

Self-Healing: Science, Engineering, and Fiction

[Position Paper]

Michael E. Locasto
Department of Computer Science
Columbia University
locasto@cs.columbia.edu

ABSTRACT

Most attacks on computing systems occur rapidly enough to frustrate manual defense or repair. It appears, therefore, that defense systems must include some degree of autonomy. Recent advances have led to an emerging interest in self-healing software as a solution to this problem. It is not clear, however, if the effort to create self-healing mechanisms is actually worth the cost in terms of development effort, deployment complexity, or runtime supervision and monitoring. Furthermore, no general purpose self-healing mechanisms have been shown to be achievable for general systems; it is hard to know beforehand exactly what an application should do in response to an arbitrary vulnerability. A number of very hard problems remain for researchers to explore before self-healing can be reliably applied to real computing systems. This paper provides a critique of the current state of the art and offers the position that self-healing as a concept should be relegated to the status of autonomic computing: a goal worth aiming for as a way to push the boundary of the possible rather than an achievable end in itself. Along the way, we identify a number of important but unsolved research problems in this space.

General Terms

Security

Keywords

self-healing, autonomic computing, self-* systems, automatic repair

1. INTRODUCTION

Research on self-healing, in the context of system security, originates from one key question: *how do we go about building systems that react correctly to an arbitrary intrusion attempt?*¹ This question is not only intellectually stimulating, but it is also of great

¹In order to limit the scope of the paper, we refer to self-healing in terms of protection from attacks rather than the general class of failures, although the particular symptoms of an attack may not differ greatly from such events.

practical concern because most current computer security systems are only geared to protecting against threats that we already know about. In an ideal virtual world, or so the reasoning goes, systems could anticipate the occurrence of unknown attacks and respond appropriately without human supervision. This hope is the essence of one facet of the concept of autonomic computing.

Current software protection techniques typically abort a process after an intrusion attempt (*e.g.*, a code injection attack). Although this approach is considered “safe”, it is unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state. In contrast, self-healing mechanisms seek to mask or correct failures such that the protected application maintains its availability [25, 26].

Recent advances in self-healing software techniques have paved the way for autonomic intrusion reaction, but real world deployments of such systems have lagged behind research efforts. The limits of detection technology have historically mandated that researchers address the shortcomings of intrusion detection before reaction mechanisms (*i.e.*, self-healing mechanisms) are considered – an attack must be detected before a response can be mounted. In addition, many system administrators are reluctant to allow a defense system to make unsupervised changes to the computing environment, even though (and precisely because) a machine can react much faster than a human. Furthermore, this speed may not be matched by any type of precision or accuracy.

Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. It seems that enabling systems to be “smarter” about security is a natural research area to explore. It is difficult, however, to determine what this combination actually means in technical terms, especially since the expectations of self-healing virtual systems go far beyond the capabilities for real computing systems.

1.1 Real & Virtual Self-Healing

Self-healing, in a larger physical sense, happens in very constrained and specific environments. Human skin heals remarkably well, but humans are unable to regenerate entire limbs, as some simpler species (*e.g.*, crustaceans) are capable of doing. The laws of thermodynamics suggest that self-renewal and regeneration always comes at a cost — it is not an intrinsic law of nature or capability of physical systems to correctly regenerate. Breakdowns of regeneration systems can lead to massive failure — some types of cancer occur when the normal process of cell replacement breaks down and cells mutate and reproduce in an unregulated fashion.

In time, all systems fail. Organisms age and die, and mechanical systems and physical structures break down in a few months or years. Even physical systems that exist on a massive timescale are not immortal: stars run out of fuel. In contrast, humans imagine systems that never fail. In particular, we conceive of information

systems as capable of dealing with errors and continuing operation in the face of failure.

As computer scientists, we use abstraction to create the illusion of a perfect world for our virtual systems. In this environment, every process has enough memory space for its code and variables. Resources are always available and execution time is plentiful. Many of the most difficult problems in computer science research originate from the failure of this little fiction as it meets the constraints of the physical machines that host it and the assumptions and expectations of the users that interact with the system.

Virtual systems, therefore, must fail, often in spectacular ways and in response to seemingly small perturbations. Although we may imagine perfect systems, we lack the ability to translate the virtual ideal to a physical form while maintaining fidelity to the original. Virtual systems are implemented, hosted, and run on physical systems that have their own limitations and failure rates. Systems contain design and implementation flaws precisely because they are the product of an inherently human process. Unintentional flaws and failures are bad enough, and such difficulties have been the focus of research in fault tolerance and dependable systems for the past few decades. A skilled and determined attacker compounds the problem by actively subverting a system to undermine the will, desire, and ownership rights of the system's user, operator, or administrator.

1.2 Self-Healing

A major problem for security researchers has been to design and implement mechanisms that automatically recover from these attacks. This requirement raises the question: what mechanisms can researchers design ahead of time (that is, before an application is deployed) that can be woven into the execution of arbitrary applications that can handle attacks the researchers or system designers may not have anticipated? In concrete terms, this is like deploying a system with an empty exception handler that is generated and filled in at runtime. Self-healing then becomes the question of how to design a code generator that assesses the current application's control flow, data flow, environment, identifies the vulnerability being exercised, and creates a set of instructions that ameliorate the effects of the exploit and prevents the future exercise of said vulnerability.

Automatically identifying certain classes of vulnerabilities appears feasible; a large number of techniques can detect code injection attacks, and tainted dataflow analysis can help capture a precise notion of a vulnerability as a conjunction of conditions on the target addresses of branch instructions (*i.e.*, the control flow path). Systems like VSEF [28] and Vigilante [10] do exactly this. Systems like DART (Directed Automated Random Testing) or EXE [5] can help identify attack inputs before a system is deployed. From there, the next logical step is to find similar vulnerabilities in the codebase, either at runtime or during compilation. Beyond this, however, it is unclear what action is best, or how such actions can be evaluated. We next review some related work in this space and then enumerate a list of problems that must be addressed so that self-healing can move from the realm of science fiction to science and engineering.

2. RELATED WORK

Systems can self-heal in a variety of ways, and we briefly review some of the recent research on these systems. Unfortunately, self-healing is somewhat of an overloaded term, and it has traditionally been used to describe distributed systems consisting of many identical or near identical components that can transparently fail over to a backup or spare replica component. We are primarily concerned with heterogeneous systems like common COTS (Commercial Off

The Shelf) software. These software systems must automatically protect themselves from attacks delivered via previously unseen inputs or for previously unknown vulnerabilities. Detecting such attacks and vulnerabilities can make use of related work in both host-based and network-based anomaly detection.

2.1 Self-Healing

Intrusion defense mechanisms typically respond to an attack by terminating the attacked process, although some also attempt to generate either vulnerability [13] or exploit signatures [24, 43]. Effective remediation strategies remain a challenge, but some first efforts include failure oblivious computing [32], error virtualization [33], crash-only software [6], and data-structure repair [14]. Both error virtualization and failure oblivious computing represent significant performance overhead and have the potential to lead the application down a semantically incorrect execution path.

Rx [31] checkpoints execution in anticipation of errors. When an error is encountered, execution is rolled back and replayed with the process's environment changed in a manner consistent with the semantics of the APIs the code uses – an attempt to avoid a semantically incorrect response [32, 33]. Ganapathy *et al.* [18] attempt to retrofit enforcement of security policies and illustrate this process by inserting authorization checks into the X server.

2.2 Hardening Software

Methods to protect a process from subversion have typically focused on protecting the integrity of a process's jump targets. For example, StackGuard [11] and related approaches attempt to detect changes to the return address (or surrounding data items) of a stack frame. If the integrity of this value is violated, execution is halted. *Program shepherding* [22] validates branch instructions in IA-32 binaries to prevent transfer of control to injected code and to ensure that calls into native libraries originate from valid sources. Control flow is often corrupted because the system eventually incorporates input into part of an instruction's opcode, a jump target, or an argument to a system call. Recent work has focused on dataflow analysis of tainted data and ways to prevent such attacks [36, 10]. Abadi *et al.* [1] propose formalizing the concept of Control Flow Integrity, observing that high-level programming often assumes properties of control flow that remain unenforced at the machine language level. CFI statically verifies that execution stays within a control-flow graph (the CFG effectively serves as a policy).

2.3 Anomaly Detection

Anomaly classification of either host events [21, 34, 7, 30, 19, 20, 15, 2, 27] or network traffic [42, 40, 41, 23] is a powerful method of detecting inputs that are probably malicious without relying on a static set of signatures. This conclusion is based on the assumption that malicious inputs are rare in the normal operation of the system. However, since a system can evolve over time, it is also likely that new *non-malicious* inputs will be seen [17]. Indeed, sophisticated forms of mimicry attacks [38] have shown that it is possible to evade anomaly-based classifiers by constructing polymorphic blending attacks [16], and Taylor and Gates have suggested that anomaly detection (as originally conceived for host-based monitoring) is an ill fit for current network traffic [37].

2.4 Vulnerability Signatures

Recent work [39, 35] calls into question the ultimate utility of exploit-based signatures, and Brumley *et al.* [4] supplies an initial exploration of some of the theoretical foundations of *vulnerability-based signatures*. Vulnerability signatures help classify an entire

set of exploit inputs rather than a particular exploit instance. Identifying the underlying vulnerability may help reason about the appropriate action to take to heal or repair an attacked process.

As an illustration of the difficulty of creating vulnerability signatures, Crandall *et al.* [12] discuss generating high quality vulnerability signatures via an empirical study of the behavior of polymorphic and metamorphic malware. They outline the difficulty of identifying enough features of an exploit to generalize about a specific vulnerability.

Cui *et al.* [13] discuss combining tainted dataflow analysis (similar to that used in the Vigilante system [10]) and protocol or data format parsing [3] to construct network or filesystem level “data patches” to filter input instances related to a particular vulnerability. The system, ShieldGen, uses a feedback loop to perform local search of the exploit space. Newsome *et al.* [28] suggest generating and distributing vulnerability-specific execution filters (VSEF) based on the identification of a particular control flow path derived from taint analysis [29].

Bouncer [9] helps revive the notion that content filtering is a viable way to defend systems even in the presence of polymorphic exploit input. Bouncer (similar to ShieldGen) captures a sample exploit instance and the effect it has on an instrumented application (namely, the binary control flow path that the exploit input causes the application to take). Bouncer then enters a feedback loop intended to identify other control flow paths and, consequently, other conditions on input that help identify other “legal” exploit forms.

While ShieldGen also uses a feedback loop, the purpose of ShieldGen’s feedback loop is to use a protocol parsing engine to generate other inputs guided by the protocol grammar. In contrast, Bouncer performs program slicing to identify actual control flow paths in the application. This approach is more robust, as there is no guarantee that the published protocol specification matches the actual application code. While no solution can be simultaneously sound and complete, it is not clear that ShieldGen provably provides either property. Bouncer filters are sound; it remains unclear at the time of writing, however, if Bouncer extensively covers the space of possible exploit inputs in a reasonable amount of time, as program slicing is a potentially expensive form of analysis.

Bouncer offers a realistic way forward for intrusion defense; it walks a fine line between simple content-based exploit filters and more drastic self-healing approaches. The former can be defeated by almost trivial polymorphic techniques, as we show in other work [35], and the latter have a major problem sustaining application semantics (a problem that this thesis, in part, addresses). Thus, system defenders are faced with a bit of quandary: they would like to use content filtering (it has fairly cheap runtime overhead, can protect more than a single application at a time, and is easy to implement), but it is almost trivial for a smart and determined attacker to bypass. On the other hand, although more invasive (and complicated) defense mechanisms can defeat a family or class of exploits, these mechanisms often cause a self-induced DoS (*i.e.*, application crash) at best.

3. DISCUSSION ITEMS

The ability for a process to automatically repair itself in the face of previously unseen attacks that exploit unknown vulnerabilities remains a key open problem in information systems security. The major difficulty stems from the fact that systems designers lack prescience: they cannot write code to handle vulnerable states they cannot or have not yet conceived. Even though they may be able to write code that detects well-known classes of attack, or that handles anticipated faults (such as the failure to open a file), systems can fail in unexpected ways, and it is non-obvious what actions

should be undertaken by the system to “self-heal.”

Furthermore, most current systems only seek to repair memory corruption attacks. True self-healing systems have to anticipate and fix unknown classes or types of attack. This requires high-level reasoning: a repair system that forms some suspicion and is then able to test those conjectures.

3.1 Attacker Intent

Attacks themselves may be symptoms of a larger effort aimed at a nebulous (from the defender’s standpoint) goal. While individual steps of an attack can be repeatedly frustrated or turned away by a self-healing mechanism, understanding the root cause or intent of an attacker’s actions based on a series of attack events may help streamline repair efforts or otherwise improve the efficiency of defense. A more nimble defense can help system defenders regain the initiative from the attacker; the philosophy behind OODA decision feedback loops are examples of this capability.

This challenge is perhaps the most difficult to surmount. Security is, at its core, the imposition of one principal’s will on another. Since this definition is morally neutral (it makes no judgment about the motivation of the principals), it creates the difficulty faced by any automated defense: inferring malicious intent. Determining the intent of some action or event is hard for humans, and it is unlikely that inferring intent is a concept that can be easily modeled by a computational process. Nevertheless, understanding the root cause of attacks requires making a judgment about the input and actions of components in the system.

3.2 Automatic Repair Validation

System owners are understandably reluctant to permit automated changes to their environment and applications in response to attacks. The risks of doing so range from legal liability to severe consequences like death or injury (in the case of automated factory machinery or medical equipment). Testing an automatic repair helps raise the confidence level in self-healing systems, although it can never prove a repair mechanism correct.

One critical part of such testing is the verification that the changes made by the self-healing mechanism actually defeat the original attack or close variations thereof. Another problem involves ensuring that the changes or repairs do not introduce new vulnerabilities. Automatically generated fixes must be subjected to rigorous testing in an automated fashion. This problem is the essence of Automatic Repair Validation (ARV), a new area of intrusion defense research. ARV consists of automatically validating each step in the process toward a newly healed configuration². ARV encompasses the entire spectrum of an automated response system’s functionality: attack detection, repair accuracy, repair precision, and impact on normal behavior:

1. *Validation of detection* — The system must verify that the events causing an alert actually produce a compromise. In

²A reader asked whether it would be enough to simply validate the final configuration. For certain ARV problems (listed above), validating the final configuration is precisely the purpose of a solution to each problem. For example, relaunching the input responsible for the original attack against the healed configuration provides an independent sanity check on the final configuration. We distinguish between testing the final configuration and testing individual steps (*i.e.*, detection, diagnosis, repair, deployment, *etc.*) because each of these steps may be prone to error or false positives themselves — we certainly do not wish to enact a potentially costly repair stage if it is not needed. In some cases, the complexity of the software application under consideration may preclude the construction of simple measurements of the final configuration.

the case where the sensor is an anomaly detector, the detector's initial classification must be confirmed.

2. *Validation of a repair's accuracy* — The system must test and verify that the repair defeats at least the exploit input that triggered the detection. Verifying accuracy requires the identification and replay of the attack inputs. However, identifying these inputs is challenging, as they may not have been captured correctly (or at all) by the defense instrumentation. The challenge is greater if the input is contained in network traffic — data that most humans find difficult to rapidly analyze by hand.
3. *Validation of a repair's precision* — The fix must be precise, in the sense that it blocks malicious variants of the original attack. For example, if the fix is an input filter, the system must ensure that the signature generation does not fall prey to an allergy attack [8].
4. *Validation of a repair's impact on application behavior* — Behavior exhibited by the application after self-healing should be similar to the previous behavior profile of the application. Researchers need to invent measures for bounding the semantic correctness or behavior of an application. Control flow graph distances may be one way to measure changes due to self-healing repairs.

Some repairs are better than others. For self-healing systems that perform some sort of state search, this implies that the search mechanism needs an evaluation function to rate progress toward a certain locally or globally optimal solution. Even if exhaustive search is employed, the solution may not be sound: it may cause a semantically incorrect response. The delay, or amount of time taken to effect a repair, therefore, is probably not the best measurement of the quality of the repair process.

The last ARV challenge depends greatly on quality behavior profiling. Profiling the behavior of applications (especially when the application is a black box or in situations where we lack source code access) can assist efforts to **detect** deviations from normal operation, **repair** aspects of the data and control flow, and **validate** those repairs.

3.3 General Problems

Even the term “self-healing” is somewhat of a misnomer. Systems need some sort of external reference monitor built as part of the execution environment to actually supervise an application's actions and decide to “fix” the outcome. The reference monitor needs some sort of insulation between itself and the attacked application (clearly, they should not reside in the same memory space without adequate protection). The reference monitor is needed to intercept and filter execution. This is a well understood mechanism, but *what* it does and *when* it does it are the research questions. The *how* is usually a choice of engineering or implementation: a VMM, emulation, dynamic binary rewriting, modified hardware, etc.

Determining just when a repair should occur is a straightforward but non-trivial decision that affects the architecture and deployment of a self-healing system. It seems as if a repair should be enacted immediately after an attack is detected. However, the attack may have corrupted much of the system state for an indeterminate amount of time before detection. Therefore, a self-healing implementation must provide some time interval to enact a repair. It has three choices: (1) anticipate an attack by speculatively executing a slice of an application, (2) attempt to vet and repair each instruction in an application as it is executed, or (3) retroactively

replace and repair actions taken by the application from the time the exploit first entered the system to the time it was detected.

Speculative, in-place, and retroactive approaches must all deal with I/O (*i.e.*, file or network I/O, interrupts, and other events) that occurs during and before the repair period. Such communication changes the global state of the world, and illegal or attacker-controlled messages cannot be taken back (*e.g.*, if the exploit causes financial information to be leaked, it is difficult to retrieve that data from an uncooperative or malicious communications partner).

Currently, repairing a system after a successful intrusion is costly because cleanup is largely a manual process, and the complexity of modern information systems makes it difficult to conclusively trace the extent of the corruption. When this process *is* automated, it typically resets systems to some initial state, thus deleting valuable data that may not have been backed up. Environments that cannot afford comprehensive backup mechanisms would be better served by a process that can work backward from an attack to undo the specific damage caused.

```
/usr/src/cmd/cmd-inet/usr.sbin/in.telnetd.c
3198
3199 } else /* default, no auth. info available, \
        login does it all */ {
3200     (void) execl(LOGIN_PROGRAM, "login",
3201                "-p", "-h", host, "-d", slavename,
3202                getenv("USER"), 0);
3203 }
```

```
/usr/src/cmd/login/login.c
1397     break;
1398
1399 case 'f':
1400     /*
1401      * Must be root to bypass authentication
1402      * otherwise we exit() as punishment for trying.
1403      */
1404     if (getuid() != 0 || geteuid() != 0) {
1405         audit_error = ADT_FAIL_VALUE_AUTH_BYPASS;
1406
1407         login_exit(1);    /* sigh */
1408         /*NOTREACHED*/
1409     }
1410     /* save fflag user name for future use */
1411     SCPYL(user_name, optarg);
1412     fflag = B_TRUE;
```

Figure 1: Solaris 10 telnet bug. The telnet code assumes that `login` will perform appropriate authentication; `login` only tests whether the daemon is running as root and, if so, allows authentication to proceed with the supplied username (given as the argument option value to the `-f` parameter). An attack that leverages this coding mistake simply provides a valid input sequence.

It's not even clear what self-healing means in the context of bugs like the Solaris 10 telnet bug (see Figure 1): no stack or heap corruption occurs, no code injection occurs, no crypto is broken, no password is guessed. Rather, a “valid” input sequence is used to provide more access than is “intended.” But how do we measure intent? One can easily say that the developer's intent was to create such a valid control flow path. Only after a review is the input path judged to be inconsistent with the intent and function of the program in some environment (*i.e.*, in other environments, it may be OK for that sort of input path to exist) is a vulnerability declared to exist.

3.4 Discussion Questions

1. How do we write and place calls to generic fault handlers in

arbitrary software? Can we package some automated reasoning solution into system libraries and train developers to use them, much like developers should (in theory, at least) remember to check the return values of function calls?

2. Does self-healing imply only restorative or stop-gap measures, or does it entail a learning response in which the system is made more robust as a result of the attack? As one reviewer put it, “The response to a cut is to repair the flesh, not to make it cut-proof.”
3. Is “self-healing” an appropriate term, given that if the integrity of the system is violated, we cannot be sure that the healing mechanism itself remains undisturbed?
4. Is a self-healing capability actually a desirable goal for commodity goods like cars and computer systems? Do designers and manufacturers price themselves out of a market by offering a highly robust (and correspondingly high cost) product? Is there a real economic incentive to make perfect, self-healing software? The system must supply a balance of “good enough” to satisfy system owners without being exorbitantly costly to design, develop, and maintain (arguably, a system that is serviced after every instruction executed may work perfectly for many years, but the cost of such maintenance is certainly unreasonable for any realistic application).
5. Current detection, defense, and self-healing systems are evaluated in an ad hoc fashion against a self-selected set of exploits or vulnerabilities. There is no community-wide benchmark suite or collection of vulnerabilities for evaluating such systems in a standard manner. What would such a testbed look like?

4. WORKSHOP DISCUSSION

A number of issues were raised during the discussion of this topic at the workshop. In particular, a distinction emerged that reflected the differing goals of biological self-healing (a very loose definition of “correct”) and the implied requirements of information systems healing. In particular, information systems healing seems to require a root cause analysis whereas adaptations in biological systems simply need to be “good enough” to fulfill a primarily reproductive purpose. While self-healing need not be perfect, it can be difficult coming up with arbitrary and relative measures of correctness or value for a self-healing response. If such measures can be formulated, it may be enough to determine a threshold rather than hold out for a perfect result.

There seems to be a fundamental distinction between individual self-healing and group or population self-healing. The concept of systems evolution and self-healing in individuals and groups is different: biological evolution has had a large amount of time to explore a space with almost random perturbations; the search does not necessarily need to be directed because it has time to cover the space. In contrast, practical information systems must operate on a human timescale and must be far more directed. While a healed configuration can be viewed as a state in a solution space, it may be unlikely to find such a solution through undirected knob-twiddling in a short amount of time. Increased search time reduces the availability of a system, directly contradicting the implied purpose of a self-healing response: to keep the system available in the face of attack.

As one participant suggested, self-healing can be accomplished by an exhaustive process of knob-twiddling, or by evolving a system over a period of years or decades. The participant suggested

an example of such a robust system is the telephone switching network. An observation was made that most code in the system was supervisory code meant to support the availability of the system rather than core code delivering the key functionality of the system. Such a system has clear similarities to biological systems that spend most of their time and energy on maintaining a homeostatic environment, but the cost of doing so may not be acceptable for current computing platforms. Perhaps as threats and attacks increase in number and sophistication, computing platforms will be driven to relinquish large amounts of performance in exchange for security and self-healing measures.

5. CONCLUSIONS

Program designs will always lack a complete description of how to handle all errors. System designers cannot remove all vulnerabilities before deployment, nor can the plethora of proactive protection techniques be used to protect any arbitrary application before deployment against new classes of attacks, or higher-order attacks, or events that can only currently be judged malicious post-hoc by a human decision.

No system can be perfectly secure, but research to provide well-formed, automatic recovery mechanisms seems like an attractive possibility. Most recent research, however, has barely scratched the surface by showing limited utility for a single mechanism on a limited set of applications. Software self-healing, therefore, is a mixture of science, engineering, and science fiction. It remains to be seen whether the science can catch up with the fiction in a way that our engineering knowledge supports.

Acknowledgments

The anonymous reviewers went a long way toward helping shape the text, cut some extraneous or overwritten verbiage, and focus on the core issues. We greatly appreciate their comments. We also thank Matt Burnside for his discussions and work on repair validation.

6. REFERENCES

- [1] ABADI, M., BUDI, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2005).
- [2] BHATKAR, S., CHATURVEDI, A., AND SEKAR, R. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).
- [3] BORISOV, N., BRUMLEY, D. J., WANG, H. J., DUNAGAN, J., JOSHI, P., AND GUO, C. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the 14th Symposium on Network & Distributed System Security (NDSS)* (Feb 2007).
- [4] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).
- [5] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006).
- [6] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS-IX)* (May 2003).

- [7] CHARI, S. N., AND CHENG, P.-C. BlueBoX: A Policy-driven, Host-Based Intrusion Detection System. In *Proceedings of the 9th Symposium on Network and Distributed Systems Security (NDSS 2002)* (2002).
- [8] CHUNG, S. P., AND MOK, A. K. Allergy Attack Against Automatic Signature Generation. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2006).
- [9] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing Software By Blocking Bad Input. In *Proceedings of the ACM Symposium on Systems and Operating Systems Principles (SOSP)* (2007).
- [10] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).
- [11] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium* (1998).
- [12] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (November 2005).
- [13] CUI, W., PEINADO, M., WANG, H. J., AND LOCASO, M. E. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2007).
- [14] DEMSKY, B., AND RINARD, M. C. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (October 2003).
- [15] FENG, H. H., KOLESNIKOV, O., FOGLA, P., LEE, W., AND GONG, W. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003).
- [16] FOGLA, P., AND LEE, W. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)* (2006), pp. 59–68.
- [17] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (1997), pp. 67–72.
- [18] GANAPATHY, V., JAEGER, T., AND JHA, S. Retrofitting Legacy Code for Authorization Policy Enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).
- [19] GAO, D., REITER, M. K., AND SONG, D. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2004).
- [20] GIFFIN, J. T., DAGON, D., JHA, S., LEE, W., AND MILLER, B. P. Environment-Sensitive Intrusion Detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005).
- [21] HOFMEYR, S. A., SOMAYAJI, A., AND FORREST, S. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security* 6, 3 (1998), 151–180.
- [22] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium* (August 2002).
- [23] KRUEGEL, C., TOTH, T., AND KIRDA, E. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)* (2002).
- [24] LIANG, Z., AND SEKAR, R. 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).
- [25] LOCASO, M. E., CRETU, G. F., STAVROU, A., AND KEROMYTIS, A. D. A Model for Automatically Repairing Execution Integrity. Tech. Rep. CUCS-005-07, Columbia University, January 2007.
- [26] LOCASO, M. E., STAVROU, A., CRETU, G. F., AND KEROMYTIS, A. D. From STEM to SEAD: Speculative Execution for Automatic Defense. In *Proceedings of the USENIX Annual Technical Conference* (June 2007), pp. 219–232.
- [27] MUTZ, D., VALEUR, F., VIGNA, G., AND KRUEGEL, C. Anomalous System Call Detection. *ACM Transactions on Information and System Security* 9, 1 (February 2006), 61–93.
- [28] NEWSOME, J., BRUMLEY, D., AND SONG, D. 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).
- [29] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS)* (February 2005).
- [30] PROVOS, N. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium* (August 2003), pp. 207–225.
- [31] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (2005).
- [32] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND W BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).
- [33] SIDIROGLOU, S., LOCASO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference* (April 2005), pp. 149–161.
- [34] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium* (August 2000).
- [35] SONG, Y., LOCASO, M. E., STAVROU, A., KEROMYTIS, A. D., AND STOLFO, S. J. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2007).

- [36] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 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).
- [37] TAYLOR, C., AND GATES, C. Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In *Proceedings of the 15th New Security Paradigms Workshop (NSPW)* (September 2006), pp. 21–29.
- [38] WAGNER, D., AND SOTO, P. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (November 2002).
- [39] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM* (August 2004).
- [40] WANG, K., CRETU, G., AND STOLFO, S. J. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 227–246.
- [41] WANG, K., PAREKH, J. J., AND STOLFO, S. J. ANAGRAM: A Content Anomaly Detector Resistant To Mimicry Attack. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2006).
- [42] WANG, K., AND STOLFO, S. J. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2004), pp. 203–222.
- [43] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (November 2005).