SANDIA REPORT

SAND2005-0819
Unlimited Release
Printed March 2005

An Immunological Basis for High-Reliability Systems Control

Wendy A. Amai, Eleanor A. Walther, and Anil B. Somayaji

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND 2005-0819
Unlimited Release
Printed March 2005

An Immunological Basis for High-Reliability Systems Control

Wendy A. Amai
Mobile Robotics Department

Eleanor A. Walther
Emergent Threats Department

Sandia National Laboratories
P.O. Box 5800, MS 1125
Albuquerque, NM 87185-1125

Anil B. Somayaji
School of Computer Science
Carleton University
5302 Herzberg Building
1125 Colonel By Drive
Ottawa, ON K1S 5B6 Canada

Sandia Contract No. PO 96623

Abstract

This reports describes the successful extension of artificial immune systems from the domain of computer security to the domain of real time control systems for robotic vehicles. A biologically-inspired computer immune system was added to the control system of two different mobile robots. As an additional layer in a multi-layered approach, the immune system is complementary to traditional error detection and error handling techniques. This can be thought of as biologically-inspired defense in depth. We demonstrated an immune system can be added with very little application developer effort, resulting in little to no performance impact. The methods described here are extensible to any system that processes a sequence of data through a software interface.

Intentionally Left Blank

Contents

List of Figures

Index of Tables

Intentionally Left Blank

1.0 Introduction: Computer Immunology

Living creatures have many capabilities far beyond those of human technology. While we can easily create machines that will go faster, fly higher, or endure more extreme conditions than any animal, we are not capable of creating machines that are anywhere near as flexible or as fault-tolerant. Further, self-repair is still a dream for human technology, and self-defense remains an urgent, unmet need. Thus, as has been recognized by the first humans who attempted to fly, there is much that can be learned through the study and imitation of nature.

Computer immunology is a field that takes inspiration from the immune systems of humans and other living creatures. Natural immune systems are distributed, massively parallel, fault-tolerant networks of agents (cells) that are capable of detecting and responding to malicious agents, whether they be viruses, bacteria, parasites, or even pre-cancerous cells. Immune systems also recognize and coordinate the repair of systems damaged by these threats.

While researchers in computer immunology have applied immunological concepts to problems such as learning [1], the most compelling work in this field has come out of applications to computer security. Of particular note has been the work in intrusion detection and response by Stephanie Forrest's group at the University of New Mexico [2,3,4]. In a series of papers, Forrest's group described a method for detecting abnormal program behavior that is analogous to the mechanism used by the human immune system to detect pathogens. They also showed that this method could detect a wide variety of security attacks with few false alarms in production environments. This work has inspired many other researchers [5,6,7,8], and has served as the basis for Sana Security, multi-million dollar Bay Area start-up company founded by Steven Hofmeyr, a former student of Forrest.

Another of Forrest's former students, Anil Somayaji, developed an online implementation of their anomalous program behavior detector for his dissertation [2]. This implementation, called Process Homeostasis (pH), is a Linux kernel extension that profiles running programs as they run with no user intervention. It also implements some simple mechanisms that minimize the damage anomalously behaving programs can inflict. Appendix A contains details about pH's implementation.

This research project has been focused on applying the concepts of pH to the robotic control domain. The central insight is that while an anomalously behaving program could be evidence of an intrusion, it could also indicate a configuration error, a software bug, a hardware fault, or even human error. All of these potential problems are particularly important to detect in real-time control systems, especially when malfunctions can result in mission failure, equipment damage, and loss of life. Existing approaches to solve these problems have included extensive use of defensive programming techniques, complex error handlers, and elaborate testing regimes. Unfortunately, the cost and complexity of such methods can often exacerbate the problem, with "error handlers" themselves leading to robot failures. The hope, then, is that simple, generic mechanisms like those of pH for detecting and responding to such problems would enable the development of more reliable, flexible robots that are easier to develop and maintain.

While the domains of computer security and real-time control systems can both benefit from anomaly detection mechanisms, the basic concerns of these fields differ in fundamental ways. In computer security, the system in question is typically a networked desktop computer that is used by one or more users. One is concerned with protecting the system against misuse by users

logged on the computer as well as those accessing it remotely over the network. In contrast, robotic control systems typically consist of one or more embedded processors onboard the robot that communicate wirelessly with a base control station, often using specialized protocols. Although malicious usage can be a problem (e.g. unauthorized use), we are focused on recognizing more common problems such as hardware failures, communication glitches, and coding bugs. In the computer security domain, pH's response to anomalous behavior is to slow down the offending process in proportion to the degree of recent misbehavior. This response is not desirable in real-time control systems where a delayed action can result in catastrophic failure; however, by coupling anomaly detection with appropriate autonomous responses, ultimately we hope to increase overall system reliability with a minimal amount of operator intervention.

2.0 Immune System Design

In 1996 [3], Forrest, Hofmeyr, and Somayaji first proposed that abnormal program behavior could be automatically detected by using a simple learning algorithm inspired by the human immune system. This natural mechanism, the MHC/T-cell response, allows the immune system to detect when a cell is running "abnormal code" (e.g. when a cell has been infected by a virus). Forrest et al. developed an abstraction of this mechanism that is applicable to a wide range of computer phenomena.

Consider code that is running on a computer system. Note that normal code is a superset of all possible code execution paths, and error paths can be normal. Bugs tend to show up in code that hasn't been frequently run, or that is invoked in unusual circumstances. Exhaustive code path testing is infeasible in real-world applications under realistic conditions. So, we want to learn actual program behavior so we can distinguish between normal and abnormal. (corresponds to immunological concepts of "self" and "non-self") When we are running "abnormal" code, we want to invoke special "anomaly handlers" (like exception handlers) to check key safety issues and potentially revert to a known safe state, or at least log a message and/or warn the operator. Code behavior is very complex, so modeling all of it would be very expensive in time and space. We cannot do that in general, but we just need to know whether the code path we are on is unusual; we do not need to know exactly what it is. In other words, we ignore variable/function call state except when it affects what code is executed. We note that the execution path can be inferred by the sequential order of references to input/output (I/O) operations, which are normally carried out through an application programming interface, or API. Interesting and/or dangerous code tends to perform I/O, so we simply have to monitor the I/O API. Assuming there are enough possible API calls, we simply have to learn which calls are made, and in what order, during normal circumstances. New sequential patterns indicate anomalies. We should tend to look for short sequential patterns, since we expect much more variation over larger amounts of code (and thus, it will be harder to learn what normal behavior is.) On UNIX, the lowest-level I/O API is known as system calls but as other researchers [8] have shown, the general principle applies to other API's such as library calls and object method invocations. Forrest et al. decided to monitor program behavior by monitoring short sequences of system calls. As one would expect, the anomalies they detected, were not just security related, but related to all kinds of variations in program behavior.

This research described in the paper has been exploring the following questions: 1) Is behavior of robotic control programs regular enough that we can capture normal behavior using Forrest et al.'s methods? 2) If so, do the anomalies detected correspond to events that are of interest to robot developers and operators? 3) If so, what sort of responses might we want to implement? Could we implement "anomaly handlers" appropriate for these robotic systems? 4) Finally, can this detection (and ideally, response) be performed without degrading robot performance? Strategy: adopt Somayaji's work on pH to a robot – it is a low-overhead, minimally intrusive implementation (with respect to userspace programs). If pH can work, then the general approach should be applicable to robotic control systems. The automatic detection and handling of anomalous situations ideally will lead to increased reliability of these robotic systems by obviating the need for constant operator attention and maintenance.

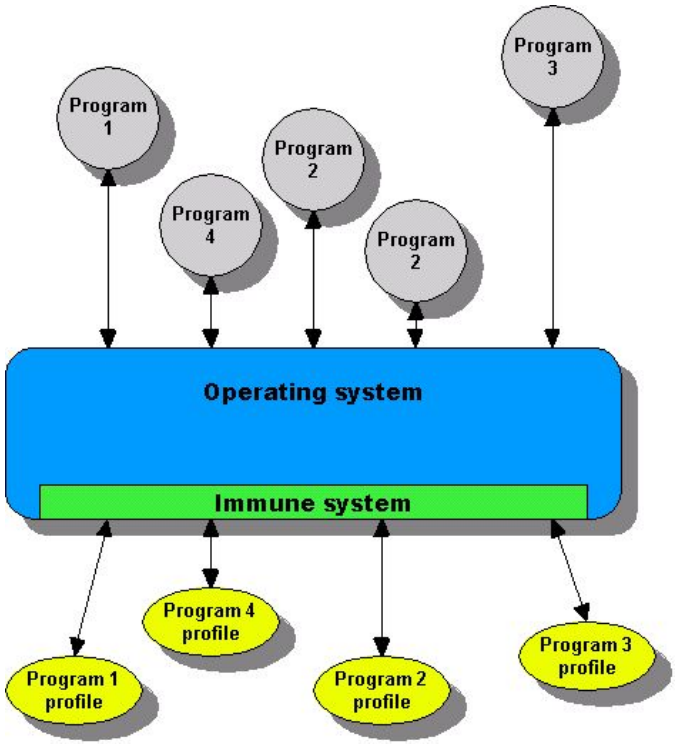


Figure 1. Computer Immune System Model

Figure 1 is a rough illustration of an immune system where each running program has a profile associated with it. We used this model of the immune system to study real-time control systems for mobile robots, using two different robots, the SandDragon and the Volant. Experimentation began with SandDragon and the concepts were later extended to the Volant. The next two sections detail the implementation of the immune system on the control system of each of the robots.

3.0 SandDragon

The SandDragon robot used for this project is the third in a series of robots based on a common platform developed by Sandia National Laboratories. This platform is a dual-body, tracked robot driven by four electric motors. See Figure 2 It is radio-controlled from an operator's control unit (OCU) which includes a radio box, a hand-held controller, and a pen computer for display of status information from the robot. See Figure 3. More information about the robot and OCU hardware can be found in Appendix B. Software details can be found in Appendix C. The SandDragon platform was selected for use in this project because it uses hardware and software typical of robots developed at Sandia National Laboratories. As such, there was existing software available for both the vehicle and the OCU. We chose to use the existing software to see how easy it would be to add an immune system.

Figure 2. SandDragon Robot

Figure 3. SandDragon OCU

3.1 Immune System

The SandDragon robot possesses a number of interfaces which could be monitored to build a model of normal robotic behavior. Table 1 lists some of these interfaces.

Interface	Description
Operator control unit communication	Commands from the operator and status from the vehicle exchanged over serial link
Sensor data	Readings from sensors over analog, digital, or serial link
Motor controller communication	Motor controller commands and status over serial link
Inter-task communication	SMART library calls
Operating system calls	Calls made by user programs to invoke operating system services

Table 1. SandDragon Control System Interfaces

One goal we had was to build a system that was independent of the specifics of the control program and robot-specific functionality. For this phase of research we ruled out the SMART

interface as being too specific for the robot platform. Other more generic robot I/O interfaces such as low-level motor controller commands and operating system calls were investigated and are detailed below.

3.1.1 Motor Controllers

The SandDragon has two motor controllers for controlling the four electric motors that propel the robot. One controller is housed in the front body half for controlling the front pair of motors, and one controller is housed in the rear body half for controlling the rear motors. The front motor controller is installed on the processor's PC104 stack. The processor communicates with the controller by writing to and reading from an address in the processor's I/O space. In contrast, the processor commands the rear motor controller over Ethernet by way of a cable housed in the umbilical between the two bodies. There are 91 distinct commands recognized by the controllers, which is enough to construct interesting profiles using the sequence monitoring technique demonstrated by Somayaji. SandDragon uses 16 of these commands.

3.1.2 SMART

When starting this research, we faced the decision of whether to use the original control program for SandDragon or develop a new one just for this project. We decided to use the original control program because it would be a realistic test for our immune system.

The control program for SandDragon was originally written using the Sandia Modular Architecture for Robotic Teleoperation (SMART) libraries and development software suite. The SMART libraries provide an abstraction layer between the user application and the operating system so that highly portable programs can be written. The SMART development suite allows the developer to create portable multi-threaded applications on several host platforms for a variety of target platforms. SMART uses the POSIX application programming interface (API) to access operating system services. A developer generates a SMART application by graphically joining together both custom and predefined software modules in the SMART Editor (SMED). Figure 4 shows the SMED grid for the SandDragon vehicle. (Note: The original SandDragon module was modified for immunology and renamed "SandTurtle" to distinguish it from other uses). In general, each graphical module includes one or more threads, or tasks, so even the very simple grid will generate a multi-threaded application with potentially complex interactions. The SandDragon control program has 10 tasks and is linked with sixteen SMART libraries containing 937 functions. There are over fifty SMART libraries extant, which together provide hundreds of functions for user applications. This suggests we could construct interesting profiles from sequences of SMART library calls. However, we ultimately decided not to monitor SMART library calls for this phase of research. We potentially would have had to instrument those hundreds of SMART library functions for complete coverage. Careful selection of a subset of these functions to instrument may be feasible for future research.

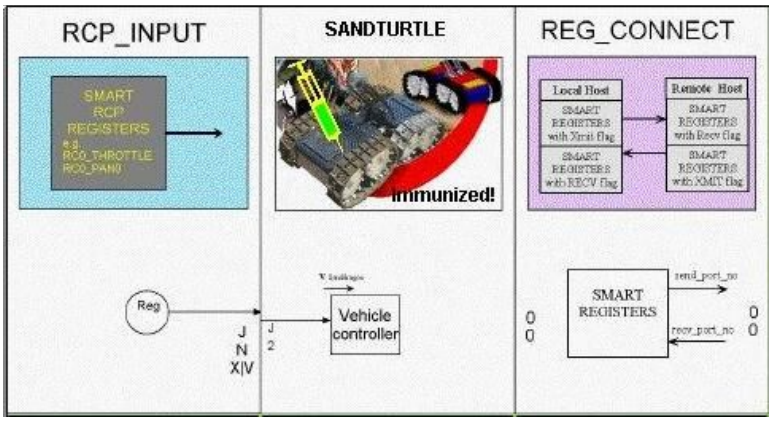


Figure 4. SMED Grid for SandDragon Robot

3.1.3 Operating System Calls

An operating system manages system resources (e.g., processor, memory, peripheral hardware access) and provides coordination services for the other processes running on a system. If a process needs access to any of these system resources or services, it makes a system call to the operating system. The operating system fulfills the request and returns the result to the calling process, if appropriate. Somayaji found that system calls provide a rich interface for monitoring. His kernel extension also ran with very little performance impact, which we believed to be acceptable for the robotic control program. We believed we would be able to leverage his existing work with pH in order to implement an immune system on the SandDragon robot.

3.1.4 Implementation

After considering the alternatives, we decided to implement the immune system by monitoring operating system calls at the kernel level. The original SandDragon software ran under the QNX real-time operating system, for which the source code is not readily available. Therefore, the immune system would have to be implemented in user space rather than in the operating system kernel space. Somayaji had a number of compelling reasons against implementing the immune system in Linux user space, such as severe performance impact and the generation of large log files. Although we could not be certain the results would be the same under QNX, we first explored the possibility of porting the control code to Linux. Since the control code was written with SMART, which makes use of the POSIX standard interface, we decided relatively few changes would be needed to port the source code to Linux. We planned to use a standard version of Linux rather than one that supported hard-real-time scheduling for simplicity and also to establish a potential "worst-case" performance baseline of the immune system.

Some changes needed to be made to the control code to run under Linux. Although only few changes needed to be made to the control code, SMART itself had to be checked thoroughly and updated to run under Linux kernel versions 2.4 and, later, 2.6.

To what degree an application exercises an interface might be used as a metric for evaluating which interface to monitor with an immune system. The Linux kernel version 2.4 has 262 system calls available, while version 2.6 has 274 system calls. The SandDragon control code

makes 49 distinct system calls, meaning that 19% and 18% of all system calls were exercised, respectively. Results indicate this was a sufficient percentage for the immune system to be able to identify anomalies.

As previously mentioned, the immune system on SandDragon is an adaptation of a Linux kernel extension called pH, originally developed as part of Anil Somayaji's dissertation. "pH" stands for process Homeostasis, which describes the method by which it attempts to maintain proper functioning of a computer system. pH monitors the system calls of all processes in a computer system, constructing a profile for each individual program.

After establishing a normal profile for a program run on the computer, all subsequent sequences of system call made by that program are looked up in the profile. If the sequence is not there, an anomaly is flagged and the response mechanism is triggered. In the original computer security application of pH, the response was to progressively slow down the offending process according to the level of misbehavior (i.e., a burst of anomalies would slow down the process more than would single anomalies spaced over time). We did not implement this response in our real-time robotic control application. Instead, we focused on identifying when the robot was experiencing problems so that the operator could be notified. Future research will consider automatic responses.

The immune system had no noticeable performance impact on the control program. The robot could be driven just as effectively with or without the control program with no lags that the operator could detect.

3.2 Monitoring

In addition to implementing an immune system on the robot, we needed a way to monitor how the immune system was performing and to log data for later study. Towards this end, we implemented a way to monitor the immune system on the robot while it was running and without having to modify the control system or OCU code explicitly to support the immune system. We also developed a valuable data logging and debugging capability.

We considered mounting a display directly on the robot, but that was inconvenient and offered no inherent data collection capabilities. Another early experiment involved triggering the beeper of the onboard computer every time an anomaly occurred, effecting a one-bit data channel. We listened for the beeps on the OCU's audio channel, but they were all but inaudible in the noise of driving. Other alternatives were to add a second radio pair or to hardwire the robot to the OCU with a serial communication cable, both cumbersome solutions. Instead, we found a way to use the existing radio data link. Ordinarily the SandDragon radio link provided only one serial data connection between the SandDragon OCU and the vehicle. This connection carried dedicated communication between the OCU and vehicle code only, and no other program on either the vehicle or the OCU could use the port. See Figure 5a. In order to make the communication link available to more than one program on each side, we increased the speed of the dedicated serial link and converted it to Point-to-Point Protocol (PPP), a protocol for tunneling the Internet Protocol (IP) over a serial link. See Figure 5b. After the switch to PPP, we modified the OCU code and vehicle control code to use network sockets. Note that the code was unchanged otherwise and was still "unaware" of the immune system.

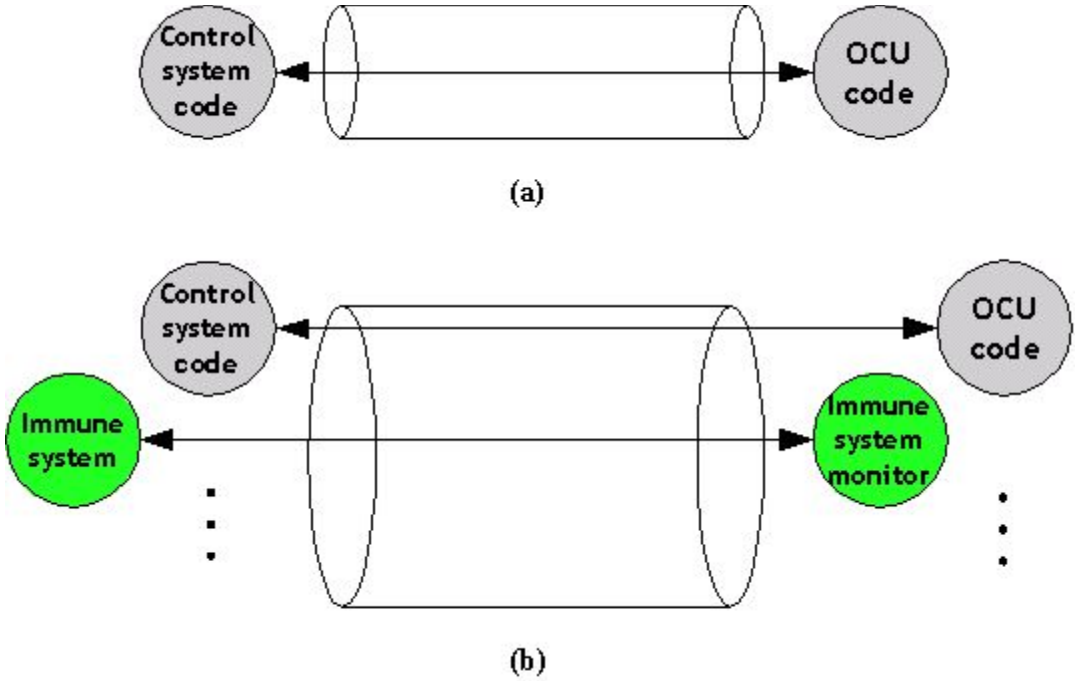


Figure 5. System Architectures. (a) Dedicated serial link; (b) PPP link

The robot-pHmon Perl program allows the operator to monitor the immune system through a graphical interface on the control console. Originally robot-pHmon was to convey only anomaly information to the operator so the anomalies could be correlated with operator actions. We have expanded robot-pHmon to provide other capabilities such as starting and stopping the robot control program, entering commands directly to the robot control program, and shutting down or rebooting the robot. See Appendix D for more information.

3.3 Experimental Results

Our experiments with the robots were aimed at determining the effect of the immune systems on performance as well as determining what useful information the immune systems could provide us. We hypothesized that we would be able to detect differences in the way human operators drove the robot, differences in operating conditions (laboratory vs. field), as well as detecting imminent system failures. Our planned test regimen with SandDragon looked like this:

1. Develop a normal profile in laboratory with robot on bench, using a single driver.
2. Test profile in field conditions with same driver. Can the change in operating environment be detected?
3. Test with different drivers, can differences between drivers be detected?

The first step involved developing a normal profile in the lab. As described previously, a profile for a program consists of the set of system call pairs which have been seen during execution of that program. After no more pairs have been added to profile, it is considered to be "normalized." Thereafter, the immune system will observe current execution of the program and will flag as anomalies any system call pairs not in the profile. In Somayaji's work, the immune

system automatically normalized a profile after a certain amount of time passed without any new

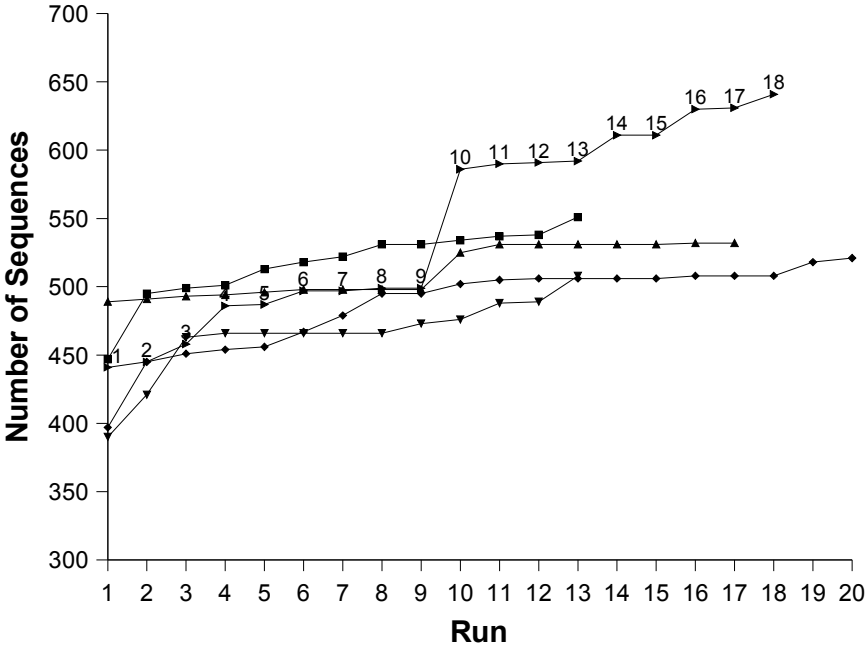


Figure 6. Series of Sequences

pairs being added. Nominally this was a week. Profiles would also be normalized automatically if more than a certain percentage of the total calls have been made without adding new pairs.

In addition to the tallies of system call pairs, a profile also records the number of sequences seen. The sequences themselves are not stored, for reasons discussed in [1]. Due to a misunderstanding, the number of sequences rather than pairs is the metric we tracked during early testing. Therefore note that although the following discussion refers to the number of sequences, each profile is actually comprised of system call pairs. The overall results are not qualitatively affected because the number of sequences increases as the number of pairs does.

We put the robot up on blocks in the laboratory and ran the control program a number of times, noting the number of sequences in the profile after each run. We expected this number eventually to stop increasing the more runs we made. Unfortunately, it never stopped increasing and sometimes showed large jumps even after many runs. Figure 6 shows the number of sequences in the profile after each run for several series of runs. In each series, we started with an empty profile, then successively ran the control program and noted the total number of sequences tallied in the profile. Note the continually increasing number of new sequences. We correlated some of the large jumps with the occurrence of communication errors with the rear motor controller. In particular, note the large jump between runs 9 and 10 in the numbered series at the top of the graph. Eighty-five new sequences were added due to automatic retries in establishing initial communication with the rear motor controller. After we tracked down and fixed a coding mistake in the motor controller driver, the rate at which new sequences were added was reduced. However, new sequences continued to be added even though we were running the control program in the same way each time and we expected the profile to stabilize.

New Sequence no.	***1***	***2***	***3***	***4***
Sequence of system calls (earliest system call is at top of each list)	rt_sigsuspend pipe clone write write sigreturn rt_sigsuspend sigreturn sigreturn	clone write write sigreturn rt_sigsuspend sigreturn sigreturn rt_sigprocmask write	rt_sigprocmask rt_sigsuspend write write sigreturn rt_sigsuspend sigreturn schedule_getscheduler sigreturn	write write sigreturn rt_sigsuspend write write sigreturn sigreturn read

Table 2. New System Call Sequences

We analyzed when sequences were being added to the profile and saw that many were added during startup. Typically the startup of a process is very stable and deterministic. We looked at the system calls making up the sequences that were flagged as new. Table 2 shows a few of these sequences. Most of the new sequences contained real-time signal handler suspends and returns (rt_sigsuspend, sigreturn). We realized this would cause interleaving of system calls between signal handlers and the mainline code of a given thread. It is important to note that although the robot control program consists of ten tasks, or ten separate threads of execution, the immune system is thread-aware and keeps the system call pair sequences separate for each thread. In other words, system calls made by different threads will never be paired when constructing sequences. However, the immune system was not separately tracking the system calls made by signal handlers, which themselves can be nested. The real-time signals are generated by the operating system rather than the user process and hence are not deterministic. We upgraded pH so it would separately track the system calls made by each signal handler.

Some of the new sequences contained scheduler calls which led us to scrutinize the use of the scheduling functions in the control code. We made some changes to the way these functions were called in order to make sure threads were being run according to their assigned priorities rather than according to round robin scheduling. Incorrect scheduling was probably not a source of new sequences, but it did cause inefficiency in the control code operation.

After upgrading pH and fixing the scheduling priority bug, we finally achieved a stable normal profile of the control program. See Figure 7 for a comparison of the number of sequences for two series of runs with non-signals aware and signals-aware pH. Notice that the number of sequences for the non-signals-aware pH continues to drift upwards, while it flattens out much sooner for signals-aware pH. Achieving a stable profile is important so that the profile can be normalized and can start detecting anomalies reliably.

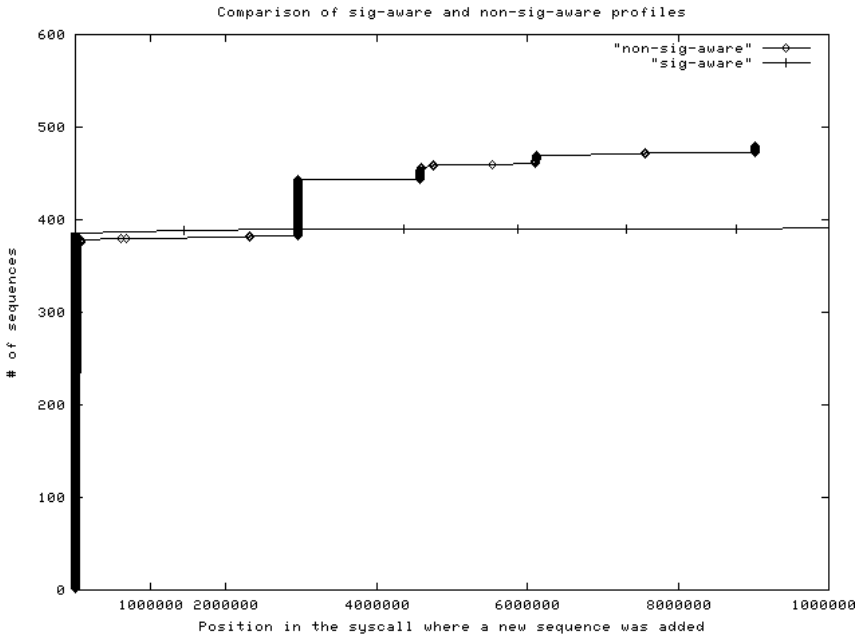


Figure 7. Non-Signal Aware vs. Signal Aware Sequence Counts

To test the ability of the immune system to detect changes in control program behavior, we normalized a profile and performed both normal and abnormal operations. Table 3 shows the typical number of anomalies produced by these operations.

Operation	***Number of Anomalies Caused***
Operate vehicle in vigorous manner	2
Startup / shutdown, non-signals aware pH	<30
Startup / shutdown, signals-aware pH	0
Print out process status	3
Interrupt communications with rear motor controller	0
Terminate OCU program	800

Table 3. Operations and Resulting Anomalies

The first four rows refer to normal operations, while the last two rows are error conditions. Operating the vehicle in a vigorous manner meant commanding various throttle settings and motor enable combinations, as might be encountered when operating in differing terrain conditions or by different operators. Interestingly, only a few sequences were added to the profile, indicating the immune system probably would not detect differences in driving style (the rate of change and the magnitude of the motor speed commands) between different people.

The non-signals aware pH typically generated less than 30 anomalies on startup and shutdown due to non-determinism in the handling of real-time signals and accounted for the slow upward

drift in the total number of sequences seen by the immune system. The signals-aware version of pH was very stable during startup and shutdown and did not register any anomalies.

Printing out process status is a normal operation but one that was not included in the profile during training. It generates a few anomalies.

We manually created an error condition by interrupting ethernet communications with the rear motor controller. The results were puzzling. Error messages were appearing on the control program console, but there were no anomalies registered by the immune system. This indicates the sequences of system calls were previously seen and were incorporated into the normal profile. This will require further investigation, but highlights the fact the immune system is not infallible but constitutes part of a layered defense.

We also interrupted communications with the operator control unit by terminating the OCU program. This also is an error condition. The immune system immediately flagged hundreds of anomalies, an order of magnitude seen when operating normally. This was a clear indication something was amiss.

4.0 Volant

We believe that smaller robotic systems, those with much less computing power than the SandDragon, can also benefit from having an immune system. To test this hypothesis, we experimented with an immune system on the Volant robot.

The Volant is a tracked robot about one ninth the mass of SandDragon. See Figure 8. In addition to being physically much smaller than the SandDragon, the Volant features a less-powerful 8-bit microprocessor for high-level control. Like SandDragon, Volant is teleoperated by way of radio modem from an operator control unit.

Figure 8. Volant Robot

As with SandDragon, the question arose of what interface would be best to monitor with the least amount of disruption to the system. The software for Volant includes the user control program (which we wrote) and a set of libraries (provided with the compiler) that provide access to the processor resources such as digital I/O, timer, and interrupt facilities. There are 22 system libraries with over 260 functions. These are merely user-level library functions rather than kernel routines as in Linux, so there is no system call dispatcher. In other words, there is no single place into which we could hook an immune system for processing all of the system calls made by the control program. Each library routine would have to be individually instrumented. We decided against this course of action in favor of a simpler solution.

We wrote the Volant control code to execute under the MicroC/OS-II real-time operating system, which is implemented as a library that is linked with control code. The control code is downloaded as a monolithic image to the flash memory in the robot, and it is the only program run when the robot is powered on. There still was no single location into which we could hook an immune system but there were many fewer system functions to instrument. The MicroC/OS-II library has 63 user-callable functions, of which 9 (14%) are called by the Volant control code. We instrumented each of the system functions so that a system call sequence profile could be built in the same manner pH does. The Volant control code is much simpler than that of SandDragon but it still is a real-time preemptive multi-tasking system with five tasks. The version of pH we implemented on Volant maintained just one profile, for the control program which is the only program that can run. This is in contrast to pH on SandDragon, which maintains a profile for every program run including the robot control program.

Like the immune system on SandDragon, pH on Volant was thread-aware. Due to memory limitations, we could not include code in the monolithic image to save the profile to non-volatile storage on the robot. The profile had to be uploaded via serial link for storage and analysis on the OCU or other host. We did not implement profile download from the OCU to the Volant.

An operator control unit based on a Pocket PC was developed for Volant. The display on the Pocket PC serves as the control input, while status is displayed simultaneously. See Figure 9. The area marked by the cross is a virtual joystick for driving the robot with the Pocket PC stylus. The grayed status areas above that are placeholders for numeric status values. Only odometry and immunology status were implemented.

In order to monitor the immune system, we modified the original control program and communications protocol to include an immune system status byte that was sent from the Volant to the OCU. This status byte was generated internally by the immune system and was not used or interpreted in any way by the control program. The byte represented whether or not the profile had been changed and was displayed on the OCU status area so we could monitor it easily as we operated the robot.

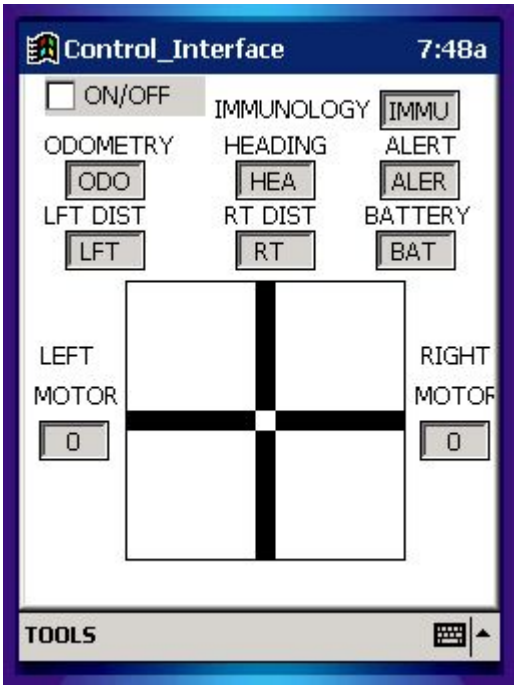


Figure 9. Volant Control Interface

Since profiles were not stored in flash on the vehicle, we uploaded the profile to a host computer for analysis. To upload the profile, the communication routines on the Volant monitored the communication stream for a special sequence of bytes. When this sequence was received, the robot dumped its profile, as text, to the communication stream for capture on the host computer. Appendix E contains more details about the Volant profile.

Because we could not store the profile on the robot, we began with an empty profile each time the robot was turned on. We used the heuristic implemented on the original pH which would

freeze the profile after a certain ratio of total system calls to calls since the last profile change had elapsed. We could monitor when this happened by observing the immunology status byte value change from 1 to 0. This would happen about a minute after start-up. We proceeded to operate the vehicle and monitored the immunology status byte for changes from 0 to 1, which would indicate one or more new system call pairs had been added to the profile. No new pairs were ever added to the profile. As with SandDragon, this may be because the control program exercised only the parts concerned with teleoperation, and as such did not execute many different control paths.

The immune system increased the control code size from 140971 bytes to 200766 bytes, or 42%. However, it added less than 5% CPU load to the system and did not cause any noticeable performance impact during driving.

5.0 Conclusions and Future Directions

This work extended the successful application of immune systems from the domain of computer security to the domain of real time control systems for robotic vehicles. We added a biologically-inspired computer immune system, based on Somayaji's pH, to the control system of the SandDragon mobile robot. No functional changes to the control code were required, and performance was not noticeably reduced. We then demonstrated a similar immune system on a smaller mobile robot and showed that the immune system performs well, although the size could be optimized considerably.

The immune system on SandDragon helped us to track down a coding bug and highlighted scheduling inefficiencies in the control application. With just the simple teleoperation control codes, the immune systems could not detect differences in operators or operating conditions. We hope to extend this research to explore how an immune system will perform with robot control code for autonomous operation. Another approach is to select a different interface to instrument, such as the data streams from environmental sensors.

Future work will consider the handling of anomalies. Anomaly information could be published by the immune system and would appear to the control system as an exception. The control system would have to be modified to include specific handlers for such exceptions, but this mechanism would work well in conjunction with traditional exception handling mechanisms.

As an additional layer in a multi-layered approach, the immune system is complementary to traditional error detection and error handling techniques. This can be thought of as biologically-inspired defense in depth. We demonstrated an immune system can be added with very little application developer effort, resulting in little to no performance impact. The particular data and interface must be selected carefully, but the methods described here are extensible to any system that processes a sequence of data through a software interface.

6.0 References

1. D. Dasgupta (Ed.), *Artificial Immune Systems and their Applications*. Springer-Verlag, Inc. Berlin, January 1999.
2. A. B. Somayaji, *Operating System Stability and Security through Process Homeostasis.* Dissertation, University of New Mexico, July 2002.
3. S. Forrest, S. A. Hofmeyr, a. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy,* IEEE Computer Society, 1996.
4. S. Forrest, S.A. Hofmeyr, and A. Somayaji. "Computer Immunology."*Communications of the ACM*, Vol. 40 Number 10, October 1997.
5. W. Lee et al. "Learning Patterns from Unix Process Execution Traces for Intrusion Detection." In *Proceedings of the AAAI97 workshop on AI methods in Fraud and Risk Management*, 1997.
6. R. A. Maxion and K. M. C. Tan. "Anomaly Detection in Embedded Systems." *IEEE Transactions on Computers*, Vol. 51 No. 2, February 2002.
7. R. Sekar et al. "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors." In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2001.
8. M. Stillerman et al. "Intrusion Detection for Distributed Applications." *Communications of the ACM*, Vol. 42 No. 7, July 1999.

Websites

http://www.carleton.edu/~soma

http://www.sandia.gov/isrc/sanddragon.html

Appendix A: pH and Computer Immunology

Background

In [A1], Forrest, et al introduced the problem of protecting computer systems as the problem of learning to distinguish self from non-self. They described a method for change detection using pattern matching ideas which was based on the generation of T-cells in the immune system. They demonstrated the feasibility of using this change-detection method for computer virus detection. Unlike many virus detection methods, this method did not require a priori information about the virus.

In [A2], Forrest, Hofmeyr, Somayaji, and Longstaff used these ideas for anomaly detection in which "normal" (i.e. 'self') was defined by short-range correlations in a process' (kernel) system calls. They showed this approach was compact with respect to sequences, distinguished between different kinds of processes, and could detect anomalies in the sequence of system calls.

The sequence method of system calls (without parameters) works surprisingly well when compared to other more sophisticated algorithms. Warrender [A3] compared the sequence method with several other algorithms including a Hidden Markov model generator and a rule inference algorithm (RIPPER). Her experiments showed that the sequence method was almost as accurate as the best algorithm while being much less computationally expensive.

The Algorithm

We will briefly summarize the concepts for anomaly intrusion detection. For a more complete description, see Somayaji's dissertation [A5]. Profiles of normal behavior are built for programs of interest, treating deviations from this profile as anomalies. First profiles or databases of normal behavior (analogous to the training stage of a learning system); secondly the databases are used to monitor process behavior for significant deviations from normal (analogous to the test phase).

The algorithm used to build the normal database is extremely simple. They scan traces of system calls generated by a particular program and build up a database of all unique sequences of a given length, *k*, that occurred during the trace.

They have defined normal in terms of short sequences of system calls. Parameters passed to the system calls are ignored. They only look at temporal ordering of the system calls. This temporal ordering will be referred to as a trace. Each program of interest has a different database, which is specific to a particular architecture, software version and configuration, and usage patterns. Once a stable database is constructed for a given program, the database can be used to monitor the ongoing behavior of the processes invoked by that program.

The construction of the normal database is illustrated with the following example. Suppose we have observed the following trace of system calls (excluding parameters):

open, read, mmap, mmap, open, read, mmap

We slide a window of size *k* across the trace, recording each unique sequence of length *k*. For example, if k=3, then we would get the unique sequences.

open, read, mmap

read, mmap, mmap

mmap, mmap, open

mmap, open, read

Once we have a database of normal behavior, we use the same method that we used to generate the database to check new traces of behavior. We examine all overlapping sequences of length *k* in the new trace and determine if they are represented in the normal database. Sequences that do not occur in the database are considered to be mismatches. By recording the number of mismatches, we can determine the strength of an anomalous signal.

If we have the following trace and the above database

open, read, mmap, mmap, open, *mmap*, mmap

we will have the following new sequences:

mmap, open, *mmap*

open, *mmap*, mmap

So this new trace contains two anomalous sequences.

Another way was devised to analyze system call traces, called look ahead pairs. In the sequences method, the literal contents of the fixed window in the profile and the set of these sequences constituted the model of normal program behavior. With the lookahead pairs method, the set of pairs of system calls is recorded, with each pair representing the current call and a preceding call. For a window of size *k*, *k* – 1 pairs are formed, one for each system call preceding the current one. The collection of unique pairs over the traces for a single program constitutes the model of normal behavior for the program.

For the trace,

open, read, mmap, mmap, open, read, mmap

we form the following pairs:

Position 2	***Position 1***	***current***
	open	read
open	read	mmap
read	mmap	mmap
mmap	mmap	open
mmap	open	read

Sequences

Position 2	*Position 1*	*current*
mmap	open	read
open or read	read or mmap	mmap
mmap	mmap	open

Look ahead pairs

The current system call forms a pair with a system call either in position 1 or 2. When forming the pair with position 2, we do not consider what system call is in postion1.

So in this example read is preceded by open in position1 and mmap in position 2.

mmap, *, read

*, open, read

mmap is preceded in position 1 by either read or mmap and in position 2, open or read.

open, *, mmap

read, * , mmap

*, read, mmap

*, mmap, mmap

Open is preceded by mmap in both positions 1 and 2.

mmap, *, open

*, mmap, open

Note that the sequences

read, read, mmap

open, mmap, mmap

are not sequences in the database but would not be detected as anomalous in the lookahead pair representation. The look aheadpairs method does not detect all anomalous sequences that would be detected by the sequence approach. The sequence method is more likely to produce a larger number of anomalies. However lookahead pairs can be updated and tested more quickly, have modest worst-case storage requirements, and can be easily implemented. Lookahead pairs have been sown to be sensitive enough in practice [A5].

Somayaji has implemented the lookahead pairs method as a patch to the Linux kernel (2.2, 2.4, 2.6). Detail can be found in [A5].

Appendix References:

[A1] 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.

[A2] S. Forrest, S. A. Hofmeyr, a. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy.* Los Alamitos, CA, 1996. IEEE Computer Society Press.

[A3] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Research in Security and Privacy, pages 133-145.* Los Alamitos, CA, 1999. IEEE Computer Society Press.

[A4] S. Hofmeyr, A. Somayaji, ans S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151-180, 1998.

[A5] Anil Somayaji. PhD dissertation, Operating System Stability and Security through Process Homeostasis. University of New Mexico, July 2002.

Appendix B: Robot Hardware Details

SandDragon as configured for this project

Dimensions:	43.5" L overall x 23" W x 20" H (110cm L x 58 cm W x 51 cm H), 80 lb (36 kg)
Propulsion:	Four electric motors achieving top speed of 4.5 mph (7.2 kph)
Processing:	PC104 stack containing a Real Time Devices 300 MHz CPU card with 32 MB RAM; network card; flash card carrier; Galil DMC-1250 motor controller. Rear body contains a Galil DMC-1400 motor controller, connected to PC104 stack by Ethernet.
Radios:	FreeWave radio modem (900 MHz). Southern California Microwave video transmitter (1.8 GHz)
Sensors:	none
OCU:	Windows 2000 pen computer with handheld USB game controller, external radio box containing radio modem, video receiver, and LCD display screen

More information can be found at http://www.sandia.gov/isrc/sanddragon.html

Volant

Dimensions:	14" L by 11.5" W by 7.75" H, 9 lb (4 kg)
Propulsion:	Four electric motors achieving top speed of 2.5 mph (4.0 kph)
Processing:	One 22.1 MHz Rabbit 2000 processor providing communications and high-level control. Two 4 MHz PIC PIC16F876 processors providing low-level motor and sensor control.
Radios:	FreeWave radio modem (900 MHz)
Sensors:	Four short-range infrared distance sensors. Motor odometry. Long range distance sensor and compass to be added.
OCU:	Pocket PC iPAQ Model 3765 with external radio; or dedicated integrated controller containing joystick and radio.

Appendix C: Robot Software Details

SandDragon Robot

Linux kernels (2.4.20, 2.6.5)
http://www.kernel.org

High-resolution timer patch for 2.4.20 kernel
hrtimers-2_4_20-3_0.patch
http://www.sourceforge.net

pH patches for Linux (0.22, 0.26)
http://www.scs.carleton.ca/~soma/pH/dist/

SandDragon OCU

Windows 2000
http://www.microsoft.com

ActivePerl 5.8.0 (Build 806)
http://www.activestate.com

Volant Robot

Dynamic C compiler v8.10
http://www.zworld.com

MicroC/OS-II v2.51 for Rabbit microprocessor
UCOS2 v1.02 Dynamic C Module
http://www.zworld.com
http://www.ucos-ii.com

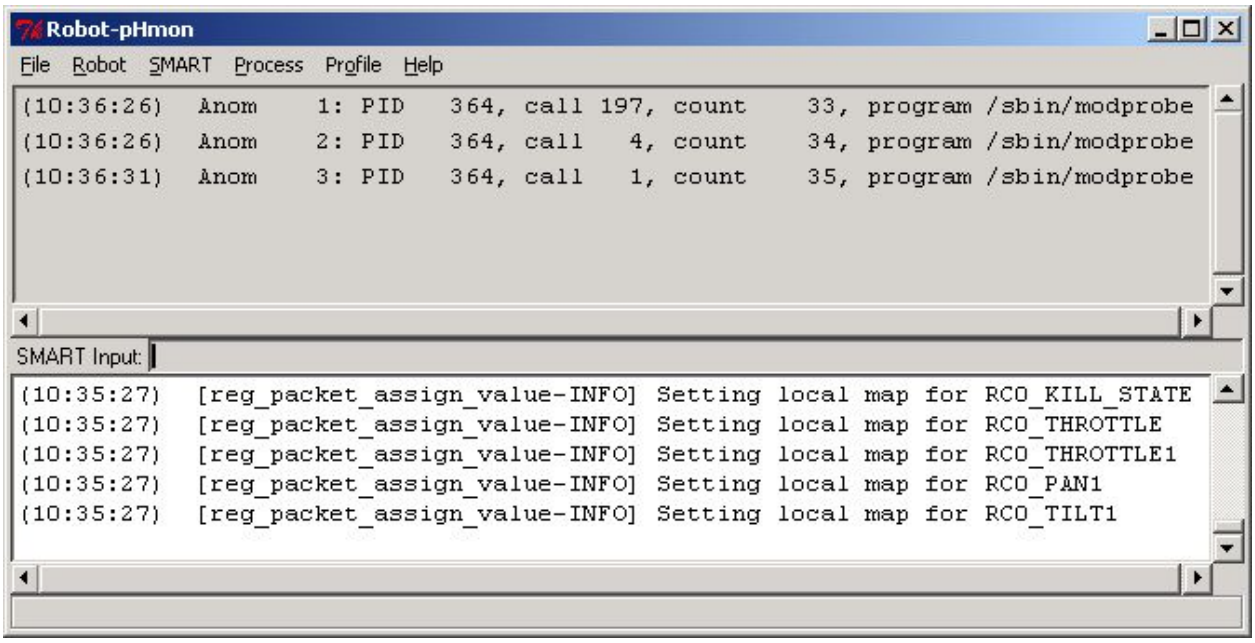
Volant OCU

Windows CE 3.0
http://www.microsoft.com

Microsoft eMbedded Visual Tools version 3.0
Microsoft Windows Platform SDK for Pocket PC 2002
http://www.microsoft.com/windows/embedded/ce

Appendix D: robot-pHmon

Robot-pHmon is the graphical interface which runs on the SandDragon OCU. We use it to run the control program remotely and monitor the immune system on the robot. It provides other utility functions as well. Refer to the diagram below. The interface consists of a menu bar, a top pane, a SMART Input edit box, a bottom pane, and a status bar. Below the figure is a description of the menu selections. Menu items ending with an ellipsis indicate these items will bring up a dialog box for additional input. The top pane lists anomalies that have occurred since boot or the last time anomalies were cleared from the log. The SMART Input edit box allows the operator to enter a command for the robot control program, which contains a shell command processor. The bottom pane contains output from the robot control program. The output is what the operator would normally see on the robot console via an external monitor attached to the robot. The status bar at the bottom is currently unused.



File menu

Save Output As...	Saves contents of top and bottom panes to a file
Clear Output	Clears the top and bottom panes
Insert Comment...	User can insert timestamped comment in output log
Quit	Quits robot-pHmon

Robot menu

Reboot Robot...	Reboots the robot remotely
Shutdown Robot...	Shuts down the robot remotely

SMART menu

Change SMART Binary...	The full path to robot control program, or any executable, run when "Run SMART" is selected
Run SMART	Runs the specified executable on the robot
Quit SMART	Issues the "quit" command to the robot executable
Kill top SMART process	Sends sigterm to each process belonging to the executable specified in "Change SMART Binary"
Normalize SMART	Normalizes the specified running process
Tolerize SMART	Tolerizes the specified running process specified
Reset SMART Profile	Resets the profile for the specified running process

Process menu

Tolerize	Tolerizes all processes with anomalies listed
Kill -TERM	Not implemented
Kill -KILL	Not implemented
Process List	Lists all processes on robot with pH information
Dump Process List to Output	Lists robot processes in output pane

Profile menu

Flush Robot Profiles to Disk	Writes all profiles to disk on robot
Save SMART Profile...	Copies profile from robot and saves on OCU
Restore SMART Profile...	Copies profile from OCU to robot

Help menu

About	Displays robot-pHmon version information

Appendix E: Volant Profile

The following is a listing of a Volant profile, captured after the robot was driven for a short time. Each bracketed pair of numbers represents a pair of system calls that have occurred together within a window of 9 system calls, with the leftmost occurring before the right. The numbers within the brackets are the unique system call identifiers we have assigned for each MicroC/OS-II function. The number following the equals sign is an 8-bit binary number representing how far apart the calls occurred within the window. The least significant (rightmost) bit is set if the first call ever occurred immediately prior to the second call, while the next bit is set if the first call ever occurred 2 calls prior to the second call, and so on. For example, [8][37] = 1001 0010 means system call #8 was seen preceding system call #37 at least three times -- once 2 calls prior, once 5 calls prior, and once 8 calls prior.

[8][37] = 1001 0010
[8][38] = 0010 0100
[8][48] = 0100 1001
[37][8] = 0000 0010
[37][37] = 0110 1100
[37][38] = 1101 1001
[37][40] = 1111 1110
[37][45] = 0100 0000
[37][48] = 1011 0110
[38][8] = 0000 0001
[38][37] = 1011 0110
[38][38] = 0110 1100
[38][40] = 1111 1111
[38][45] = 0010 0000
[38][48] = 1101 1011
[40][40] = 0000 1111
[40][45] = 0001 1111
[42][42] = 1111 1111
[43][43] = 1010 0000
[43][56] = 1010 1010
[43][59] = 0101 0101
[48][8] = 0000 0100
[48][37] = 1101 1001
[48][38] = 1011 0010
[48][40] = 1111 1100
[48][45] = 1000 0000
[48][48] = 0110 1100
[56][43] = 1010 1010
[56][56] = 1111 1111
[56][59] = 0101 0101
[59][43] = 0101 0101
[59][56] = 1111 1111
[59][59] = 1111 1111

The following list shows our arbitrary assignment of system call numbers to MicroC/OS-II functions. The numbers are not continuous from 0 through 62. This is because after the original assignment, we deemed certain functions to be time-critical and therefore inappropriate to track with the immune system, e.g., OSTimeTick.

2	OSFlagAccept	33	OSQPost
3	OSFlagCreate	34	OSQPostFront
4	OSFlagDel	35	OSQPostOpt
5	OSFlagPend	36	OSQQuery
6	OSFlagPost	37	OSSchedLock
7	OSFlagQuery	38	OSSchedUnlock
8	OSInit	39	OSSemAccept
11	OSMboxAccept	40	OSSemCreate
12	OSMboxCreate	41	OSSemDel
13	OSMboxDel	42	OSSemPend
14	OSMboxPend	43	OSSemPost
15	OSMboxPost	44	OSSemQuery
16	OSMboxPostOpt	45	OSStart
17	OSMboxQuery	46	OSStatInit
18	OSMemCreate	47	OSTaskChangePrio
19	OSMemGet	48	OSTaskCreate
20	OSMemPut	49	OSTaskCreateExt
21	OSMemQuery	50	OSTaskDel
22	OSMutexAccept	51	OSTaskDelReq
23	OSMutexCreate	52	OSTaskQuery
24	OSMutexDel	53	OSTaskResume
25	OSMutexPend	54	OSTaskStkChk
26	OSMutexPost	55	OSTaskSuspend
27	OSMutexQuery	56	OSTimeDly
28	OSQAccept	57	OSTimeDlyHMSM
29	OSQCreate	58	OSTimeDlyResume
30	OSQDel	59	OSTimeGet
31	OSQFlush	60	OSTimeSet
32	OSQPend	62	OSVersion

Distribution:

3 Anil Somayaji
Carleton University
5302 Herzberg Building
1125 Colonel By Drive
Ottawa, ON K1R 5B2 Canada

1 Stephanie Forrest
Department of Computer Science
MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131-0001

1	MS	1005	R. D. Skocypec
1		1007	P. D. Heermann
1		1007	S. C. Roehrig
1		1010	K. M. Hays
10		1125	W. A. Amai
1		1125	P. C. Bennett
1		1176	R. M. Cranwell
1		1188	M. R. Glickman
1		1188	J. S. Wagner
10		1207	E. A. Walther
1		1207	J. R. Yoder
1		1209	J. M. Chavez
1		9018	Central Technical Files, 8945-1
2		0899	Technical Library, 9616