Bloodhound: Searching Out Malicious Input in Network Flows for Automatic Repair Validation

Michael E. Locasto, Matthew Burnside, and Angelos D. Keromytis

Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
{*locasto, mb, angelos*}*@cs.columbia.edu*

Abstract. Many current systems security research efforts focus on mechanisms for Intrusion Prevention and Self-Healing Software. Unfortunately, such systems find it difficult to gain traction in many deployment scenarios. For self-healing techniques to be realistically employed, system owners and administrators must have enough confidence in the quality of a generated fix that they are willing to allow its automatic deployment.

In order to increase the level of confidence in these systems, the efficacy of a 'fix' must be tested and validated after it has been automatically developed, but before it is actually deployed. Due to the nature of attacks, such verification must proceed automatically. We call this problem Automatic Repair Validation (ARV). As a way to illustrate the difficulties faced by ARV, we propose the design of a system, *Bloodhound*, that tracks and stores malicious network flows for later replay in the validation phase for self-healing software.

Keywords: intrusion reaction, secure self-healing, automatic repair validation

1 Introduction

Finding and identifying malicious input, events, and output are the fundamental tasks of intrusion detection systems. Constructing such systems and reasoning about their properties are the major aims of intrusion detection research.

A key problem in this space is the inability of systems to automatically protect themselves from attack. In order to have a reasonable chance at surviving or deflecting current and emerging attacks, a system must incorporate automatic reaction mechanisms. Recent advances in self-healing software techniques (we refer the reader to Section 2 for in-depth coverage of this topic) have paved the way for autonomic intrusion reaction. However, realistic deployments of such systems have lagged behind research efforts.

The limits of detection technology have historically mandated that the shortcomings of intrusion detection (false positives and negatives, fail-open nature,

performance, etc.) be addressed before reaction mechanisms are considered – an attack must be detected before any response can be mounted. In addition, many system administrators are understandably reluctant to allow an automated defense system to make unsupervised changes to the computing environment, even though (and precisely because) a machine can react orders of magnitude faster than a human.

1.1 Automatic Repair Validation

The unpredictable nature of attacks, the imprecision of detection mechanisms, and the dearth of analysis for automatic response systems combine to foment a lack of confidence and a justifiable degree of skepticism about the use of automated defense and self-healing systems. Thus, automatically-generated fixes must be subjected to very detailed, rigorous, and intense testing in an automated fashion. This problem is the essence of Automatic Repair Validation (ARV), an area of intrusion defense research deserving of further exploration. It is important to note that we are not advocating relinquishing human oversight, but rather advocating the reduction of human involvement (and response time) in the critical path of protection.

The ARV concept includes the automation of a variety of tests, including:

- an **installation test** that both performs and checks the application of the fix or patch to the protected system (e.g., configuration changes, compilation from source, assembly, relinking, or binary rewriting/patching)
- a series of **unit regression tests** to verify that the fix has not introduced any new errors or changed the normal operation of the system
- a series of **end-to-end regression tests** on both normal and malformed input to verify that the "healed" system behaves as expected when interacting with external components
- an **efficacy test** to show that the fix protects against the input that triggered the self-healing procedure

While the first three types of tests are standard best practices for software engineering, it is the efficacy test that we are particularly concerned with. It entails the identification and replay of the attack inputs. However, identifying these inputs is challenging; they may not have been captured correctly (or at all) by the defense instrumentation.

The challenge is increased if the input is contained in network traffic – data that many humans find difficult to rapidly classify, analyze, and understand "by hand." Performing automated testing in a manner that is transparent to the application is even more difficult.

Even in the face of these challenges, performing the efficacy test is worthwhile because it provides some assurance that the "healed" application is actually more resilient to attack.

1.2 Contributions

This paper makes the following three main contributions:

1. the identification of the problem of Automatic Repair Validation (ARV), which is the problem that Self-Healing Software faces when trying to validate the results of its self-healing mechanisms.
2. the recognition of the need for developing high-quality, widely used benchmarks and standards for assessing the response of a self-healing mechanism. ARV is an attempt to frame the discussion about formal testing and analysis of self-healing techniques; such analysis in the literature is self-selected and not easily reproducible.
3. an overview of the challenges faced by ARV-capable systems, especially in a networked environment. This overview provides the broad outlines for a research agenda.

We also provide a summary of current literature on self-healing systems and network capture and replay tools. Finally, we consider the design of an ARV system (Bloodhound) dealing with attack input that is contained in network flows. For example, Bloodhound is designed to act as the ARV component of a system for dynamically containing buffer overflows [17] in network-facing services.

Although the general ARV problem exists for many types of systems, this paper focuses on the challenges for an ARV system that monitors network-centric applications. Network-centric applications are popular targets for attack due to the relative anonymity an attacker has and the ease with which input can be sent to the system. The complexity of networked systems tends to be greater than other systems – this more complicated problem domain is useful to illustrate the array of possible challenges faced by an ARV system.

2 Related Work

The classification of input and events is the fundamental purpose of intrusion detection systems, and ARV is predicated on appropriately classifying system input. The primary task of network-based intrusion detection systems is to scan network packets and flows for content that matches known signatures or falls outside the range of the normal model of traffic [24] [8]. The goal of ARV is to further classify suspected malicious flows, correlate them with failures and fixes, and then faithfully replay the relevant flows that triggered the initial exploit or vulnerability detection instrumentation.

2.1 Identifying Malicious Traffic

The first crucial task for an ARV system is to keep track of input that is potentially malicious and correlate it with events that are produced by the self-healing instrumentation. In order to classify malicious input, a network-centric ARV system can take advantage of a variety of NIDS, but the quality and nature of its

classification will change based on the actual system in use. For example, misuse detectors (e.g., Snort's primary use case) that rely on signatures to detect malicious traffic will be unable to tell the ARV system about new attacks. Instead, ARV will only be able to validate repair of attacks that are already well-known enough to have a signature. Since such signatures are often manually created, it seems somewhat futile to base an ARV's classification capability on these types of misuse detectors – unless the signature is automatically generated as well, something that several systems [6] [19] [12] [11] [9] [25] [10] aim at doing.

To generate a signature, most of these systems either examine the content or characteristics of network traffic or instrument the host to identify malicious input. Two other interesting systems take a hybrid approach and seek to perform host-type processing on network flow data.

The key idea of Abstract Payload Execution (APE) [22] is to identify network traffic that contains malicious code by treating the content of a logical packet (in their experiments, an HTTP request) as machine instructions. Abstract execution (essentially instruction decoding) of various snippets of packet payload can help to identify the *sledge* (also known as a *sled*) – the sequence of instructions in an exploit whose sole purpose is to guide the program counter toward the actual meat of the exploit code. The main hypothesis is that a relatively long sequence of successful decodings of bytes indicates a sledge. Their experiments with "normal" traffic reveal very short (average of about 4, maximum of 16) executable byte sequences. In contrast, the tested exploits contained executable sequences on the order of hundreds of bytes long.

In [2], the authors present a system for detecting exploit code inside network flows. The exposition of the challenges involved complement the issues we raise in this paper. The authors discuss the use of *convergent static analysis*, a limited form of disassembly and binary interpretation that aims at revealing the rough control and data flow of a random sequence of bytes. This technique is similar to [7]'s proposal to detect polymorphic worms by learning a control flow graph for the worm binary.

2.2 Replaying Traffic

The ability to replay traffic is the second crucial aspect of a network-centric ARV system. Hong and Wu [5] describe the design and implementation of a system that can interactively replay network traffic. While TCPopera is broadly applicable to problems that require the ability to quickly and repeatedly produce large amounts of realistic network data (testing new applications, protocols, network stacks, etc.), it is particularly well–suited to ARV. In fact, the authors mention the related problem of testing Intrusion Prevention Systems (IPSs) as a useful application of TCPopera. One advantage of TCPopera is that the system carefully avoids generating side-effect "ghost" packets during replay, and it faithfully reproduces the timing semantics of the original traffic.

The major challenges in reproducing network traffic depend on which of the two major approaches to traffic generation is selected. First, the raw packet streams can be recorded and replayed, but this approach lacks finesse because

it treats packets as black boxes. Thus, it is difficult to adjust or modify the parameters to reflect realistic test conditions. In addition, it may require large amounts of storage. The second approach is based on building an analytical model of traffic and then generating traffic that matches these characteristics. While this approach avoids the cost of storing large amounts of data and provides flexibility in configuration, the predictions of the model may not be correct.

The RolePlayer system of Cui *et al.* [4] tackles the problem of reconstructing and mimicking application-level messages from network flows with very little contextual data and a few guiding heuristics. One of the use cases for RolePlayer that the authors mention is as a way to drive testing of network defense systems.

2.3 Self-Healing Software

ARV is only meaningful if a system incorporates or is protected by a self-healing mechanism. Most of these mechanisms follow what we term the ROAR (Recognize, Orient, Adapt, Respond) workflow. ARV is a logical and important part of the "Respond" stage: verification of the system adaptation.

Self-healing mechanisms are the subject of active research efforts. Rinard *et al.* [15] have developed compiler extensions that insert code to deal with access to unallocated memory by expanding the target buffer (in the case of writes) or manufacturing a value (in the case of reads). This technique is leveraged for *failure-oblivious computing* [16].

A related idea is that of *error virtualization*, which forms the basis for the self-healing mechanism in STEM [18], an emulator that performs transactional monitoring of application functions. 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 are explicitly handled by the program code. By virtualizing the errors, an application can continue execution through a fault or exploited vulnerability by nullifying the effects of such a fault or exploit and using a manufactured return value for the function where the fault occurred.

The Rx system [14] seeks to improve on the error virtualization and failure oblivious computing approaches by performing only safe perturbations of application state to self-heal. The key idea of Rx is that when an error occurs, execution should be rolled back and replayed, but with the process's environment changed in a way that does not violate the API's its code expects. For example, the result of `malloc()` must be a buffer of at least the requested size, but that buffer may be located at a different offset than the original, be padded at both or either end, or be cleared to zero. This procedure is iterated over, with 'fixes' becoming more expensive, until execution proceeds past the detected error point. The error and the transparent environment fix are then recorded for the programmer to debug. This is a clever way to avoid the semantically incorrect fixes of failure oblivious computing and error virtualization.

DIRA [20] is a compiler extension that adds instrumentation to keep track of memory reads and writes and check the integrity of control flow transfer data

structures. If the integrity fails, then the changed data is extracted and a network filter is created from it. Execution is recovered to a safe state.

There are a number of systems that are closely related to DIRA's basic goals. Liang and Sekar [9] and Xu et al. [25] concurrently propose using address space randomization to drive the detection of memory corruption vulnerabilities and create a signature to block further exploits of this type. In addition, to improve detection, such errors are correlated with program state. FLIPS [10] is a system that attempts to self-heal by providing feedback to an anomaly detector to block confirmed code injection attacks. FLIPS uses instruction set randomization to detect injection attacks and employs STEM's [18] error virtualization to self-heal. The idea of Shadow Honeypots [1] is a similar proposal.

Vigilante [3] is a system motivated by the need to contain Internet worms. To that end, Vigilante supplies a mechanism to detect an exploited vulnerability. A major advantage of this vulnerability-specific approach is that Vigilante is exploit-agnostic and can be used to defend against polymorphic worms. While Vigilante doesn't address the self-healing of a piece of exploited software, it defines an architecture for production and verification of Self-Certifying Alerts (SCA's), a data structure for exchanging information about the discovered vulnerability. Vigilante works by analyzing the control flow path taken by executing injected code.

The pH system [21] is an automatic reaction system that is aimed at frustrating an attacker by using system call interposition to slow down an attacker's code. While not strictly self-healing, this system was among the first to propose an active reaction mechanism to foil attacks, and is representative of the seminal work in artificial immune systems.

Song and Newsome [13] propose dynamic taint analysis for automatic detection of overwrite attacks. Tainted data is monitored throughout the program execution and modified buffers with tainted information will result in protection faults. Once an attack has been identified, signatures are generated using automatic semantic analysis. The technique is implemented as an extension to Valgrind and does not require any modifications to the program's source code. However, like other binary rewriting or emulator-based approaches, it suffers from a significant performance degradation. The authors extend the system [11] with vulnerability-specific execution filters (VSEF), an idea with a different mechanism, but similar goals to Shield [23].

Finally, it is important to distinguish between secure self-healing systems and Intrusion Prevention Systems, as ARV is meant to be used in conjunction with the sort of systems mentioned in the literature. On the other hand, IPS, at least as it is realized in commercial systems today, is only one primitive form of self-healing. The holy grail of research in self-healing software is a system that automatically recognizes previously unseen attacks (and their polymorphic variants) that exploit previously unknown vulnerabilities, prevents or undoes any damage incurred from the malicious input, and alters itself to defeat future instances or variants of the attack – all without having any detrimental impact on normal operation.

In contrast, the state of the art in IPS essentially deploys coarse network filtering rules based on recognizing a signature of malicious traffic. The research literature highlighted above describes more comprehensive approaches, and it is these approaches that ARV is meant to be used with. Something as straightforward as testing whether or not a network switch is blocking a host infected with Code Red from sending messages to other hosts on port 80, although entirely within its scope, is not the primary goal of ARV.

3 Approach

Assuming that a system exists for generating a self-healing patch or fix, the primary goal of an ARV-capable system is to automatically validate a system's newly "healed" configuration. In order to automatically employ this fix, the ARV must ensure two properties. First and foremost, the fix must defend against the actual attack input that initiated the self-healing process. Second, the patch must not interfere with the normal operation of the system.

While the second problem can be addressed by traditional regression testing, the first problem remains unsolved, primarily because of the difficulty of obtaining the attack input. Although one potential approach is to use the audit information available from IDS software as hints to focus the search, there are a number of challenges to be dealt with when solving this problem.

3.1 Challenges

A system cannot maintain an indefinite log of traffic. Even if such a log were available, searching through it may be a lengthy process and thus slow the automatic deployment of a fix. Furthermore, it is moderately difficult to identify traffic that is related to a particular alert. If the log only contains alerts, the information contained in an alert is often not detailed enough to reconstruct the packets of the attacking flow(s).

A simple approach would be to replay the entire traffic log to ensure that the attack packets are replayed, but it is most likely not necessary or expedient to replay (for example) three months of traffic, particularly if the attack is contained in one or two recent flows of only a few packets each. In addition, replaying an entire traffic log may represent an exorbitant use of resources, especially on a busy server.

Regardless of the specific mechanisms used, this process should be as transparent as possible to the attacked application. However, the cost of transparency may be quite high, requiring entire mirror networks to be set up (either *a priori* or on the fly) to provide a testbed.

Finally, the replay must balance the tension between fidelity to the original stream and the current state of the world. While this problem can potentially be solved by creating an isolated virtual network in which to replay traffic against checkpoints of the application, care must still be taken in selecting which state to preserve or retain.

3.2 Bloodhound

The three main tasks of Bloodhound are to (*a*) preferentially record network traffic, (*b*) search through recorded flows, and (*c*) replay the appropriate flows to test the auto-generated fix. There are a number of ways to preferentially record flows. One simple approach is to only record those that match signatures of known malicious content (such as known worm signatures). Another is to use an AD like PayL [24] or Anagram to mark flows that are anomalous with respect to the normal model of traffic for the host. Systems like APE (and the related control flow graph modeling) can also be used. Tools like `tcpdump`, TCPopera, TCPrecord, RolePlayer, or TCPflow are useful for actually recording and/or replaying traffic. A more elegant solution can involve marking variables or memory locations as tainted by particular packets.

3.3 Caveats and Limitations

Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed or even well-formed. Ideal computing systems would recover from attacks and errors without human intervention. However, the state of the art is far from mature, and most existing response mechanisms are external to the system they protect. Some simply crash the process that was attacked (and do nothing to fix the fault, thereby ensuring that the system is still vulnerable when it is rebooted). Other systems may restrict network connectivity or resource consumption. The main challenge of ARV is to provide some evidence that an automated response at least protects against input the triggered the self-healing mechanism.

The Bloodhound system faces two types of difficulties. The first is the horizon problem: malicious input may have entered the system before Bloodhound started operating. Additionally, Bloodhound may be unable to keep a record of all malicious flows due to physical memory limitations. The second problem involves correlating the form of input that triggers the self-healing instrumentation with the input that originally enters the system. Input may have been transformed by the system a number of times before it is finally recognized as malicious. There may be no way to transform the input back to its original form, and an ARV system may find it difficult to correlate a particular host-based event with malicious network traffic.

However, these hybrid ARV systems (those that examine both network and host-based data) still stand a better chance than a system that only considers either type of information in isolation – many host-based instrumentation mechanisms would be completely unable to reconstruct the original input (or a suitable representation thereof) because they either do not actually capture the attack input, or the input has undergone difficult-to-reverse transformations. ARV systems that examine network traffic in addition to host-based events are in a position to record the input before it is used by the application. Unfortunately, this capability is also a liability, especially in the case of encrypted

traffic. Therefore, a hybrid approach that treats the application as a gray box needs to be employed. One promising solution is to tag various packets with the addresses of the memory locations and data fields it causes to change. This sort of capability should be standardized as a feature of application execution, either as instrumentation added by a compiler or as an operating system service.

Another potential criticism of Bloodhound is that it appears to be exploit-specific and therefore doesn't perform as well as vulnerability-specific systems like Vigilante, Shield, or VSEF. These systems make the excellent point that exploit-specific protection is ineffective against polymorphic malware, and as a result, research efforts should concentrate on developing vulnerability-specific and exploit-generic defense mechanisms.

While Bloodhound's primary operational goal is focused on identifying a particular exploit input, its task does not conflict with the goals of exploit-generic defense systems. Instead, the protection that is offered by the self-healing mechanism can be vulnerability-specific, but Bloodhound can provide these systems with more confidence that the fix is correct and blocks *at the very least* the malicious input that triggered their instrumentation. Future work on Bloodhound can look into using the input traffic as a template to generate other semantically correct instances of the flow (this sort of task is exactly what the RolePlayer system is designed for).

4 Design

We divide attacks into several broad categories to make it easier to analyze techniques for identifying flows to replay. The categories allow us to compare techniques based on which categories of attack they will replay.

4.1 Classes of Attack

The simplest attacks are those for which the entire attack is encompassed by a single TCP flow. These are the most common attacks today; typical examples are worms like Slammer or Blaster. The attacker opens a TCP connection, transmits the exploit code, and then closes the connection. The entire flow must be replayed during testing.

In a slightly more complex version, the attack may only be a subset of a larger, innocent flow. Consider an SSH connection where the attacker behaves innocently for several hours and then runs an exploit. The subset of the flow that must be replayed during testing is only a very small portion of the total flow. It is possible to replay the entire flow during testing, but we differentiate this class of attacks from the previous because, for example, it may not be desirable to store all of a long-duration flow if only a portion is suspicious.

An attack may be spread out across multiple TCP flows, each of which taken on its own appears innocent. Consider a hypothetical attack where one TCP flow overflows a buffer and a second flow delivers the remainder of the exploit. An attack of this form could be distributed over arbitrarily many flows, and *all* of

those flows must be replayed during testing. To further complicate this attack, the malicious flows may be distributed over time, with days or weeks between each flow.

An attack may be contained within the timing relationship between multiple flows, or the timing relationship between packets of a single flow. Attacks of this form exploit race conditions in multi-threaded code, and recreating the circumstances of race conditions is notoriously difficult. At the very least, the timing relationship between the packets or flows must be preserved for testing. This requirement presents difficulties for a *rapid* automatic response, as deployment time is constrained by characteristics of the attack – parameters that are under the control of the attacker.

An attack may be polymorphic. That is, the algorithm for generating the exploit code may use cryptographic or other heuristics to change the form of the attack over time. Attacks of this form cannot necessarily be identified by searching a flow or flows for particular bit patterns.

Finally, the attack may depend on innocent user action. Consider an exploit where the adversary sends an attack packet, followed by 100 innocent requests, the last of which triggers the exploit. Any replay of the flows must include the attack packet *and* all 100 innocent requests. This class is particularly difficult because, to the untrained eye, the 100th packet is very suspicious, while the true attack packet does not necessarily stand out. Worst of all, naively testing against that 100th packet (because it truly is an innocent request) is almost always guaranteed to be successful, regardless of the validity of a particular patch.

In the most pathological case, a particular attack may consist of any combination of the above classes of attack. That is, an attack may be polymorphic, spread across multiple TCP flows, *and* depend on some form of user action.

4.2 Basic Design

The simplest version of a Bloodhound-like system records all network traffic and replays the entire archive on demand. We feel this approach is untenable; a system must have finite resources. Therefore, it can only store a finite amount traffic, and replaying or searching an arbitrarily large archive is likely not feasible in a timing-dependent application like self-healing software. In this section, we examine extensions and variations on the basic system with the goal of constructing a viable alternative.

We take as a given that we cannot store all flows. Further, the size of the flow archive must be such that the duration for replaying all flows is reasonable. Here we consider several heuristics for choosing which flows to store.

- Save a sliding window of the last n days worth of traffic. This heuristic has some nice properties. It is simple to implement, and it is simple to tune the size of the archive to optimize storage size or replay duration. To test a patch against the archive, simply replay all stored flows. The obvious down side to

this heuristic is that it fails against attacks that last longer than the n days stored in the archive.

- Save a probabilistic window. That is, rather than using a window with a fixed horizon, probabilistically eject flows from the archive as they age. There are several options here. One option is that as flows age, they are more likely to be ejected from the archive. Another option is to eject an archived flow at random each time a new flow arrives. Regardless of the details chosen, under this heuristic, the self-healing software can only make probabilistic statements about its confidence on a particular patch, based on the likelihood that the entire exploit was contained in the archive tested against.
- Archive those flows identified by a signature-based misuse detector (e.g., Snort). This heuristic has the advantage of simplicity, but it is a poor match for a self-healing system that can patch against new or `0`-day attacks. The self-healing system can patch against never-before-seen attacks, but the testing framework would only be able to replay flows for which there is a pre-existing signature. We examine this heuristic in more detail in the next section.
- Archive those flows identified by a payload-based anomaly detection system such as PayL[24]. This heuristic allows for the detection and storage of suspicious flows that have never been seen before, unlike the previous example. The viability of a heuristic like this depends entirely on the abilities of the anomaly detection system. If the anomaly detection system has a low false negative rate, then the likelihood that the entire exploit package is archived is very high. We examine this heuristic in more detail in the next section.
- A final heuristic that focuses on reducing the duration of testing, rather than reducing storage space, consists of following taint propagation. As a particular piece of software handles each flow, it will modify (or taint) various data structures. If each flow is indexed based on the data structures it taints, then during testing only those flows which taint the data structures involved in the patch under question need to be replayed.

These heuristics may also be combined to achieve various points on the trade-off curve between archive size, replay duration, and confidence. We explore archiving flows based on signature-based misuse detectors and payload-based anomaly detectors further in the following section.

5 Experiments

We used `tcpdump` to collect raw TCP traffic on three machines; a workstation `fae`, a server `sos17`, and a honeypot `sos1`. For each machine, we collected all inbound and outbound traffic for a duration of two weeks. The workstation `fae` is a Red Hat Linux machine under daily usage for word processing, email and instant messaging, surfing the web, and an occasional large file transfer. `sos17` is a Red Hat Linux machine used predominantly as a web and file server, and `sos1` is an OpenBSD machine set up as a honey pot, with no services running.

We separated the first third of each data set out to use as training on those sensors (described later) that required it.

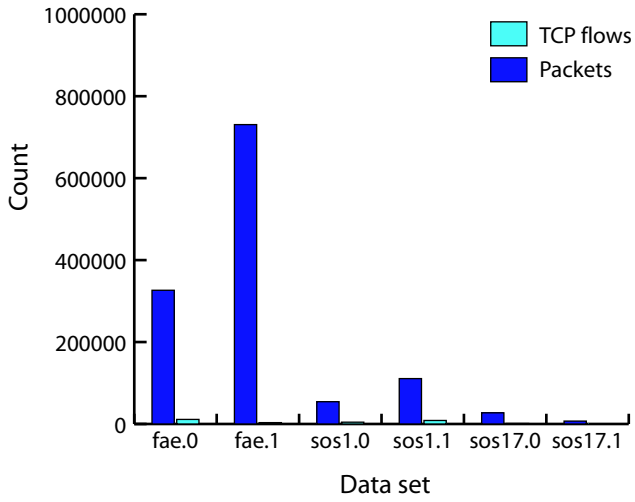

Fig. 1. Comparing TCP flows and packet counts for both training and test data on each machine. `fae.0` and `fae.1` are the training and test data from the workstation; `sos1.0` and `sos1.1` are the training and test data from the honeypot; `sos17.0` and `sos17.1` are the training and test data from the server.

Figure 1 gives a rough profile of each data set. The left-hand columns represent the packet count in the data set; the right-hand columns show how many TCP flows there are in each set. In the `fae.1` data set, for example, there were several large file transfers in the collection period, so the packet count is high, but the TCP flow count is low. Conversely, the `sos17.0` data set shows a web server, where there are only a few packets in each new TCP flow, so the packet and flow counts are closer together.

For each data set, we performed several experiments in order to explore what kind of data-set reduction we could get from signature-based misuse detectors and from payload-based anomaly detectors. For the former, we used Snort, a popular open-source IDS, and for the latter, we used Anagram, a payload-based anomaly detection system.

Figure 2 shows the reduction of the data set after filtering through Snort. Each complete `tcpdump` file was processed through Snort and we retained only those TCP flows that generated alerts. A large percentage of these alerts were ICMP destination-unreachable alerts related to the firewall mechanism on the Linux machines. We further filtered these alerts to give a final working set of suspicious flows. The left-hand bars in Figure 2 represent the raw packet counts from each data set; the middle bars represent all packets which were alerted

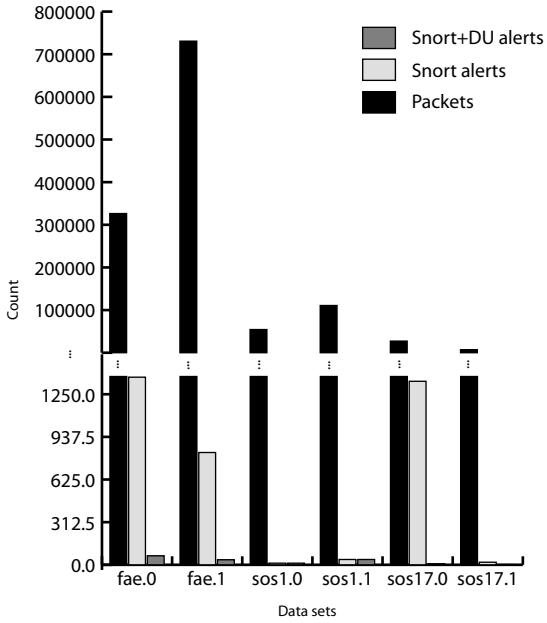

Fig. 2. Data sets after filtering with Snort. On firewalled machines, Snort generates destination unreachable alerts on all messages destined to blocked networks. We filter these messages to further reduce the data set.

by Snort; the right-hand bars are only those alert packets which are not just destination-unreachable alerts. As the graph shows, in all cases, there was a better than 99.9% reduction in the size of the data set.

Figure 3 shows the same data sets when filtered by the payload-based misuse detection system called Anagram. Anagram requires a training period, so the first third of each dataset was used for the training data, while the latter two thirds were used as the test data. The graph in Figure 3 shows that Anagram can reduce the data set by 65%-80%.

Note that, while Anagram is apparently less effective than Snort (99.9% vs. 65% reduction), it is not so clear cut, because Anagram is generating alerts on all suspicious packets, while Snort can only generate alerts on those packets that match known patterns.

6 Evaluation

The major purpose of our experiments was to determine if it was feasible to reduce the amount of traffic that an ARV system would need to store. ARV is essentially a problem of search; we can reduce search time by indexing flows, but indexing for all possible terms isn't feasible and still requires additional storage for the index itself. Performing an online search, even of an indexed corpus,

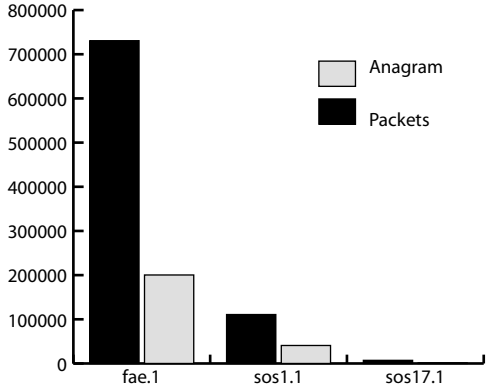

Fig. 3. The three test data sets after filtering by Anagram.

can be sped up if the size of the corpus is reduced. An apparent application of this principle is Google's search: while a large portion of the Web is indexed offline, Google uses the PageRank algorithm to identify the most likely items that would be interesting as the result of a particular search – it does not consider all pages at once. An ARV system like Bloodhound would need to create a "PacketRank" system that only considers packets relevant to the current vulnerability or exploit.

Based on the experimental setup described in Section 5, we obtained fairly good results when using straightforward packet classification schemes (Snort and Anagram[1]). Figure 2 and Figure 3 illustrate the results obtained. In particular, we are able to achieve a reduction in flows of greater than 99% for most of the dumps using Snort and a filtered subset of the Snort alerts. While Anagram does not perform as well, the results are still encouraging. Filtering using Anagram obtains a reduction that ranges from about 60% to 85%. As explained in Section 2, using a misuse-based detector with well-known signatures of old exploits is not optimal for filtering flows that may contain previously unseen exploits. Anomaly-based detectors are better at such a task, and Anagram indicates that they would be useful for ARV.

6.1 Future Work

Our future work on the system focuses largely on the problem of correlating network flows with host-based events. Since we haven't yet implemented the correlation scheme involving marking or indexing flows by what data structures they touch in the application, we could not evaluate its efficacy or performance impact. The purpose of this paper is to outline an important challenge, so such a study is outside of our present scope.

[1] Anagram is an anomaly detection system currently under development by other researchers.

However, our next steps concentrate on mechanisms for marking flows based on which of an application's internal data structures the flow "taints." Designing a systematic and general mechanism (rather than an application-specific, *ad hoc* hack) for this type of synchronous audit logging and tagging is an important challenge. Such a system breaks the normal abstraction between low-level network data and high-level application data objects.

Another area of considerable importance is the creation of a standard suite or benchmark to test self-healing systems against. Such a benchmark does not exist, and most testing performed in the literature is against self-selected or synthesized vulnerabilities and exploits. Finally, we hope that this paper raises a number of questions involving the cross-fertilization of Intrusion Detection with research on search technologies.

7 Acknowledgments

We encountered the problem of automated verification of repairs during the design and construction of a system that is able to self-heal popular network server applications against buffer overflow attacks. We would like to recognize the hard work of Stelios Sidiroglou and Gabriela Cretu, the other developers on that project. In addition, we'd like to thank Felix Wu and Seung-Sun Hong for sharing TCPopera with us. Finally, Ke Wang provided us with Anagram and was very helpful in getting it running.

8 Conclusions

Since many attacks are automated, it appears that defense systems must also include some degree of autonomy. Recent advances in secure systems have led to an emerging interest in self-healing software as a solution to this problem. However, system owners are understandably reluctant to allow automated changes to their environment and applications in response to attacks.

Testing the auto-generated fix to a system is a critical part of raising the confidence level in self-healing systems. One part of this testing is the verification that the micro-patch or changes made by the self-healing mechanism actually defeat the original attack (or close variations thereof).

This paper identifies the important challenge of Automatic Repair Validation (ARV): using audit information to test the resilience and efficacy of a self-healing fix. We propose Bloodhound, a system for recording suspicious network flows and replaying those flows that are related to the exercise of a particular vulnerability. The design process of this system reveals a number of challenging problems that the research community needs to address in order to make automatically self-securing systems a reality.

References

1. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium.*, August 2005.
2. R. Chinchani and E. V. D. Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 284–304, September 2005.
3. M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)*, 2005.
4. W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adatpive Replay of Application Dialog. In *Proceedings of the* 13^{st} *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.
5. S.-S. Hong and S. F. Wu. On Interactive Internet Traffic Replay. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 247–264, September 2005.
6. H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.
7. C. Krugel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 207–226, September 2005.
8. C. Krugel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2002.
9. Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.
10. M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 82–101, September 2005.
11. J. Newsome, D. Brumley, and D. Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{st} *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.
12. J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.
13. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.
14. F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. 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)*, 2005.
15. 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* 20^{th} *Annual Computer Security Applications Conference (ACSAC) 2004*, December 2004.

16. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
17. 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.
18. 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.
19. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
20. A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.
21. A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.
22. T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 274–291, October 2002.
23. H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM*, August 2004.
24. K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.
25. J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.