Characterizing Self-Healing Software Systems

Angelos D. Keromytis

Department of Computer Science, Columbia University
1214 Amsterdam Avenue, New York, NY 10027, USA
angelos@cs.columbia.edu

Abstract. The introduction of self-healing capabilities to software systems could offer a way to alter the current, unfavorable imbalance in the software security arms race. Consequently, self-healing software systems have emerged as a research area of particular interest in recent years. Motivated by the inability of traditional techniques to guarantee software integrity and availability, especially against motivated human adversaries, self-healing approaches are meant to complement existing approaches to security.
In this paper, we provide a first attempt to characterize self-healing software systems by surveying some of the existing work in the field. We focus on systems that effect structural changes to the software under protection, as opposed to block-level system reconfiguration. Our goal is to begin mapping the space of software self-healing capabilities. We believe this to be a necessary first step in exploring the boundaries of the research space and understanding the possibilities that such systems enable, as well as determining the risks and limitations inherent in automatic-reaction schemes.

Key words: Self-healing, reliability, availability, software security

1 Introduction

Self-healing capabilities have begun to emerge as an exciting and potentially valuable weapon in the software security arms race. Although much of the work to date has remained confined to the realm of academic publications and prototype-building, we believe that it is only a matter of time before deployed systems begin to incorporate elements of automated reaction and self healing. Consequently, we believe it is important to understand what self-healing systems are, why they evolved in the first place, and how they operate. Given the infancy of the area as a research focus, we only have a relatively small and, given the potential application scope of self-healing techniques, highly diverse sample set to draw upon in defining this space. Despite this, we believe that some high-level common characteristics of self-healing software systems can be discerned from the existing work.

What, exactly, is a self-healing system? For the purposes of our discussion, a self-healing software system is a software architecture that enables the continuous and automatic monitoring, diagnosis, and remediation of software faults. Generally, such an architecture is composed of two high-level elements: the software service whose integrity and availability we are interested in improving, and the elements of the system that perform the monitoring, diagnosis and healing. The self-healing components can

be viewed as a form of middleware — although, in some systems, such a separation is difficult to delineate.

Self-healing systems evolved primarily as a result of the failure of other techniques, whether in isolation or combination, to provide an adequate solution to the problem of software reliability. More specifically, self-healing techniques try to strike a balance between reliability, assurance, and performance (with performance generally in an inverse relationship to the first two). An important difference between self-healing and the traditional fault-tolerant architectures and techniques is that the former try to identify and eliminate (or at least mitigate) the root cause of the fault, while the latter generally only bring the system to a state from which it can resume execution. Thus, fault-tolerant systems can be viewed as primarily geared against rarely occurring failures.

The diversity of techniques for effecting self-healing reflects both the relative immaturity of the field and the large scope of failure and threat models such systems must cope with. Most work to date has focused on a narrow class of faults and/or the refinement of a specific detection or mitigation technique. Although comprehensive frameworks have been proposed [1, 2], none has been fully implemented and demonstrated to date. Although we expect this picture to remain the same for the foreseeable future, more comprehensive techniques and systems will emerge as we achieve a better understanding of the capabilities and limitations of existing proposed approaches.

Finally, we wish to distinguish between block-level system reconfiguration-based healing, and lower-level structural modification-based healing techniques. The former treat software as a black box with some configurable parameters, and focus on rearranging the way the system components interact with each other. The latter depend on specific, "low-level" techniques to detect, mitigate and mask/correct defects in the software. As a starting point, we focus on the latter approach, both because of our familiarity with this space and because of the use of such structural-modification techniques as the building elements for system-reconfiguration approaches. To the extent that software becomes componentized, we believe that these two high-level approaches are likely to converge in terms of tools and techniques used.

In the remainder of this paper, we will expand on the main three questions we posed at the beginning of this section: what are self-healing systems, why have they emerged, and how they operate, using specific examples from the literature.

2 Self-healing Systems: What

At a high level, self-healing software techniques are modeled after the concept of an Observe Orient Decide Act (OODA) feedback loop, as shown in Figure 1.

The high-level intuition is that, if proactive or runtime protection mechanisms are too expensive to use in a blanket manner, they should instead be used in a targeted manner. Identifying where and how to apply protection is done by observing the behavior of the system in a non-invasive manner. The goal of this monitoring is to detect the occurrence of a 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.

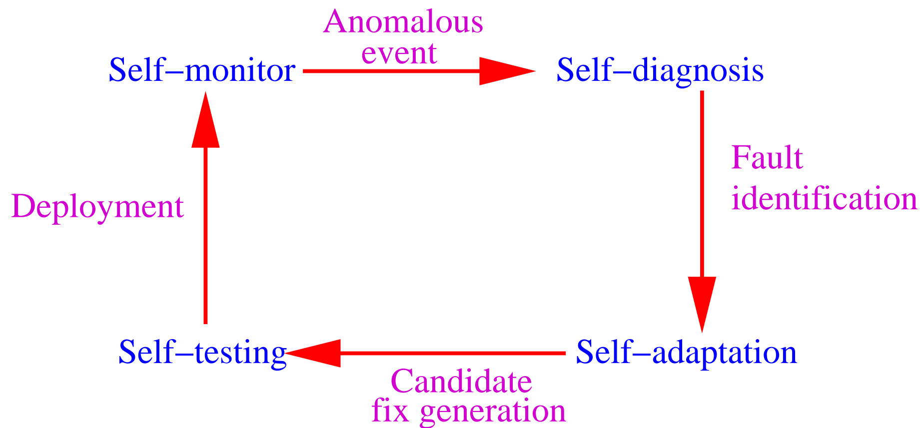

Fig. 1. General architecture of a self-healing system. The system monitors itself for indications of anomalous behavior. When such behavior is detected, the system enters a self-diagnosis mode that aims to identify the fault and extract as much information as possible with respect to its cause, symptoms, 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.

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 may then be tested 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 running pre-defined test-suites, replaying previously seen traffic (including the input that triggered the fault), *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.

Note that different fixes, or fixes of different accuracy/performance levels, may be successively applied as the system spends more time analyzing a fault. For example, the initial reaction to a failure may be a firewall reconfiguration. After further analysis, the system may produce a software patch or a content filter that blocks the specific input that caused the fault. Finally, the system may then replace the specific content filter with a generalized signature, obtained through signature generalization [3], dynamic analysis of the targeted software [4], or other means.

3 Self-healing Systems: Why

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 [5]), libraries [6] and compilers [7, 8], code analysis tools and formal methods [9–11], 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 [12] and FormatGuard [13], 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 [14], or operating system constructs such as Unix *chroot()*, FreeBSD's *jail* facility, and others [15, 16].
- **Byzantine fault-tolerance and quorum techniques** rely on redundancy and diversity to create reliable systems out of unreliable components [1, 17, 18].

These approaches offer a poor tradeoff between assurance, reliability in the face of faults, and performance impact of protection mechanisms. In particular, software availability has emerged as a concern of equal importance as integrity.

The need for techniques that address the issue of recovering execution in the presence of faults is reflected by recent emergence of a few novel research ideas [19, 20]. For example, *error virtualization* [19, 21] operates under the assumption that there exists a mapping between the set of errors that *could* occur during a program's execution (*e.g.,* a caught buffer overflow attack, or an illegal memory reference exception) and the limited set of errors that are explicitly handled by the program's code. Thus, a failure that would cause the program to crash is translated into a "return with an error code" from the function in which the fault occurred (or from one of its ancestors in the stack). These techniques, despite their novelty in dealing with this pressing issue, have met much controversy, primarily due to the lack of guarantees, in terms of altering program semantics, that can be provided. Masking the occurrence of faults will always carry this stigma since it forces programs down unexpected execution paths. However, we believe that the basic premise of masking failures to permit continued program execution is promising.

In general, we believe that a new class 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.,* [22]. 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 out-pace Moore's Law [23].
– 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 on-demand, opportunistic type effectively prevents inspection-based systems from looking inside packets, or even at packet headers.
– Finally, worms and other malware have started using polymorphism or metamorphism [24] 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.

4 Self-healing Systems: How

Most defense mechanisms usually respond to an attack by terminating the attacked process. Even though it is considered "safe", this approach is unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state.

Self-healing mechanisms complement approaches that stop attacks from succeeding by preventing the injection of code, transfer of control to injected code, or misuse of existing code. Approaches to automatically defending software systems have typically focused on ways to proactively or at runtime protect an application from attack. Examples of these proactive approaches include writing the system in a "safe" language, linking the system with "safe" libraries [6], transforming the program with artificial diversity, or compiling the program with stack integrity checking [12]. Some defense systems also externalize their response by generating either vulnerability [4, 25, 26] or exploit [3, 27–30] signatures to prevent malicious input from reaching the protected system.

Starting with the technique of *program shepherding* [31], the idea of enforcing the integrity of control flow has been increasingly researched. Program shepherding validates branch instructions to prevent transfer of control to injected code and to make sure that calls into native libraries originate from valid sources. Control flow is often corrupted because input is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call. Recent work focuses on ways to prevent these attacks using tainted dataflow analysis [4,22,32].

Abadi *et al.* [33] propose formalizing the concept of Control Flow Integrity (CFI), observing that high-level programming often assumes properties of control flow that are not enforced at the machine level. CFI provides a way to statically verify that execution proceeds within a given control-flow graph (the CFG effectively serves as a policy). The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. However, such techniques generally focus on integrity protection at the expense of availability.

The acceptability envelope, a region of imperfect but acceptable software systems that surround a perfect system, as introduced by Rinard [34] promotes the idea that current software development efforts might be misdirected. Rinard explains that certain regions of a program can be neglected without adversely affecting the overall availability of the system. To support these claims, a number of case studies are presented where introducing faults such as an off-by-one error does not produce unacceptable behavior. This work supports the claim that most complex systems contain the necessary framework to propagate faults gracefully and the error toleration allowed (or exploited) by some self-healing systems expands the acceptability envelope of a given application.

4.1 Self-healing Techniques

Some first efforts at providing effective remediation strategies include failure-oblivious computing [20], error virtualization [19,21], rollback of memory updates [29,35], crash-only software [36], and data-structure repair [37]. The first two approaches may cause a semantically incorrect continuation of execution (although the Rx system [38] attempts to address this difficulty by exploring semantically safe alterations of the program's environment).

TLS Oplinger and Lam [35] employ hardware Thread-Level Speculation to improve software reliability. They execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code.

Failure-Oblivious Computing Rinard *et al.* [39] developed a compiler that inserts code to deal with writes to unallocated memory by virtually expanding the target buffer. Such a capability aims to provide a more robust fault response rather than simply crashing. The technique presented by Rinard *et al.* [39] is subsequently introduced in a modified form as *failure-oblivious computing* [20]. Because the program code is extensively rewritten to include the necessary checks for *every* memory access, the system incurs overheads ranging from 80% up to 500% for a variety of different applications. Failure-oblivious computing specifically targets memory errors.

Data-structure Repair One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. An important contribution in this area is that of Demsky and Rinard [37], which discusses mechanisms for detecting corrupted data structures and fixing them to match some pre-specified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate

when faults were randomly injected. Similar results are shown by Wang *et al.* [40]: when program execution is forced to take the "wrong" path at a branch instruction, program behavior remains the same in over half the times.

Rx In Rx [38], applications are periodically checkpointed and continuously monitored for faults. When an error occurs, the process state is rolled back and replayed in a new "environment". If the changes in the environment do not cause the bug to manifest, the program will have survived that specific software failure. However, previous work [41, 42] found that over 86% of application faults are independent of the operating environment and entirely deterministic and repeatable, and that recovery is likely to be successful only through application-specific (or application-aware) techniques.

Error Virtualization Error virtualization [19, 21] assumes that portions of an application can be treated as a transaction. Functions serve as a convenient abstraction and fit the transaction role well in most situations [19]. Each transaction (vulnerable code slice) can be speculatively executed in a sandbox environment. In much the same way that a processor speculatively executes past a branch instruction and discards the mispredicted code path, error virtualization executes the transaction's instruction stream, optimistically "speculating" that the results of these computations are benign. If this *microspeculation* succeeds, then the computation simply carries on. If the transaction experiences a fault or exploited vulnerability, then the results are ignored or replaced according to the particular response strategy being employed. The key assumption underlying error virtualization is that a mapping can be created between the set of errors that *could* occur during a program's execution and the limited set of errors that the program code explicitly handles. By virtualizing errors, an application can continue execution through a fault or exploited vulnerability by nullifying its effects and using a manufactured return value for the function where the fault occurred.

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (specifically, Java) [43,44]. That work focused on safely terminating misbehaving threads, introducing the concept of "soft termination". Soft termination allows threads to be terminated while preserving the stability of the language runtime, without imposing unreasonable performance overheads. In that approach, threads (or *codelets*) are each executed in their own transaction, applying standard ACID semantics. This allows changes to the runtime's (and other threads') state made by the terminated codelet to be rolled back. The performance overhead of that system can range from 200% up to 2,300%.

One approach that can be employed by error virtualization techniques is the one described by Locasto *et al.* [45], where function-level profiles are constructed during a training phase that can, in turn, be used to predict function return values. While this technique is useful for predicting appropriate return values, especially in the absence of return type information, it suffers from the same problems as error virtualization, *i.e.,* it is oblivious to errors.

ASSURE ASSURE [46] is an attempt to minimize the likelihood of a semantically incorrect response to a fault or attack. ASSURE proposes the notion of *error virtualization rescue points*. A rescue point is a program location that is known to successfully

propagate errors and recover execution. The insight is that a program will respond to malformed input differently than legal input; locations in the code that successfully handle these sorts of anticipated input “faults” are good candidates for recovering to a safe execution flow. ASSURE can be understood as a type of exception handling that dynamically identifies the best scope to handle an error.

DIRA DIRA [29] is a technique for automatic detection, identification and repair of control-hijacking attacks. This solution is implemented as a GCC compiler extension that transforms a program's source code adding heavy instrumentation so that the resulting program can perform these tasks. The use of checkpoints throughout the program ensures that corruption of state can be detected if control sensitive data structures are overwritten. Unfortunately, the performance implications of the system make it unusable as a front-line defense mechanism.

Vigilante Vigilante [4] is a system motivated by the need to contain rapid malcode. Vigilante supplies a mechanism to detect an exploited vulnerability (by analyzing the control flow path taken by executing injected code) and defines a data structure (Self-Certifying Alert) for exchanging information about this discovery. A major advantage of this vulnerability-specific approach is that Vigilante is exploit-agnostic and can be used to defend against polymorphic worms.

A problem that is inherent with all techniques that try to be oblivious to the fact that an error has occurred is the ability to guarantee session semantics. Altering the functionality of the memory manager often leads to the uncovering of latent bugs in the code [47].

5 Self-healing Systems: Future Directions

Given the embryonic state of the research in self-healing software systems, it should come as no surprise that there are significant gaps in our knowledge and understanding of such systems' capabilities and limitations. In other words, this is an extremely fertile area for further research. Rather than describe in detail specific research topics, we outline three general research thrusts: fault detection, fault recovery/mitigation, and assurance.

Fault Detection One of the constraining factors on the effectiveness of self-healing systems is their ability to detect faults. Thus, ways to improve fault detection at lower memory and computation cost will always be of importance. One interesting direction of research in this topic concerns the use of hardware features to improve fault detection and mitigation [48]. Another interesting area of research is the use of collections of nodes that collaborate in the detection of attacks and faults by exchanging profiling or fault-occurrence information [49]. Although such an approach can leverage the size and inherent usage-diversity of popular software monocultures, it also raises significant practical issues, not the least of which is data privacy among the collaborating nodes.

We also believe that the next generation of self-healing defense mechanisms will require a much more detailed dynamic analysis of application behavior than is currently

done, possibly combined with *a priori* behavior profiling and code analysis techniques. This will be necessary to detect application-specific semantic faults, as opposed to the "obvious" faults, such as application crashes or control-hijack attacks, with which existing systems have concerned themselves to date. Profiling an application can allow self-healing systems to "learn" common behavior [45]. The complementary approach is to use application-specific integrity policies, which specify acceptable values for (aspects of) the application's internal runtime state, to detect attacks and anomalies [50].

Fault Recovery/Mitigation To date, most systems have depended on snapshot/recovery, often combined with input filtering. The three notable exceptions are error virtualization, failure-oblivious computing, and data-structure repair. The former two techniques can be characterized as "fault masking", while the last uses learning to correct possibly corrupt data values.

The technical success of self-healing systems will depend, to a large extent, on their ability to effectively mitigate or recover from detected faults, while ensuring system availability. Thus, research on additional fault-recovery/masking/mitigation techniques is of key importance, especially when dealing with faults at different (higher) semantic levels. Similar to fault detection, two high-level approaches seem promising: profiling applications to identify likely "correct" ways for fault recovery [46], and using application-specific recovery policies that identify steps that the system must undertake to recover from different types of faults [50].

Assurance Finally, to gain acceptance in the real world, self-healing systems and techniques must provide reasonable assurances that they will not cause damage in the course of healing an application, and that they cannot be exploited by an adversary to attack an otherwise secure system. This is perhaps the biggest challenge faced by automated defense systems in general. Self-healing software systems of the type we have been discussing in this paper may need to meet an even higher standard of assurance, given the intrusiveness and elevated privileges they require to operate. Although to a large extent acceptance is dictated by market perceptions and factors largely outside the technical realm, there are certain measures that can make self-healing systems more palatable to system managers and operators. In particular, transparency of operation, the ability to operate under close human supervision (at the expense of reaction speed), "undo" functionality, and comprehensive fix-testing and reporting capabilities all seem necessary elements for a successful system. Although some of these elements primarily involve system engineering, there remain several interesting and unsolved research problems, especially in the area of testing.

6 Conclusions

We have outlined some of the recent and current work on self-healing software systems, describing the origins and design philosophy of such systems. We believe that self-healing systems will prove increasingly important in countering software-based attacks, assuming that the numerous research challenges (and opportunities), some of which we have outlined in this paper, can be overcome.

Acknowledgements This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-06-2-0221 and by NSF Grant 06-27473. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. 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* 36th *Hawaii International Conference on System Sciences (HICSS)*, January 2003.
2. A. D. Keromytis, J. Parekh, P. N. Gross, G. Kaiser, V. Misra, J. Nieh, D. Rubenstein, and S. Stolfo. A Holistic Approach to Service Survivability. In *Proceedings of the ACM Survivable and Self-Regenerative Systems Workshop*, October 2003.
3. X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet Vaccine: Black-box Exploit Detection and Signature Generation. In *Proceedings of the* 13th *ACM Conference on Computer and Communications Security (CCS)*, pages 37–46, November 2006.
4. Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, 1996.
6. A. Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.
7. 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.
8. 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.
9. H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.
10. V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 345–364, October 2003.
11. 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 10th ACM Conference on Computer and Communications Security (CCS)*, pages 321–334, October 2003.
12. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium*, 1998.
13. 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.
14. Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12th *USENIX Security Symposium*, pages 257–272, August 2003.
15. 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.

16. 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.
17. 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.
18. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the* 15th *USENIX Security Symposium*, pages 105–120, July/August 2005.
19. Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.
20. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of OSDI*, December 2004.
21. S. Sidiroglou, Y. Giovanidis, and A.D. Keromytis. A Dynamic Mechanism for Recovery from Buffer Overflow attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005.
22. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.
23. M. Dahlin. *Serverless Network File Systems*. PhD thesis, UC Berkeley, December 1995.
24. P. Ször and P. Ferrie. Hunting for Metamorphic. Technical report, Symantec Corporation, June 2003.
25. James Newsome, David Brumley, and Dawn Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13th *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.
26. Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael E. Locasto. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.
27. Zhenkai Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)*, November 2005.
28. Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 82–101, September 2005.
29. A. Smirnov and T .Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12th *ISOC Symposium on Network and Distributed System Security (SNDSS)*, February 2005.
30. Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)*, November 2005.
31. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11th *USENIX Security Symposium*, August 2002.
32. G. Edward Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.
33. Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

34. M. Rinard. Acceptability-oriented Computing. In *Proceedings of ACM OOPSLA*, October 2003.
35. Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.
36. George Candea and Armando Fox. Crash-Only Software. In *Proceedings of the* 9th *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.
37. B. Demsky and M. C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of ACM OOPSLA*, October 2003.
38. Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In Andrew Herbert and Kenneth P. Birman, editors, *Proceedings of ACM SOSP)*, pages 235–248, 2005.
39. M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In *Proceedings of ACSAC*, December 2004.
40. Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the* 12th *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.
41. S. Chandra and P. M. Chen. Wither Generic Recovery from Application Faults? A Fault Study using Open-Source Software. In *Proceedings of DSN/FTCS*, June 2000.
42. S. Chandra. *An Evaluation of the Recovery-related Properties of Software Faults*. PhD thesis, University of Michigan, 2000.
43. A. Rudys and D. S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.
44. A. Rudys and D. S. Wallach. Transactional Rollback for Language-Based Systems. In *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2001.
45. M. E. Locasto, A. Stavrou, G. F. Cretu, A. D. Keromytis, and S. J. Stolfo. Quantifying Application Behavior Space for Detection and Self-Healing. Technical Report CUCS-017-06, Columbia University Computer Science Department, April 2006.
46. Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh. Using Rescue Points to Navigate Software Recovery (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.
47. Frederick P. Brooks. *Mythical Man-Month*. Addison-Wesley, first edition, 1975.
48. M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Speculative Virtual Verification: Policy-Constrained Speculative Execution. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 170–175, September 2005.
49. M. Locasto, S. Sidiroglou, and A.D Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (SNDSS)*, Febrary 2006.
50. M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From STEM to SEAD: Speculative Execution for Automated Defense. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.