*Book Title*
*Book Editors*
*IOS Press, 2003*

# The Case for Self-Healing Software

Angelos D. Keromytis [1]
*Columbia University*

**Abstract.** Existing approaches to software security and reliability have proven inadequate in offering a good tradeoff between the assurance, reliability, availability, and performance. We argue that **reactive** protection mechanisms need to be added to our panoply of defenses. Furthermore, we argue that such mechanisms need to be much more invasive than previously envisioned. We discuss our approach to such mechanisms by introducing the concept of **self-healing software**. We describe the principles behind self-healing software systems and showcase the concepts by giving an overview the Worm Vaccine architecture.

**Keywords.** Self-healing Software, Reactive Systems, OODA Feedback Loop

## 1. Motivation

Despite considerable work in fault tolerance and reliability, software remains notoriously buggy and crash-prone. The current approach to ensuring the security and availability of software consists of a mix of different techniques:

- **Proactive techniques** seek to make the code as dependable as possible, through a combination of safe languages (*e.g.,* Java [8]), libraries [1] and compilers [9,14], code analysis tools and formal methods [3,7,25], and development methodologies.
- **Debugging techniques** aim to make post-fault analysis and recovery as easy as possible for the programmer that is responsible for producing a fix.
- **Runtime protection techniques** try to detect the fault using some type of fault isolation such as StackGuard [6] and FormatGuard [4], which address specific types of faults or security vulnerabilities.
- **Containment techniques** seek to minimize the scope of a successful exploit by isolating the process from the rest of the system, *e.g.,* through use of virtual machine monitors such as VMWare or Xen, system call sandboxes such as Systrace [16], or operating system constructs such as Unix *chroot()*, FreeBSD's *jail* facility, and others [24,13].
- **Byzantine fault-tolerance and quorum techniques** rely on redundancy and diversity to create reliable systems out of unreliable components [26,17].

These approaches offer a poor tradeoff between assurance, reliability in the face of faults, and performance impact of protection mechanisms. We believe that a new class

[1]Correspondence to: Angelos D. Keromytis, Department of Computer Science, Columbia University, M.C. 0401, 1214 Amsterdam Avenue, New York, New York, 10027, USA; Tel.: +1 212 939 7095; Fax: +1 212 666 0140; E-mail: angelos@cs.columbia.edu

of **reactive** protection mechanisms need to be added to the above list. Some techniques that can be classified as reactive include Intrusion Prevention Systems (IPS) and automatically generated content-signature blockers, *e.g.,* [15]. Most such systems have focused on network-based prevention, augmenting the functionality of firewalls. However, a number of trends make the use of such packet inspection technologies unlikely to work well in the future:

- Due to the increasing line speeds and the more computation-intensive protocols that a firewall must support (such as IPsec), firewalls tend to become congestion points. This gap between processing and networking speeds is likely to increase, at least for the foreseeable future; while computers (and hence firewalls) are getting faster, the combination of more complex protocols and the tremendous increase in the amount of data that must be passed through the firewall has been and likely will continue to outpace Moore's Law [5].
- The complexity of existing and future protocols makes packet inspection an expensive proposition, especially in the context of increasing line speeds. Furthermore, a number of protocols are inherently difficult to process in the network because of lack of knowledge that is readily available at the endpoints (*etc.* FTP and RealAudio port numbers).
- End-to-end encryption, especially of the opportunistic type[1] effectively prevents inspection-based systems from looking inside packets, or even at packet headers.
- Finally, we believe that it is only a matter of time until exploits such as worms start using polymorphism or metamorphism [23] as cloaking techniques. The effect of these is to increase the analysis requirements, in terms of processing cycles, beyond the budget available to routers or firewalls.

All these factors argue for host-based reactive protection mechanisms. In the space of such mechanisms, we focus on **self-healing software** as a reactive protection technique. In the next section we outline our view of self-healing software systems and give our first thoughts on the structure of such systems. In Section 3 we give a brief overview of an instance of a self-healing software system, the Worm Vaccine architecture. As the name implies, this is a system designed to protect against network worms that spread through software-based vulnerabilities such as buffer overflows. Although this class of vulnerabilities has been studied extensively, we discuss our architecture as a concrete example of a self-healing software system.

## 2. Principles of Self-Healing Software

Our approach to self-healing software, shown in Figure 1, is modeled after the concept of an Observe Orient Decide Act (OODA) feedback loop. Our high-level intuition is that, if proactive or runtime protection mechanisms are too expensive to use in a blanket manner, we should instead use them in a targeted manner. Identifying where and how to apply protection is done by observing the behavior of the system in a non-invasive (or minimally invasive) manner. The goal of this monitoring is to detect the occurrence of a

[1]By "opportunistic" we mean that client-side, and often server-side, authentication is often not strictly required, as is the case with the majority of web servers or with SMTP over TLS (*e.g.,* sendmail's STARTSSL option).

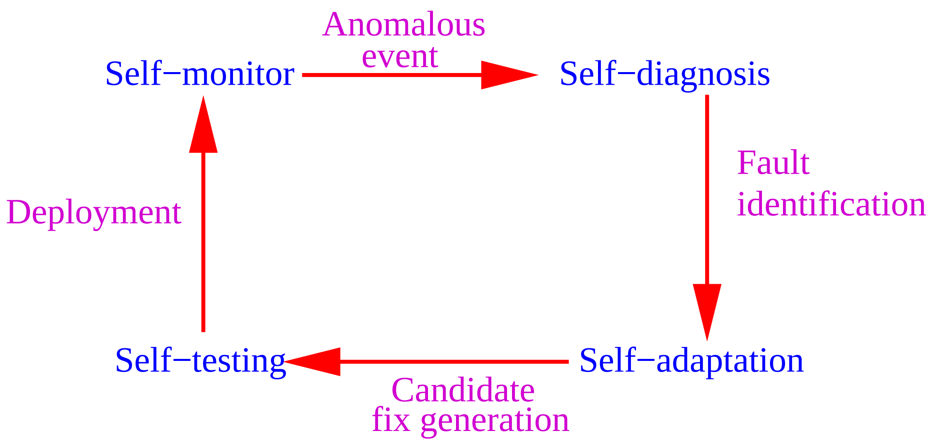

**Figure 1.** General architecture of a self-healing system. The system monitors itself for indications of anomalous behavior. When such is detected, the system enters a self-diagnosis mode that aims to identify the fault and extract as much information as possible with respect to its cause, symptoms, and impact on the system. Once these are identified, the system tries to adapt itself by generating candidate fixes, which are tested to find the best target state.

fault and determine its parameters, *e.g.*, the type of fault, the input or sequence of events that led to the it, the approximate region of code where the fault manifests itself, and any other information that may be useful in creating fixes.

Following identification, the system will need to create one or more possible fixes tailored to the particular instance of the fault. The nature of these fixes depends on types of faults and the available protection mechanisms. Potential fixes to software faults include snapshot-rollback, input filtering, increased monitoring or isolation for the vulnerable process, selective application of any runtime protection mechanism, and others.

Each candidate fix produced by the system is then tested, ideally in an isolated environment, to verify its efficacy and impact on the application (*e.g.*, in terms of side effects or performance degradation). This testing can take several forms, including (but not limited to) running pre-defined test-suites, replaying previously seen traffic (including the input that triggered the fault), *etc.* The various fixes are rank-ordered based on the results of the testing phase, as well as other information (*e.g.*, how many lines of code are modified, what the down-time for deploying the fix will be, how many components need to be reconfigured, *etc.*). If an acceptable fix is produced, the system is updated accordingly. This can be done through established patch-management and configuration-management mechanisms, or any other suitable mechanism.

Our particular approach to self-healing software, shown in Figure 2, is to apply structural transformations to the application itself, aimed at eliminating the root cause of the vulnerability. Our system uses a set of software probes that monitor the application for unknown instances of specific types of faults (*e.g.*, application crash, buffer overflow exploit, *etc.*). Upon detection of a fault, we invoke a localized recovery mechanism that seeks to recognize and prevent the specific failure in future executions of the program. Using continuous hypothesis testing, we verify whether the fault has been repaired by re-running the application against the event sequence that apparently caused the failure. Our initial focus is on automatic healing of services against newly detected low-level software faults. We emphasize that we seek to address a wide variety of software failures, not just attacks.

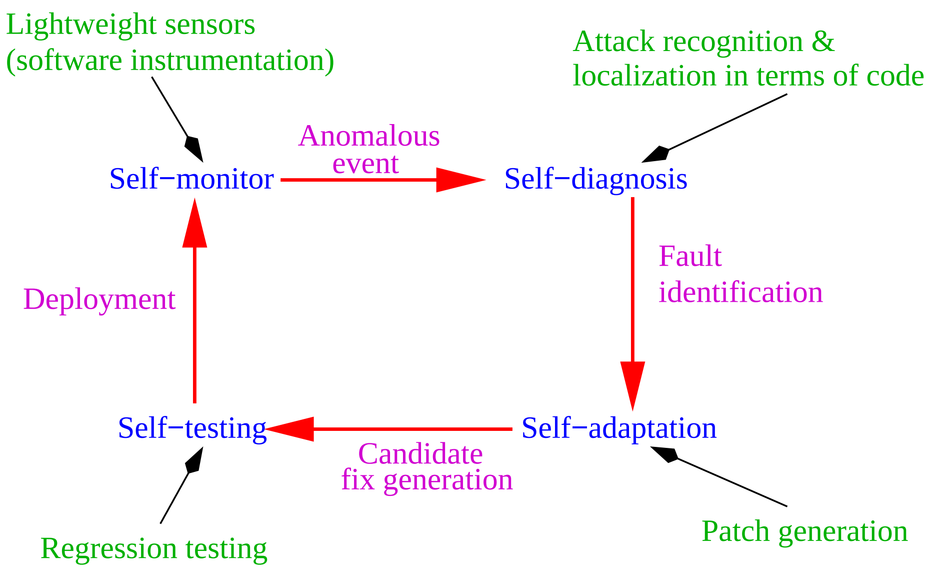

**Figure 2.** Our approach to self-healing software systems. We instrument the production system with lightweight sensors that do not affect its performance or functionality. If an attack or other anomalous event if flagged, we use an instrumented version of the system to diagnose the event. Once the fault is localized in terms of code region, we produce a set of candidate patches that are appropriate for that specific class of fault, or are generic band-aids. The resulting system images are evaluated against a test-suite as well as the malicious input. A human may be required to approve deployment, or the system may perform all the steps automatically.

In our overall approach, we treat faults as exceptions. In determining how to recover from such exceptions, we introduce the hypothesis of an **execution transaction.** Very simply, we posit that for the majority of code, we can treat each function execution as a transaction (in a manner similar to a sequence of operations in a database) that can be aborted without adversely affecting the graceful termination of the computation. Each function call from inside that function can itself be treated as a transaction, whose success or failure does not contribute to the success or failure of its enclosing transaction. Under this hypothesis, it is sufficient to snapshot the state of the program execution when a new transaction begins, detect a failure per our previous discussion, and recover by aborting this transaction and continuing the execution of its enclosing transaction. Note that our hypothesis does not imply anything about the correctness of the resulting computation, when a failure occurs. Rather, it merely states that if a function is prevented from (for example) overflowing a buffer, it is sufficient to continue execution at its enclosing function, “pretending” the aborted function returned an error. We call this approach *error virtualization*. Depending on the return type of the function, a set of heuristics are employed to determine an appropriate error return value that is, in turn, used by the program to handle error condition. For more details, including our preliminary experimental validation of the concept of execution transactions, see [18].

However, saving the application’s state prior to each function call is likely to be an expensive proposition. Instead, we introduce such transactional processing as directed by the fault-identification phase, *i.e.*, only for the function(s) where the fault exhibited itself. One way to view our system is that we speculatively execute code that we have previously determined to be susceptible to faults. This approach, which we call *micro-speculation*, can be implemented through source-level code transformations [18], a selective emulator [20], or by modifying the compiler [22].

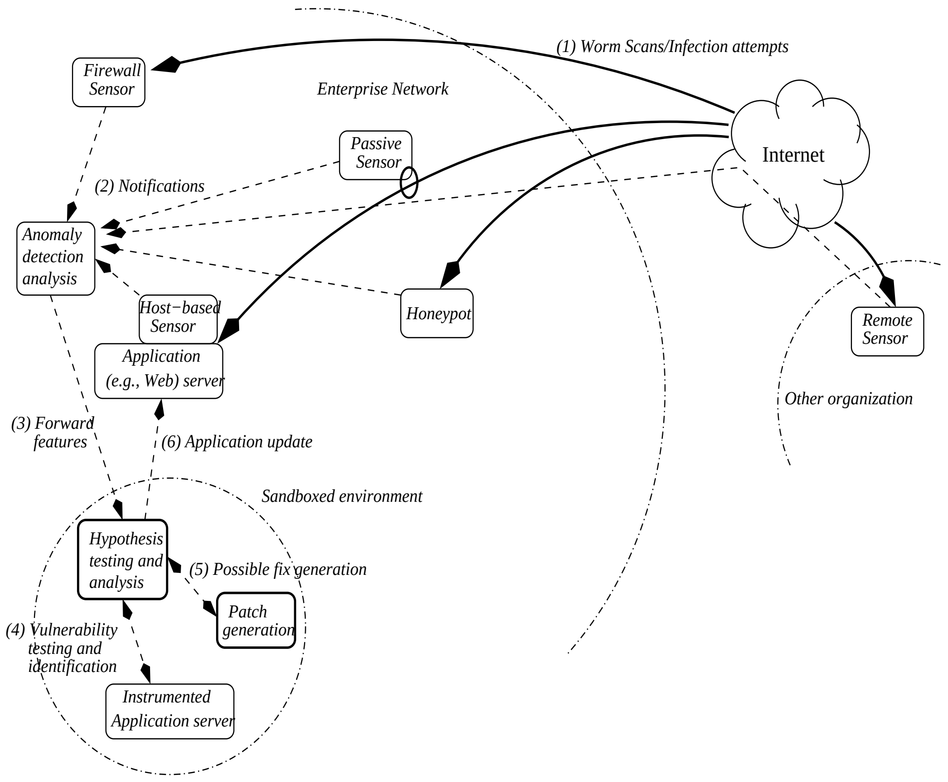
(1) Worm Scans/Infection attempts
Firewall Sensor
Enterprise Network
Internet
Passive Sensor
(2) Notifications
Anomaly detection analysis
Host−based Sensor
Application (e.g., Web) server
Honeypot
Remote Sensor
Other organization
(3) Forward features
(6) Application update
Sandboxed environment
Hypothesis testing and analysis
(5) Possible fix generation
Patch generation
(4) Vulnerability testing and identification
Instrumented Application server

**Figure 3.** Worm vaccination architecture: sensors deployed at various locations in the network detect a potential worm (1), notify an analysis engine (2) which forwards the infection vector and relevant information to a protected environment (3). The potential infection vector is tested against an appropriately-instrumented version of the targeted application, identifying the vulnerability (4). Several software patches are generated and tested using several different heuristics (5). If one of them is not susceptible to the infection and does not impact functionality, the main application server is updated (6).

## 3. A Case Study: Worm Vaccine Architecture

We now describe one instance of a self-healing software architecture geared specifically against worms spreading via buffer overflow vulnerabilities. The architecture is shown graphically in Figure 3, and is described in more detail in [21]. Our system assumes access to source code, and automatically generates source-level patches that can be applied against the application's source code to address newly discovered vulnerabilities. Our approach creates an OODA loop using a properly instrumented version of the application (called the *Oracle*). The instrumentation is specific to the types of faults that we are interested in addressing, *e.g.,* buffer overflow vulnerabilities. We use the Oracle to verify hypotheses that a particular connection or packet is malicious, *i.e.,* that it will cause a fault when processed by the application. Such hypotheses can be generated in a number of different ways (all of which may be used simultaneously):

- Implicitly, any traffic that reaches the Oracle from a source external to the organization is considered suspicious. In this case, the Oracle operates in a honeypot-like mode.
- Explicitly, if an anomaly detector that inspects network traffic generates a suspicion that a particular connection or packet is anomalous. Such traffic may be forwarded to the Oracle for further inspection.
- Reactively, as a result of an observed fault on the production version of the application by a host-resident sensor or anomaly detector. Traffic that *may* have contributed to the fault is replayed to the Oracle, under the assumption that the fault manifests itself shortly after the attack.

Alternatively, we can construct a distributed Oracle (which we have named an Application Community [12,11]) by spreading the task of monitoring for failures across multiple independent instances of the software. For example, each instance of a web server in a server farm may only be monitoring one code function; or each instance may be randomly selecting requests during the handling of which full monitoring will be enabled. These nodes then exchange alerts or fixes, or notify a central site.

In addition to confirming a causal relationship between a particular connection/packet and a fault, the Oracle localizes it in terms of the functions and buffers involved. There are several ways to achieve this, and many of the existing buffer-overflow protection mechanisms can be used to that end. We developed our wrapping technique, DYBOC [18], which performs three source code transformations:

1. DYBOC instruments all function entry and exit points to record which functions are active at any time. Thus, we can obtain an accurate snapshot of the call graph at the time a fault occurs.
2. DYBOC transforms all buffers that were allocated on the stack such that they are allocated on the heap. Furthermore, the allocation routine allocates two extra memory pages that surround the desired memory region. These extra memory pages are marked as read only, so any write operations (as may be the case when a buffer overflow occurs) generate a fault that is delivered to the process in the form of a signal (SIGSEGV). DYBOC also records the name of the buffer and the function in which it was allocated.
3. DYBOC inserts a signal handler that prints the call stack and the name of the buffer on which the overflow (or underflow) occurred.

These transformations can lead to a significant performance slowdown for the Oracle, relative to the unmodified application. Since the Oracle is not used to serve actual requests (unless so desired), this slowdown only affects our ability to test hypotheses.

Once the Oracle confirms and localizes the failure, the system attempts to generate a series of source-level patches that eliminate the vulnerability. These fixes have the same form as the instrumentation we described previously, but are tailored to the specifics of the vulnerability. Specifically:

- We transform the buffer that is overflowed, as above.
- We snapshot the program state at the point where the function during the execution of which the overflow actually occurs is called. We use *sigsetjmp()* as a lightweight mechanism for achieving this, although more comprehensive and computationally expensive mechanisms may be used if necessary [20].
- We insert a signal handler that will catch the signal generated by a buffer overflow on the instrumented buffer, and recover program execution. The program will continue executing from the location immediately after the call to the vulnerable function.

If no overflow occurs, the program will continue behaving as before. However, if an overflow occurs, our patch will cause execution of the vulnerable function to terminate and control to be transferred to its caller. The hypothesis behind this approach is that in most cases the caller will gracefully handle this condition. Since we cannot depend on this to always be true, the system includes a testing phase during which it examines the impact of the generated patch. This testing phase involves instantiating a patched version of the application and then testing its behavior against $(a)$ a pre-defined set of test inputs (*e.g.*, a regression test suite previously created by the administrator), $(b)$ the specific input the caused the fault, as well as any other inputs that caused faults in the past, and $(c)$ previously seen traffic to the production system (assuming it is safe to do so). The goal of the testing phase is to identify whether the patch has eliminated the vulnerability and whether any instability has been introduced. As a proactive measure, we can use DYBOC at system creation time to perform exhaustive fault-injection and analysis to determine *a priori* whether any side effects manifest under specific transformations. If we are satisfied, we update the production server image and restart the process. If we failed to produce a good fix, we move up the call stack and restore execution to the caller of the caller of the vulnerable function. The same testing phase occurs, *etc.* Naturally, it is possible that no fix is possible, at which point other mechanisms, *e.g.*, code randomization [10,2], will have to be used.

All the steps described occur automatically. Thus, it is possible to construct a fully automated self-healing system. Alternatively, we can allow human intervention and inspection once a patch has been generated and tested by displaying a message on the administration console. The administrator can then inspect the incident details, view the generated fix and the results of the testing phase, and even interact with the patched application (if possible), before deciding whether to roll out the patch.

Since these steps (especially the recompilation and testing of the application) can take a non-trivial amount of time to complete, we can engage additional protection mechanisms while we are attempting to create a suitable patch. For example, we can restrict or prohibit access to the production server by installing a port-blocking or signature-blocking rule on the firewall, or we can turn on complete logging of all the actions of the production server, so that we can later on audit and recover from a compromise.

For a detailed description of the system and its evaluation, see [21,20,18].

## 4. Conclusions

We have made a case for the desirability and feasibility of self-healing software systems, as a class of reactive software-protection mechanisms. Such systems represent a new approach to system security and reliability. We discussed our early thoughts on the principles behind such systems and described the Worm Vaccine architecture, the first instance of a self-healing software system. Although work in this area is in its early stages, our results so far have been very encouraging [18,20,21,11,12,19]. We hope that this paper will motivate additional research in this promising new area.

## References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* $12^{th}$ *USENIX Security Symposium*, pages 105–120, August 2003.

[3] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the* $9^{th}$ *ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[4] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[5] M. Dahlin. *Serverless Network File Systems*. PhD thesis, UC Berkeley, December 1995.

[6] C. Cowan et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the* $7^{th}$ *USENIX Security Symposium*, January 1998.

[7] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the* $10^{th}$ *ACM Conference on Computer and Communications Security (CCS)*, pages 345–364, October 2003.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.

[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.

[10] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* $10^{th}$ *ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[11] M. Locasto, S. Sidiroglou, and A. D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2006.

[12] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* $1^{st}$ *Workshop on Hot Topics in Systme Dependability (HotDep)*, pages 288–292, June 2005.

[13] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 29–40, June 2001.

[14] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages (PoPL)*, January 2002.

[15] J. Newsome and D. Dong. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* $12^{th}$ *Annual Symposium on Network and Distributed System Security (SNDSS)*, February 2005.

[16] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the* $12^{th}$ *USENIX Security Symposium*, pages 257–272, August 2003.

[17] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the* $36^{th}$ *Hawaii International Conference on System Sciences (HICSS)*, January 2003.

[18] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the* $8^{th}$ *Information Security Conference (ISC)*, pages 1–15, September 2005.

[19] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[20] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[21] Stelios Sidiroglou and A. D. Keromytis. Countering Network Worms Through Automatic Patch Generation. *IEEE Security & Privacy Magazine*, 3(6):52–60, November/December 2005.

[22] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* $12^{th}$ *ISOC Symposium on Network and Distributed System Security (SNDSS)*, February 2005.

[23] P. Ször and P. Ferrie. Hunting for Metamorphic. Technical report, Symantec Corporation, June 2003.

[24] R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 15–28, June 2001.

[25] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the* $10^{th}$ *ACM Conference on Computer and Communications Security (CCS)*, pages 321–334, October 2003.

[26] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.