing homeostatic operating systems, software genetics, modeling agent behavior, information immune systems, and automated diversity. This report describes our research results in each of those areas.				
14. SUBJECT TERMS Agent based systems, information immune systems, homeostatic operating systems				15. NUMBER OF PAGES 28
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. Executive summary	1
1.1. Project Objective.....	1
1.2. Research focus areas	2
1.3. Highlights of Research Results	3
1.3.1. The homeostasis project.....	3
1.3.2. The software genetics project	3
1.3.3. The agent modeling project	3
1.3.4. The information immune systems project.....	4
1.3.5. Automated diversity project.....	4
2. Process Homeostasis (pH)	4
3. Software Genetics	6
3.1. Evolving modular code	7
3.2. Analyzing and improving real code.....	9
3.2.1. Metrics on language features: Likelihood and impact.....	12
3.2.2. Metrics on modularity: Breadth and weight	14
4. Modeling Agent Behavior.....	16
5. Information Immune Systems.....	17
5.1. Experiments	17
5.2. Theory	19
5.3. New Applications.....	20
6. Automated Diversity	21
7. Conclusion	21
8. Publications Supported by the Project	22
9. References.....	22

List of Figures

Figure 1: A hand-constructed sample genetic program employing an automatically-defined function (ADF) to represent a modularized sub-expression. See text.	8
Figure 2: Fraction of runs, at each epoch, that contained correct solutions.....	9
Figure 3: Evolution of the files in the Jikes RVM project.....	11
Figure 4: Scatterplot of impact vs. likelihood for various element types in the three projects, on a log-log axis.	13
Figure 5: Breadth and Weight scores for two automatic modularization algorithms	15
Figure 6: Refactoring recommended by FastModPartition to best handle multiple architectures in Jikes RVM while minimizing breadth and weight.....	15

1. Executive summary

This is the Final Technical Report on Contract F30602-00-2-0584. It presents highlights and details of the work performed and technical results achieved by the ‘Computation in the Wild: Moving Beyond the Metaphor’ project at the University of New Mexico, part of the DARPA TASK program.

The project explored the hypothesis that networks of agent-based systems can be better understood, controlled, and developed when viewed from the perspective of living systems, treating the rich and dynamic network computer environment formed by diverse benign and malicious software collectively as a software ecosystem.

The world of computing has greatly changed in the last decades. The marriage of the personal computer and the Internet has led to explosive growth in the contacts between separately administered computing resources, creating new opportunities and risks. Applets, agents, viruses, email attachments, and downloadable software routinely escape the confines of their original systems and spread through communications networks. Our computers are disabled by network-borne infections; our browsers crash due to unforeseen interactions between an applet and a language implementation; our application programs are broken by operating system upgrades.

Such a world is a far cry from the carefully isolated, controlled, and managed computer systems of the past—this is *computation in the wild*. The connections between computer systems and living systems are superficially obvious and yet deep in their implications. This report presents an overview of the research objective and main research areas explored in the project work (Section 2.1), then provides additional details of each (Sections 3–7). Section 8 summarizes the project and Section 9 provides a list of publications supported by the project.

1.1. Project Objective

Taking seriously the analogy between computer systems and living systems encourages rethinking many aspects of current computing practice, ranging from operating system design to communications mechanisms to computer security. To that end, the project explored several design strategies taken from biology, seeking to construct software that can better survive in the wild, and developing a better understanding of the current and emerging software ecosystems.

Biological principles stand to unify many scattered current research efforts addressing robust operation, survivability and security, while also suggesting new avenues for research. In addition, as the size and scope of software systems continues to grow, and global computer networks continue to expand, tools and methodologies from biological research will be increasingly relevant for understanding and monitoring the results.

Overall, the project's objective was to exploit the analogy between agent-based systems and living systems, using the analogy to understand and improve agent security, adaptability, and evolvability in the face of dynamic and unpredictable environments. This objective addressed the TASK technical topic areas of agent systems modeling and well-founded agent creation tools.

1.2. Research focus areas

The investigators conducted research on several related projects to demonstrate concrete examples and to develop theoretical models of the appealing metaphor of biology in computation. The projects included: Homeostatic Operating Systems, Software Genetics, Modeling Agent Behavior, and Information Immune Systems. In the original proposal, one optional project was proposed (depending on how long the project funding lasted), based on the idea of automated diversity. The diversity project was conducted with partial funding from F30602-00-2-0584, as well as partial support from a completed seedling project F30602-02-1-0146 and NSF grant CCR-0311686 which began Aug. 2003.

The following five sections provide details on the results of each of these efforts; here we provide just the 'high concepts' of each of the major research areas:

- **Homeostatic Operating Systems:** Augmenting a computer operating system with mechanisms similar in spirit to those of biological systems, which maintain homeostasis, by coupling system-call based process monitoring to feedback mechanisms.
- **Software Genetics:** Applying quantitative methods from biology to the study of the current software ecosystem, discovering scaling relations and automated methods for improving software modularity.
- **Modeling Agent Behavior:** Studying the behavior of Java Virtual Machines running agent implementations to extract regularities and detect anomalies.
- **Information Immune Systems:** Developing new applications in collaborative design, extended an existing prototype TCP-based intrusion-detection system based on immunological principles, and developing a formal framework for anomaly detection.
- **Automated Diversity:** Developing methods by which software can automatically be made more diverse, and thus more robust to attack, with a particular emphasis on methods that do not disrupt existing correct program semantics, leading to the possibility of 'drop in' technologies to improve security and robustness.

1.3. Highlights of Research Results

1.3.1. The homeostasis project

The homeostasis project augmented a computer operating system with mechanisms similar in spirit to those of biological systems, which maintain homeostasis, by coupling system-call based process monitoring to feedback mechanisms. Under this project, software was developed that detects when programs begin behaving anomalously and tries to limit damage by delaying subsequent system calls for the anomalous process. The more anomalous the process behaves the more aggressively it is delayed. A modified Linux kernel was produced and extensive testing performed that showed the system could detect and either delay or sometimes completely disrupt a variety of attacks.

1.3.2. The software genetics project

The software genetics project sought to understand and improve the evolvability of software systems. This includes studying well-chosen small abstract problems designed to isolate key factors related to evolvability, as well as applying the resulting ideas and insights to concrete software systems. On the abstract side, one key result showed how computer code problem can evolve *automatically* to be well-modularized, in the face of an environment that changes in a partially structured manner. On the concrete side, evolvability studies of large Java systems developed metrics for measuring code evolvability and demonstrated prototypes algorithms for automatically remodularizing existing code based on the code evolutionary history.

1.3.3. The agent modeling project

The agent modeling project instrumented Java Virtual Machines to record the dynamical information exhibited by executing Java programs. By exploring patterns of method invocations, we have discovered highly regular behaviors that can in some cases be leveraged for agent security and anomaly detection.

In addition, studies of method invocation sequences and object lifetimes trying to understand their regularities and anomalies lead to the surprisingly revelation that object lifetimes can often be predicted *exactly* by the calling context of their creation, a result which may have application in future garbage collection research as well as security and anomaly detection.

1.3.4. The information immune systems project

The information immune systems project builds on an earlier intrusion detection system (LISYS) which incorporates a significant amount of textbook immunology into a computational framework. Key developments in this area included conducting new experiments to confirm our earlier results and compare the approach with other machine learning algorithms, developing a stronger theoretical understanding of its behavior, and exploring an automated response component.

1.3.5. Automated diversity project

The diversity project investigates methods by which software can be made more diverse. Our initial focus was on using diversity to improve security, and that is where our key results are for this project funding, but we also believe that in the long run, other benefits such as correctness, robustness, efficiency and so forth, will be equally important.

More details about the results obtained in each of these areas are provided in the following sections.

2. Process Homeostasis (pH)

The project developed prototype software (pH) that detects when programs begin behaving anomalously and tries to limit damage by delaying subsequent system calls for the anomalous process. The more anomalously the process behaves the more aggressively it is delayed. The software runs under Linux, monitoring all executing processes in real time at the system-call level. The software consists of a modified linux kernel and a user interface (pH-mon). It is available under GPL licensing from <http://www.cs.unm.edu/soma/pH/>.

This pH prototype was ported to a mobile robot through the Sandia Laboratories LDRD program (not directly funded by this DARPA contract).

pH monitors all the system calls (without arguments) made by an executing program on a per-process basis. That is, each time a process is invoked, we begin a new trace, logging all the system calls for that process. Thus, for every process the trace consists of an ordered list (a time-series) of the system calls it made during its execution. For commonly executed programs, especially those that run with privilege, we collect such traces over many invocations of the program, when it is behaving normally. We then use the collection of all such traces (for one program) to develop an empirical model of its normal behavior. The model that results from this training phase is imperfect because it captures only the normal activity that occurred during our observation period. It is, of course, possible that undesirable behavior occurs during the training phase and is incorporated into the model, resulting in false negatives.

Given a collection of system call traces, how do we use them to construct a model? This is an active area of research in the field of machine learning, and there are literally hundreds of good methods available to choose from, including hidden Markov models, decision trees, neural networks, and a variety of methods based on deterministic finite automata (DFAs). We chose the simplest method we could think of within the following constraints. First, the method must be suitable for on-line training and testing. That is, we must be able to construct the model “on the fly” in one pass over the data, and both training and testing must be efficient enough to be performed in real-time. Next, the method must be suitable for large alphabet sizes. Our alphabet consists of all the different system calls—typically about 200 for UNIX systems. Finally, the method must create models that are sensitive to common forms of intrusion. Traces of intrusions are often 99% the same as normal traces, with very small, temporally clumped deviations from normal behavior. In the following, we describe a simple method, which we call “time-delay embedding” and others have called “sequence-based detection”

We define normal behavior in terms of short n -grams of system calls. Conceptually, we define a small fixed size window and “slide” it over each trace, recording which calls precede the current call within the sliding window. The current call and a call at a fixed preceding window position form a “pair,” with the contents of a window of length x being represented by $x - 1$ pairs. The collection of unique pairs over all the traces for a single program constitutes our model of normal behavior for the program.

This table can be stored using a fixed-size bit array. If $|S|$ is the size of the alphabet (number of different possible system calls) and w is the window size, then we can store the complete model in a bit array of size: $|S| \times |S| \times (w-1)$. Because w is small (6 is our standard default), our current implementation uses a 200×200 byte array, with masks to access the individual bits.

At testing time, system call pairs from test traces are compared against those in the normal profile. Any system call pair (the current call and a preceding call within the current window) not present in the normal profile is called a *mismatch*. Any individual mismatch could indicate anomalous behavior (a true positive), or it could be a sequence that was not included in the normal training data (a false positive). The current system call is defined as anomalous if there are any mismatches within its window.

pH responds to anomalies by delaying system call execution. The amount of delay is an exponential function of the current LFC, regardless of whether the current call is anomalous or not. The unscaled delay for a system call is $d = 2^{\text{LFC}}$. The effective delay for a system call is $d \times \text{delay_factor}$, where *delay_factor* is another user-defined parameter. Note that delays may be disabled by setting *delay_factor* to 0. If the LFC ever exceeds the *tolerization_limit* parameter (which is 12 for the experiments described below), the training array is reset, preventing truly anomalous behavior from being incorporated into the testing array.

Because pH monitors process behavior based on the executable that is currently running, the `execve` system call causes a new profile to be loaded. Thus, if an attacker were able to subvert a process and cause it to make an `execve` call, pH might be tricked into treating the current process as normal, based on the data for the newly-loaded executable. To avoid this possibility the maximum LFC count (`maxLFC`) for a process is recorded. If `maxLFC` exceeds the *abort execve* threshold, then all `execve`'s are aborted for the anomalous process.

Once the software was developed, extensive field tests of pH were conducted in a variety of production environments (including the UNMCS departmental web and mail server, and two desktop computers), an analysis was conducted to understand how pH detects anomalies (by detecting the use of novel code paths), and pH's effectiveness was studied as a system-administration tool.

For example, we tested pH's ability to stop intrusions on one common method of intrusion—the trojan program. To do this, we patched the current Debian release of the program `login` with a common trojan and installed it as the default login program. This trojan allows a user to gain root access via a back door, activated simply by entering a special username which isn't listed in the password file. We then performed several normal logins, exercising common variations such as mistyped passwords and login names, logins as root and as a regular user, and timeout after 60 seconds. These exemplars of normal activity produced a profile consisting of 342 unique lookahead pairs out of a total of 12180 system calls (across all the exemplar logins). This profile was manually installed as the testing database. Subsequent normal logins did not produce any anomalies. We then tested the pH response to `login`'s backdoor in the following way. First, we activated the back door with *delay_factor* set to 0 (no automated response). pH reported 25 anomalies and allowed the login to succeed. Even though the automated response component was turned off, the 25 anomalies caused an unscaled delay larger than the 10,000 *tolerization_limit*; thus, the training array for `login` was reset. We then set *delay_factor* to 2, our default value for automated response. With automated response turned on, we re-executed the back door, and this caused a total delay of 43.6 seconds over the course of 12 system calls). This delay was long enough to cause the login program to timeout, thus defeating the attempted security compromise. To our knowledge, this was the first time that an anomaly intrusion detection system had been demonstrated to respond quickly enough to stop a bona-fide attack in real-time.

The results of this work are documented in the following references [21, 22].

3. Software Genetics

Computation in the wild views the relationships between evolved biological systems and manufactured computing systems as much more than happenstance or metaphor. The software genetics project obtained new results working from both sides of that connection.

For example, considering the problem of ‘code bloat’ in genetic programming in light of various biological mutation mechanisms, we developed a new mechanism for genetic programming mutation that significantly reduces evolved GP tree sizes [25].

Similarly, developing abstract mathematical models of evolvability, based on biological inspirations, allowed us to explore simple models with *adaptive evolvability*, to compare them to traditional models which most often (without even noticing) employ fixed evolvabilities, and to demonstrate concrete cases where adaptive evolvability significantly outperforms a more traditional fixed-evolvability model [5].

As a principle focus, the software genetics project developed a new framework for investigating connections between ‘evolvability’ in the sense of biological genetics and adaptive systems on the one hand, and ‘evolvability’ in the sense of software engineering and software maintenance on the other hand.

3.1. Evolving modular code

Software forms a bridge between a useful function or desired purpose and a physical computing system that is in principle capable of performing the function—but that is only its most obvious job, and not its most challenging one. Of all the myriad possible ways that any given function could be realized by a physical system, high-quality software must be organized in such a way that, insofar as it is possible, likely future changes in needs or means can be accommodated at low cost, without largely or entirely discarding the software and starting over.

Building on the framework of “Automatically-Defined Functions” in genetic programming, we show circumstances under simple, well-modularized code to solve a problem can evolve. Instead of the ‘rats nest’ of impenetrable code typical of automatic code evolution systems, we identified a mechanism whereby evolvability can itself evolve, even though there is no direct ‘fitness’ advantage for the well-structured code, in terms of its ability to solve the current problem.

We observed this evolution of evolvability in experiments using genetic programming to solve a symbolic regression problem that varies in a partially unpredictable manner. When ADFs are part of a tree’s architecture, then not only do average populations recover from periodic changes in the fitness function, but that recovery rate itself increases over time, as the trees adopt modular software designs more suited to the changing requirements of their environment.

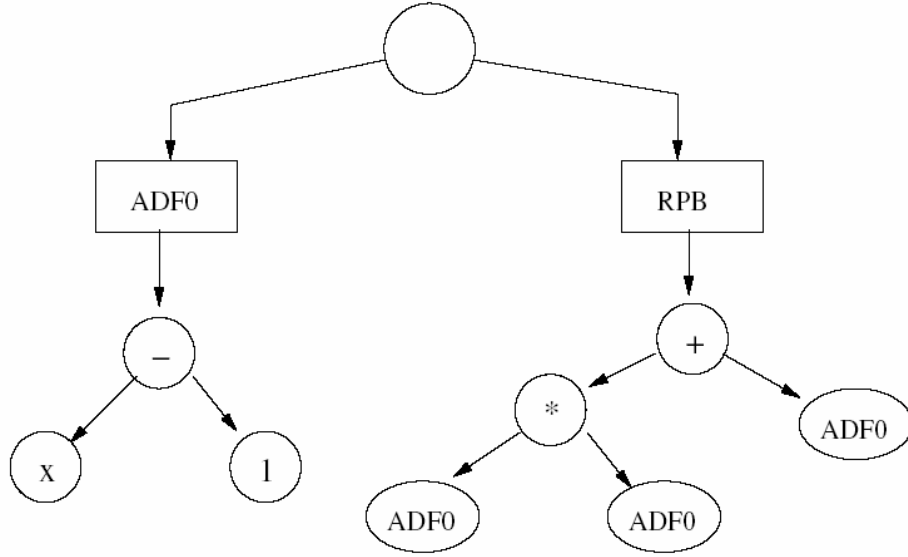


Figure 1: A hand-constructed sample genetic program employing an automatically-defined function (ADF) to represent a modularized sub-expression. See text.

The details of this research are described in [24].

Figure 1 illustrates a sample ADF-based genetic program, in that case reusing the ADF three times to compute the expression

$$y = (x - 1) * (x - 1) + (x - 1).$$

We considered a ‘problem environment’ in which the task is to evolve programs that can compute

$$y = A \sin(Ax)$$

where A is a constant—but with a twist: The value of A changes periodically over evolutionary time. Our hope was that ‘properly modularized’ code—that isolated the computation of A inside the ADF—might in the long run out-compete poorer ‘monolithic’ code solutions that perform the computation of A twice, thus requiring multiple synchronized mutations to adjust when the value of A eventually changes.

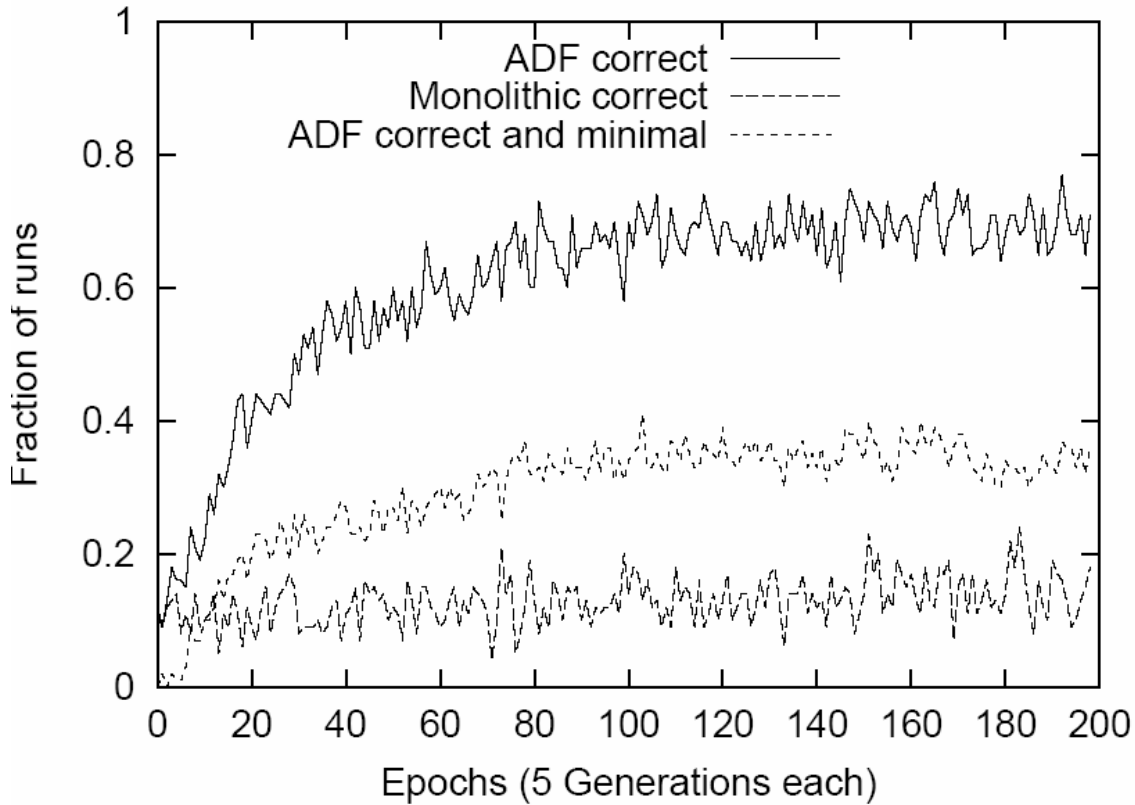


Figure 2: Fraction of runs, at each epoch, that contained correct solutions (hits = 200). Also plotted is the percent of ADF runs that contains a minimal (RPB size 6) correct solution.

Figure 2, drawn from [24] (see also [23] for more details), illustrates typical results we obtained. The horizontal axis is time (measured in 5 generation ‘epochs’) and the vertical axis is the fraction of test runs that achieved some particular criterion at each time period—note that since the environmental function changes each epoch when a new value of A is drawn, the fitness of any given individual or population is not constant. The upper curve shows that after the first 80 epochs or so, typically over 60% of runs using ADF’s were able to find a correct solution within each epoch. By contrast (bottom curve) the ‘monolithic code’ approach typically only generated correct solutions approximately 10% of the time, throughout the epochs.

3.2. Analyzing and improving real code

Software engineering experience has developed many best practices about how to produce robust, maintainable, high-quality software, but much of it is essentially ‘folk wisdom’, unsupported by any strong theory of software quality. Many traditional software engineering metrics, such as lines of code or other complexity metrics, are measured as static properties of a code base, and can therefore only hypothetically and indirectly—if at all—capture primary qualities like code maintainability over time.

As software construction tools and techniques have developed, however, and particularly with the rise of open source software and software development systems, it is now easier than ever to analyze the *actual* evolution of real software systems over significant periods of time. Historical analysis of actual patterns of code evolution support the development of *dynamic* software quality metrics—defined explicitly in terms of time and code change—and can provide a degree of ‘ground truth’ to support and refine software engineering intuition and best practices.

The ‘real software’ side of our software genetics project did just that, focusing on dynamic metrics of evolvability in the sense of software engineering and maintenance. A number of metrics were explored—see [23] for details—here we present an overview of two of them.

We developed tools to acquire and analyze codebase evolutionary data from open source code repositories, allowing us to seek out the patterns of change that *actually* occur in the development of real software, independent of any *a priori* theory about how software should be expected to change over time. For example, Figure 3, taken from [23], depicts approximately two years of the evolutionary history of the code base for the Jikes Research Virtual Machine (RVM), an experimental Java Virtual Machine implementation. Changes large and small can be seen, as well as a large range of sizes of *coordinated change*—changes happening in multiple files at (more or less nearly) the same time.

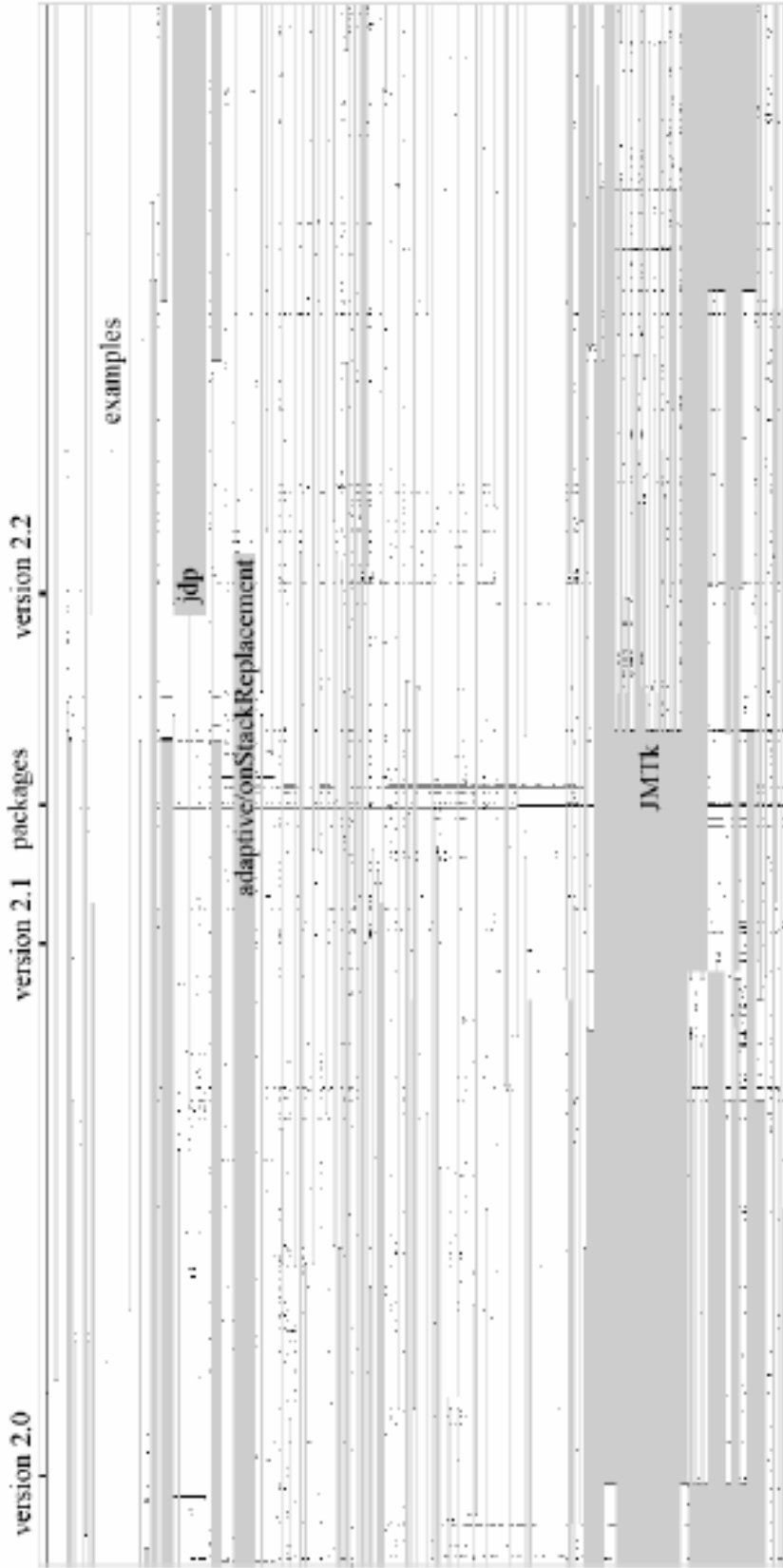


Figure 3: Evolution of the files in the Jikes RVM project. The horizontal axis is time measured in changes. The vertical axis is file pathnames, sorted alphabetically. Files that changed at a certain point in time are marked by a black dot. If the file did not exist at that point in time, it is marked with a gray dot.

3.2.1. Metrics on language features: Likelihood and impact

One way we analyzed the resulting data was in terms of two new software evolvability metrics—“impact” and “likelihood”—aimed at measuring precisely those patterns of coordinated change that occurred during the development of software systems. The metrics were used to make predictions about software changes, predicting that most changes will occur in low impact features, and that high impact features will change only rarely, roughly corresponding to major transitions in the evolutionary history of the system.

The impact metric was used to analyze three Java-based software systems—the Jikes

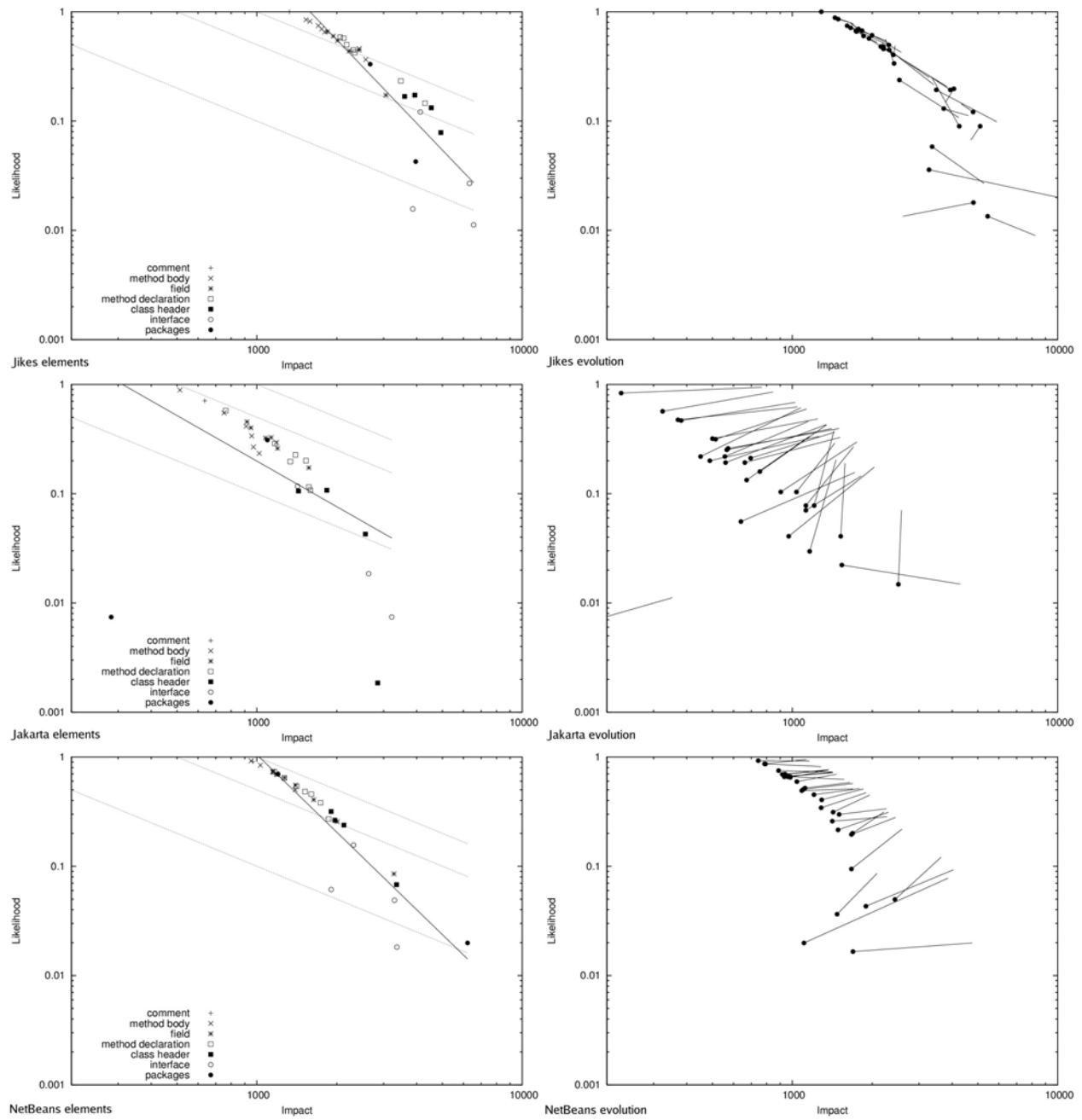


Figure 4: Scatterplot of impact vs. likelihood for various element types in the three projects, on a log-log axis. The small-dash lines represent contours of constant work. The solid line is a least-squares linear regression on the data points. The second column of graphs shows the movement of points from the first half of the project history to the second half.

RVM as mentioned above, the Jakarta web server applet, and the NetBeans IDE; see Figure 4. The analysis suggested the existence of a power law relating impact and the likelihood of a feature changing. Earlier and later software versions were also compared to see how the values of the impact and likelihood metrics change over evolutionary time: Code features in Jikes tend to evolve toward the more ‘chronic’ condition (with frequent but low-impact changes), while in the other two projects, the software evolved toward the ‘acute’ rare-but-high-impact end of the spectrum.

3.2.2. Metrics on modularity: Breadth and weight

Modularity is probably the single most important factor in software quality recognized by software engineering. In addition to examining software change at the level of individual features, we also studied the data at the level of changes in entire files—focusing in particular on changes occurring in multiple files within modules nearly simultaneously. By hypothesis, one would imagine that any particular change needed in well-modularized code would tend to be concentrated within one or perhaps a few modules, rather than requiring changes all over the code base—but how often is that really true?

At the file and module level we developed a pair of metrics called “breadth” and “weight” to measure the efficiency of evolutionary change: Breadth measures how many modules are affected by a given change, and weight measures the aggregate size of those affected modules, in terms of number of files.

Breadth and weight trade-off against each other: Breadth is minimized by putting all files in one module, but that maximizes the weight of every change, while putting each file in its own module accomplishes the reverse.

Given a modularity M , a change history C , and some chosen value for γ indicating the relative importance of breadth versus weight, then

$$\text{badness}(M,C) = \gamma \cdot \text{breadth}(M,C) + \text{weight}(M,C)$$

gives a simple scalar measure of the ‘true’ quality of a given modularization, according to the observed change history.

With such a fitness function in hand, we went even further: Beyond just measuring the badness of human-designed modularizations, we developed algorithms for *automatically generating* modularizations to minimize the badness measure, without human

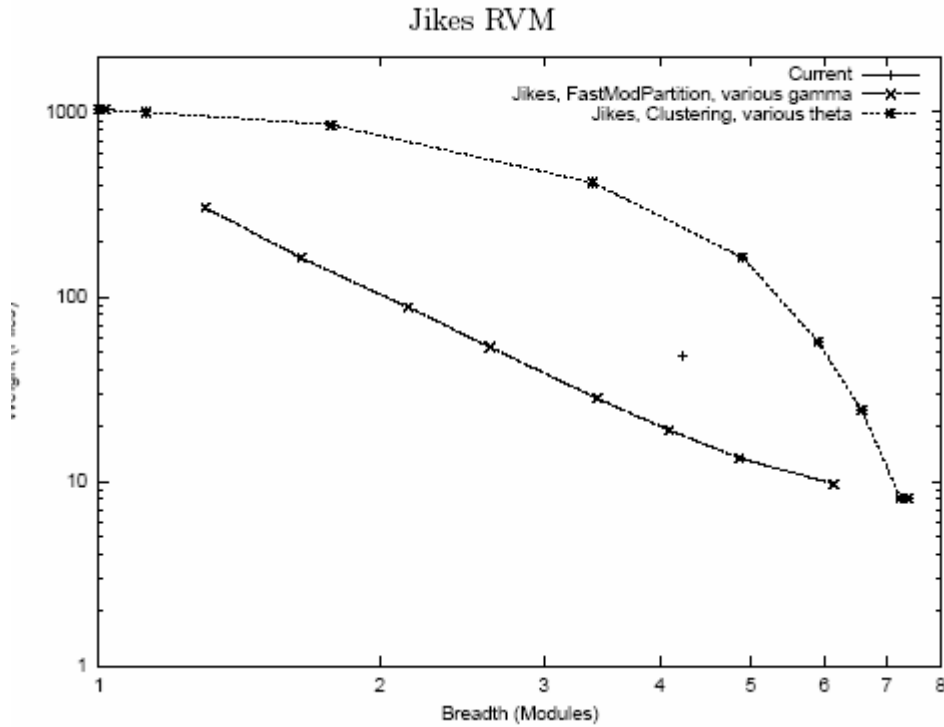


Figure 5: Breadth and Weight scores for two automatic modularization algorithms (Clustering and FastModPartition), for various tradeoff values between breadth and weight. The plus sign marks the score of the current package structure.

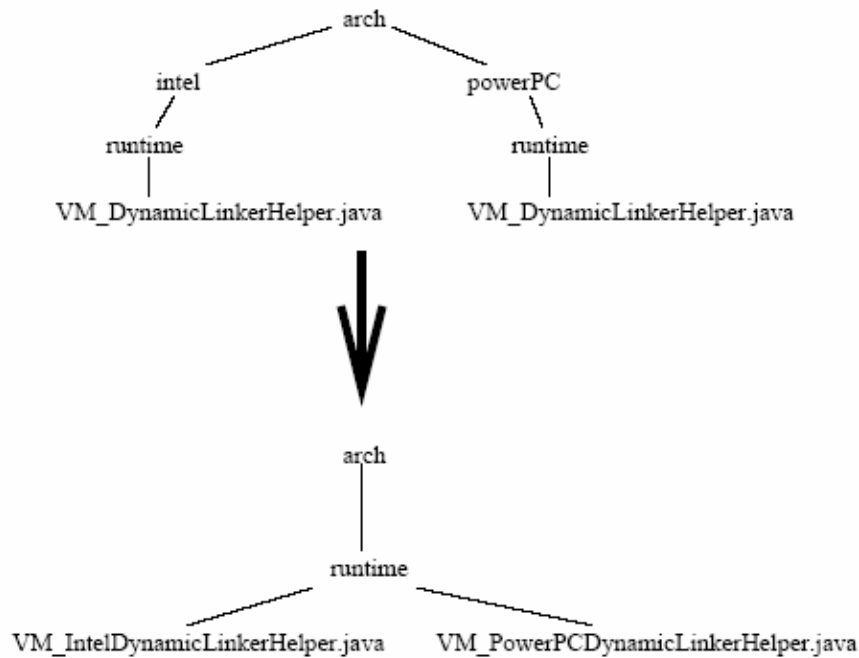


Figure 6: Refactoring recommended by FastModPartition to best handle multiple architectures in Jikes RVM while minimizing breadth and weight.

intervention beyond the choice of γ . As shown in Figure 5, with certain γ values the algorithm called ‘FastModPartition’ actually found modularizations that bettered the human-designed modularization on both breadth and weight metrics simultaneously, considering the actual evolutionary history of the Jikes project.

Although the automatically discovered modularizations are often nonsensical—because many files rarely if ever changed over the period of data collection, and so looked equally good no matter where they were placed—we also found that sometimes, the automatic discoveries were highly plausible from the point of view of traditional software engineering. For example, in the Jikes project, the automatic method suggested refactoring part of the code base to be organized according to function first, rather than architecture, as was done—as suggested by Figure 6.

4. Modeling Agent Behavior

Several methods were developed for measuring object lifetimes and patterns of method invocations in Java Virtual Machines. The instrumentation tools were used to study anomaly detection in Java programs in two quite different contexts: (1) for garbage collection, and (2) agent-based anomaly detection. A method for measuring zero lifetime objects in Java was developed, and it was used to discover that zero lifetime objects are highly predictable (that is, running a Java program once and using the observed object lifetimes as a training set produces good results on subsequent test-set runs). This predictability has potential application for garbage collection systems. An extensive series of experiments on this application was conducted and a preliminary design for a new “Death-Oriented Garbage Collector” was developed.

Modern language run-time environments provide a wealth of profiling mechanisms to investigate the state of an application. In virtual machine (VM) environments such as C# and Java, profiling is an important part of the (JIT) compilation process. One goal of object lifetime prediction is to improve performance by providing run-time advice to the memory allocation subsystem about an object’s likely lifetime at the time it is allocated. We accomplished this by constructing an object lifetime *predictor*, which based its predictions on information available at allocation time. This includes the context of the allocation request, namely the dynamic sequence of method calls that led to the request, and the actual type of the object being allocated. We refer to this information as the *allocation site*; if the observed lifetimes of all objects with the same allocation site are identical, then the predictor should predict that value at run-time for all objects allocated at the site.

In the project, we showed how information available to the VM can be leveraged to improve object lifetime prediction and how that ability might be exploited by the JIT compiler and collection system. First we demonstrated that there is a significant correlation between the state of the stack at an allocation point and the allocated object’s lifetime. Then, we showed how this information can be used to predict object lifetimes at the time they are allocated. We then showed that a significant proportion of objects have

zero lifetime (a surprising result). We also simulated the behavior of a hypothetical hybrid GC (the death-oriented collector) that uses our prediction method, showing the best-case behavior of such a system.

In a second application of these techniques, we developed a technique for automatically enforcing the principle of least privilege in Java programs. Complex enterprise and server-level applications are often written in Java because of its reputation for security. Java allows for extremely precise descriptions of security policy. The ability to express fine-grained policies makes it difficult to determine the correct policy with respect to the principle of least privilege. Our method automatically learned the minimum security policy by observing an executing program and recording the inferred policy in the standard Java policy language. We call this technique *dynamic sandboxing*. We showed that the minimum policy stops Java exploits and that learning the policy does not cripple performance, allowing realistic runs during training.

Our instrumentation methods were also applied to a large-scale simulation of UAVs, developed by Metron. Various kinds of anomalies were tested in the simulation, ranging from simulated sensor faults in the simulated UAVs to bugs in the simulation code itself. The instrumentation tools were extended for this application, to support monitoring the simulation on a per-thread basis. The analysis showed that per-thread monitoring improved anomaly-detection results dramatically for this application. With per-thread monitoring, the vast majority of the tested anomalies were detected.

The anomaly detection technology was demonstrated at the TASK Demo in Washington DC in 2004. Results from the project are documented in the following references [17, 20, 17, 19, 18].

5. Information Immune Systems

This project extended work on an earlier system (LISYS) that incorporates a significant amount of textbook immunology into a computational framework. LISYS is an artificial immune system framework that contains several interesting adaptive mechanisms. However, it was originally developed for the problem of network intrusion detection at the TCP syn packet level. Experiments were conducted to confirm earlier published results and to compare LISYS to other machine learning algorithms. Theoretical analysis was conducted leading to deeper understanding of LISYS' behavior, and several new applications beyond computer security were explored. Results from this project are summarized in three areas, Experiments, Theory, and New Applications.

5.1. Experiments

The various LISYS mechanisms were characterized in terms of their machine-learning counterparts, and a series of experiments was conducted, each of which isolated a different LISYS mechanism. The experiments were conducted on a new data set, which features one-class learning, concept drift and on-line learning. The experiments studied in

detail how LISYS achieves success on this data set and studied the potential value of LISYS' mechanisms in a broader machine learning context. With minimal parameter tuning, LISYS detected approximately 644 out of 5 total attacks) while incurring on average only 5 false positives per day (out of 6,200 total packets per day) over a 32-day period. The experiments confirmed and helped elucidate Hofmeyr's original results on the network immune system.

LISYS performance was compared to several popular machine-learning methods, including k-nearest neighbors, expectation maximization, support vector machines and compression neural networks. The experiments confirmed that concept drift is a significant issue in network intrusion detection. Two important subparts of the concept-drift problem are: adapting to changes in normal behavior and training in the presence of attacks. The empirical results showed that, for the new intrusion-detection data set, concept drift has substantial impact on performance. Failure to adapt to a non-stationary distribution of "normal" data leads to increasingly inaccurate classifications and degrades IDS performance. Two important factors were identified for tracking concept drift in the anomaly-detection domain: updating the model of the normal class and avoiding incorporating hostile data into the model. For the first factor, straightforward methods of handling drift (sliding windows in k-NN and continual training in the CNN) perform well, largely maintaining both false positive and true positive rates. Both suffer, however, when subjected to hostile data intermixed with normal during online learning. Any level of false negative error (falsely labeling anomalies as normal) leads to incorporating hostile data into the model of normalcy and degraded true positive rates. The LISYS algorithms were much more resistant to this problem than the simpler concept drift learners. These results led to the hypothesis that LISYS' use of negative detection and activation thresholds are the most important factors in its performance under the hostile training scenario. These experiments revealed that the adaptive immune system has important adaptations for learning non-stationary concepts.

LISYS performed well on a new more-challenging data set with minimal tuning. The experiments showed that (nearly) every component positively affects performance. Many of the components that help LISYS perform well in the intrusion-detection domain may prove valuable in other challenging ML domains which involve one-class learning, concept-drift, and/or on-line learning. These components include: co-stimulation and memory detectors, rolling coverage; activation thresholds and sensitivity levels, r-contiguous bits matching, and secondary representations. Co-stimulation is a mechanism for incorporating feedback from an expert teacher during operation and memory detectors are a mechanism for storing this feedback for future use. Together, they enhance LISYS' capacity for on-line learning and extend LISYS beyond a strict one-class learning framework. Rolling coverage handles concept drift. In contrast with fixed- and adaptive-window schemes, the probabilistic deletion and regeneration of detectors produces a window which is probabilistic, decaying gradually with time. Coupled with co-stimulation, the window size is also adaptive. Activation thresholds and sensitivity levels bias the system against occasional or sporadic alarms and towards high-frequency bursts of anomalies. The principal role of these mechanisms is to reduce false positives. Similar mechanisms may be useful in other anomaly-detection domains where some stream of

data is being monitored, particularly when false positives are costly. The r-contiguous bits matching rule was found to be more powerful than Hamming distance-based matching in LISYS.

This work is documented in the following papers [2, 1, 16, 15].

5.2. Theory

Mathematical characterizations were developed that account for the behavior of the various intrusion-detection systems based on the immune system metaphor. The work specifically addressed the question of when negative detection is an advantage over positive detection, when r-contiguous bits (rcb) matching rule is a good choice and why and how the immune-based methods relate to the more traditional statistical approaches to similar problems.

Specific results include:

- r-chunks matching rule: A simplification of the original immune-inspired matching rule known as “r-contiguous bits” (rcb) was developed. The new match rule is called “r-chunks.” r-chunks has four main advantages over rcb: (1) it is easier to analyze mathematically, (2) fewer detectors are required to provide the same degree of coverage, (3) it results in fewer “holes” in coverage, and (4) it has natural computational efficiencies for the case of very long detectors.

- The expected number of positive detectors to provide maximal coverage under rchunks matching is:

$$(E_p(|D|) = (l - r + 1)(2^r - 2^r (1 - 2^{-r})^{|S|}))$$

where $E_p(|D|)$ is the expected number of positive detectors, l is the length of the strings, r is the size of the window, and $|S|$ is the number of strings in self.

- The expected number of negative detectors to provide maximal coverage under r-chunks matching is:

$$E_n(|D|) = (l - r + 1) + 2^r (1 - 2^{-r})^{|S|}.$$

With these two formulae one can compute how large the self set needs to be before negative detection begins to pay off independently of the need for distributed detection (which is an advantage of negative detection regardless of the self set size). This calculation depends on l and r .

- Because the windows are overlapping, the formulae above have some redundancy in them. The size of a minimum set of negative detectors was calculated by eliminating the redundancies caused by overlaps:

$$E_{min}(|D|) = 2^r - E_r + (l - r)(E_r - 2(E_r - E_{r-1})) \text{ where } 2(E_r - E_{r-1})$$

is the number of patterns that differ only in their last bit (the redundancy). E_r is the expected number of distinct patterns in a window of size r bits.

- Generalization and the crossover closure: Under r-chunks, what strings are considered to crossover? That is, what is the generalization induced by the r-chunks matching rule? Two windows $w_i = v_i..v_{i+r-1}$ and $w_{i+1} = u_i+1..u_i+r$ are defined to “crossover” if they agree in their specified bits. The set of all crossover strings for a fixed value of r is the crossover closure (CC(r)). The generalization of a set S under r-chunks matching is exactly CC(r). The generalization of S under rcb is CC(r) plus some other strings. Thus, the generalization of S under r-chunks is smaller than under rcb, and can be controlled through the parameter r.
- Expected size of CC(r): How large is the generalization induced by r-chunks matching? This can be calculated by solving a recurrence. The final formula is quite complicated (and appears in the referenced papers). The formula was verified with simulation.
- Permutations: Permutations reduce the size of the generalization, potentially reducing false negatives. For rcb, one can study how many permutations are needed to provide the maximal reduction in the size of the generalization. As a preliminary result, the number is estimated to be about 1. Through the experiments described above it was discovered that some permutations are much more effective than others.

A formal connection between partial-match detection (the rcb rule and its relatives) and relational database theory was developed, and subsequently the consequences of this connection have been explored, focusing on the possibility that the immune-system framework could be used to enhance the privacy of relational databases under the constraints of homeland security. This line of work is being continued under NSF ITR grant CCR-0331580.

These results are documented in detail in the following references [11, 12, 13, 10].

5.3. New Applications

Several potential new application areas were explored for a LYSIS style artificial immune system.

A demonstration application of immune-inspired collaborative filtering was developed, known as Adaptive Radio. Adaptive Radio is a Linux-based music broadcaster that uses the negative-detection principle of the immune system. It is available under GPL licensing from: <http://www.cs.unm.edu/immsec>.

Matt Williamson’s work on virus throttling was extended to a more general model that can be used to control to all TCP connections in a network. This more general throttling approach was combined with a modified form of LISYS to create an adaptive desktop firewall, called RIOT. This work is documented in [8, 7, 9, 6].

6. Automated Diversity

This project deliberately introduced diversity into computers—even those running identical software—so that successful attacks on one computer will not necessarily work on others. Diversity is one aspect of the kinds of adaptive and robust methods used routinely in biological systems.

The project explored three potential diversity mechanisms: Dynamic translation of machine code (to defend against code-injection), randomizing the system-call interface (to defend against code-injection attacks), and evolving diverse implementations of TCP resource management policies (to defend against denial-of-service attacks). The primary focus, however, was on dynamic translation. Dynamic translation of machine code is used to achieve randomized instruction set emulation, which thwarts malicious code injection attacks by making injected code appear random and thus illegal to the native processor.

The project developed a prototype implementation of machine code randomization, called RISE (randomized instruction set emulation). RISE is available under GPL licensing from: <http://www.cs.unm.edu/gbarrante>. RISE's ability to stop attacks was tested using the Core Impact testing software. Experiments were conducted, both to test the effectiveness of RISE at stopping attacks and to test the safety of executing random sequences of instructions (during a code-injection attack the attack code is effectively randomized and we wanted to know how quickly and how certainly such randomized code would fail).

The project conducted a detailed analysis of RISE's safety—that is, how likely random bytes of code are to execute at all (or do damage). Experiments and developed theoretical models were developed, both for IA32 and Power PC architectures, to estimate the probabilities of various events (e.g., outright crash, infinite loop, escape into valid code, etc.). RISE succeeds in converting a very high percentage of malicious code injection attacks into no more than detectable denials of service. The results showed that for the IA32 architecture, short sequences of random bytes have a surprisingly high probability of either escaping or entering an infinite loop. The probabilities are much lower for the Power PC architecture.

Note: This project was also supported by NSF grant CCR-0311686 and by a DARPA grant. This work is documented in [3, 14, 4].

7. Conclusion

Distributed and robust computing will play a central role in almost any future networked computer system, and the project contributed a principled approach to many of the important issues. In particular, the project is relevant to the goal of deploying active agents across the Internet, both in terms of how to ensure that the agents and the environment they live in are safe and in terms of turning active code loose in an environment with no central control. The projects described above contributed several

new and important approaches, usable and widely distributed implementations, important theoretical results, and perhaps most importantly, had a significant impact on how people think about and approach the design of distributed, robust software.

A secondary benefit of this research was that it extended the bridge between biology and computation. Most research involving biology and computation falls into a few well studied areas, including neural processing, vision and speech, some aspects of robotics, computational biology, genetic algorithms, and DNA computation. Although the work reported here has much in common with such efforts seeking to apply knowledge and inspiration from biological systems to computer science, it emphasized concrete implementations in real-time systems that address real computing problems.

8. Publications Supported by the Project

9. References

- [1] J. Balthrop, F. Esponda, S. Forrest, and M. Glickman. Coverage and generalization in an artificial immune system. In W.B.Langdon, E.Cantu-Paz, K.Mathias, R. Roy, D.Davis, R. Poli, K.Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A.C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 3–10, New York, 2002. Morgan Kaufmann.
- [2] J. Balthrop, S. Forrest, and M. Glickman. Revisiting lissys: Parameters and normal behavior. In *Proceedings of the 2002 Congress on Evolutionary Computation*, 2002.
- [3] G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.
- [4] Gabriela Barrantes. *Automated methods for creating diversity in computer systems*. PhD thesis, The University of New Mexico, Albuquerque, NM, 2005.
- [5] Terry Van Belle and David H. Ackley. Adaptation and ruggedness in an evolvability landscape. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *GECCO*, volume 2723 of *Lecture Notes in Computer Science*, pages 150–151. Springer, 2003.
- [6] D. L. Chao, J. Balthrop, and S. Forrest. Adaptive radio: Achieving consensus using negative preferences. In *Proceedings of ACM Group*, 2005.

- [7] Dennis L. Chao and Stephanie Forrest. Generating biomorphs with an aesthetic immune system. In *Artificial Life VIII: Proceedings of the Eighth International Conference on the Simulation and Synthesis of Living Systems*, Cambridge, Massachusetts, 2002. MIT Press.
- [8] Dennis L. Chao and Stephanie Forrest. Information immune systems. In *International Conference on Artificial Immune Systems (ICARIS)*, pages 132–140, 2002.
- [9] Dennis L. Chao and Stephanie Forrest. Information immune systems. *Genetic Programming and Evolvable Machines*, 4(4):311–331, 2003.
- [10] F. Esponda, S. Forrest, and P. Helman. The crossover closure and partial match detection. In *Proceedings of The second International Conference on Artificial Immune Systems (ICARIS)*, number 2787 in *Lecture Notes in Computer Science*, Berlin, 2003. Springer-Verlag. Best paper award.
- [11] Fernando Esponda and Stephanie Forrest. Defining self: Positive and negative detection. Technical report, The University of New Mexico, Albuquerque, NM, 2002.
- [12] Fernando Esponda and Stephanie Forrest. Detector coverage under the r-contiguous bits matching rule. Technical report, The University of New Mexico, Albuquerque, NM, 2002.
- [13] Fernando Esponda, Stephanie Forrest, and Paul Helman. A formal framework for positive and negative detection schemes. *IEEE Transactions on System, Man, and Cybernetics*, 34(1):357–373, 2004.
- [14] G.Barrantes, D. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set randomization. *ACM Transactions on Information Systems Security (TISSEC)*, 8(1):3–40, 2005.
- [15] M. Glickman, J. Balthrop, and S. Forrest. A machine learning evaluation of an artificial immune system. *Evolutionary Computation Journal*, 13(2):179–212, 2005.
- [16] S. Forrest J. Balthrop M. Glickman and D. Ackley. Computation in the wild. In E. Jen, editor, *Robust Design: A Repertoire of Biological, Ecological, and Engineering Case Studies*, pages 207–230. Oxford University Press, 2004. Reprinted in K. Park and W. Willinger Eds. *The Internet as a Large-Scale Complex System*, pp. 227-250. Oxford University Press (2005).
- [17] H. Inoue and S. Forrest. Generic application intrusion detection. Technical Report TR-CS-2002-07, University of New Mexico, Albuquerque, NM, 2002.
- [18] H. Inoue, D. Stefanovic, and S. Forrest. On the prediction of java object lifetimes. *IEEE Transactions on Computers*, in press.

- [19] Hajime Inoue. *Anomaly detection in dynamic execution environments*. PhD thesis, The University of New Mexico, Albuquerque, NM 87131, 2005.
- [20] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the New Security Paradigms Workshop*, pages 52–60, Danvers, MA, 2003. ACM Press.
- [21] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Usenix Security Symposium*, 2000.
- [22] Anil Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, Albuquerque, NM, 2002.
- [23] Terry Van Belle. *Modularity and the Evolution of Software Evolvability*. PhD thesis, The University of New Mexico, Albuquerque, NM, 2004.
- [24] Terry Van Belle and David H. Ackley. Code factoring and the evolution of evolvability. In W. B. Langdon, E. Cant’u-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1383–1390, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
- [25] Terry Van Belle and David H. Ackley. Uniform subtree mutation. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of LNCS, pages 152–161, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.