Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

Life After Self-Healing: Assessing Post-Repair Program Behavior

Michael E. Locasto mlocasto@gmu.edu
Angelos Stavrou astavrou@gmu.edu
Gabriela F. Cretu gcretu@cs.columbia.edu

Technical Report GMU-CS-TR-2008-3

Abstract

One promising technique for defending software systems against vulnerabilities involves the use of self–healing. Such efforts, however, carry a great deal of risk because they largely bypass the cycle of human–driven patching and testing used to vet both vendor and internally developed patches. In particular, it is difficult to predict if a repair will keep the behavior of the system consistent with "normal" behavior. Assuring that post–repair behavior does not deviate from normal behavior is a major challenge to which no satisfactory solutions exist.

We investigate the feasibility of automatically measuring behavioral deviations in software that has undergone a self–healing repair. We provide a first examination of the problem of assessing a repair's impact on execution behavior, and we define a model for representing post–repair behavior using machine–level intraprocedural control flow. In essence, we advocate performing anomaly detection on an application's execution *after* it has been attacked and subsequently repaired. Our goal, however, is not to detect an attack, but rather to provide a tool for assisting a system administrator to perform vulnerability triage. Our system can help them discover the relative impact of a repair so that they can begin to track down and analyze the cause of post–repair execution anomalies.

1 Introduction

This paper presents an initial foray into the field of assessing the behavior of systems that have undergone an automatic repair such as those proposed by various "self–healing" frameworks [1, 2, 3, 4]. While it stands to reason that some of the lessons learned in creating a post-repair behavior evaluation framework can be applied to assessing software patches in general, we explicitly consider the analysis of traditional source or binary patches as out of the scope of this paper. In particular, source and binary patches introduce a level of variability into a system that interferes the underlying structure of the model we employ in this paper. In addition, we focus narrowly on security–related faults, as we expect the "repairs" for this type of fault to be small (relative to larger changes involving the addition of features or other refactoring). Section 5 contains the results of a survey supporting this notion.

Software systems have large, complex codebase that usually contain a number of vulnerabilities. Patching these errors in anticipation of or in response to attacks that exploit them remains one of the primary methods of software defense. Patching, however, is a disruptive activity. Patches have the potential to change system behavior in unanticipated ways. Administrators must therefore thoroughly vet patches before deployment: a time–consuming activity usually involving little formal analysis and a great deal of exhaustive regression testing. Nevertheless, the alternative is less than appealing, as the "patches for patches for patches" problem shows that patching is far from being an exact science.

1.1 Self–Healing

One alternative to traditional binary or source patching may be to adopt and deploy "self–healing" repairs; such systems ultimately envision a process of *online reactive patching*. Given that offline patching and patch testing are still difficult tasks, and that the technical chal-

lenges of actually patching a running system have only recently been researched [5], online reactive patching is an ambitious goal. Even if such systems existed[1], the system administrator has little information to examine whether such a repair perturbs the application and its environment in a non-desirable way before it is deployed[2]. Hence, real-world deployments of self–healing systems have lagged research efforts.

In short, completely automated repairs present both technical and policy difficulties. Leaving aside the technical challenges of achieving a repair or successfully applying a patch to a running system [5], system owners and administrators may hesitate to embrace such technology because of the possibility of making an incorrect, ineffective, or detrimental repair. 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. They therefore need to have some way of measuring the impact of such repairs. Our goal in this paper is to provide a tool for measuring this impact; we leave the construction of automated root cause analysis for future work.

1.2 Contributions

This paper explores the feasibility of a technique that ensures that the behavior of an application after a repair is applied matches a profile of normal behavior: we propose *post-attack* anomaly detection to complement *pre-attack* anomaly detection. While anomaly detection is normally applied to a system to detect exploit attempts in progress or new behavior of an already compromised process or host, the application of anomaly detection post-attack can provide insight into four different types of scenarios:

1. new emergent behavior (*e.g.,* a flash crowd)
2. legal behavioral updates (*e.g.,* software patches)
3. failure of a repair (malcode still in control)
4. *successful repair (execution consistent with normal behavior)*

It is this last scenario that we are concerned with in this paper.

Most previous work in self–healing has focused on the feasibility of the core mechanisms (*i.e.,* what actions to take to attempt a repair). Relatively little work has been done to understand the process of automatically validating that the repair was ultimately successful. The meaning of success can range from correctly stopping future exploit instances to not destabilizing the software application. We investigate whether a repair perturbs the execution of a process such that it experiences radically different behavior after a fault or exploit attempt. Briefly, we focus on the "execution tail": the control flow paths that follow the exercise of a repair. We derive the main features of the execution tail model from both intra-procedural and inter-procedural control flow artifacts. Execution tails provide a foundation for measuring the subsequent behavior of a software program.

We define the execution tail model (discussed in Section 2) for capturing the behavior of a system after a repair has been applied. This model of machine–level control flow complements function or system call level behavior models (the level of granularity is drastically different). We created *Calypso*, a system that computes and compares these types of behavior profiles. The concepts underlying Calypso are agnostic to the particular repair mechanism in use (it looks at the behavior of the system, not the content of the repair). As a result, it provides a way to compare behavioral impact across different self–healing techniques.

Calypso operates in a fashion similar to existing dataflow analysis tools [6, 7, 8]: it intercepts machine instructions to record the branch target addresses (along with other information) that the program has encountered post–repair. In Section 4, we use our system to observe the behavior of a program after the repair of a real vulnerability. We also observe its behavior on a published testbed of synthetic vulnerabilities [9].

1.3 Limitations

Our approach can be considered a form of whitebox validation. Our approach fundamentally differs from blackbox approaches that only observe if the application has crashed, or if the application has returned a similar status code to past successful requests (*e.g.,* 200 OK). While our tools do not require source code access, they do require the ability to supervise the system's instruction stream. Blackbox approaches have the advantage of being less disruptive by only looking at I/O. Consequently, they can also be less precise. We expect that system adminstrators will have an evironment to automatically dispatch a run of a supervised, post–repair copy of an application (perhaps in an infrastructure that hosts multiple disposable virtual machines, one per detected attack).

We assume that most self–healing systems do not generate fixes in an adversarial fashion; that is, the legal behavioral changes induced by these fixes only tweak small

[1] Of course, no general procedure exists that can generate correct fixes for arbitrary faults, vulnerabilities, or other system problems, but we can repair certain known classes of vulnerabilities.

[2] A self–healing system *could* "pause" after generating a repair and allow the administrator to examine it. Attacks, however, occur at machine speed, and pausing for the human decision–making process should be minimized.

amounts of state or control flow and do not vastly perturb the behavior of the application. Many security–related patches support availiability by refusing to process input aimed at subverting it. Given that such behavior is similar to existing program code, most patches (and the self–healing fixes proposed in the research community) simply redirect execution along existing control paths (for example, calling an internal logging routine and exiting the current function with an error code). It is, however, conceivable for repairs to drastically change the behavior of the program, making it difficult to differentiate between normal post–repair behavior and anomalous post–repair behavior. We consider the problem of efficiently *retraining* anomaly sensors in response to sanctioned or legal behavioral changes in other work [10].

As we mention above, we do not attempt to address a switch to a new version of an application or large, widespread changes in the program text. We also refrain from dealing with traditional patches (although our evaluation compares the post–repair behavior of a library to later versions of the library that contain a patch for the vulnerability in question). Patches present at least two challenges that the self–healing techniques we examine do not. First, patches for proprietary[3] COTS systems often come in binary form that (1) combines multiple patches into one binary delta and (2) is often obfuscated to make it harder for blackhats or competitors to reverse–engineer the vulnerability from the patch [11]. Second, our model relies on the address values of program components (*i.e.,* basic blocks) remaining consistent across training and testing runs. Patching (and the subsequent recompilation), however, introduces a different layout of virtual addresses, thus adding a large amount of noise to the "testing" run: new, valid control flow transfer sequences occur that represent legal (but now de-synchronized) updates to existing parts of the model derived from the training data.

Automatically detecting attacks and automatically repairing attacks can lead to an infrastructure capable of autonomously managing its own defense posture, but such a capability is dangeous without some automated repair validation mechanism. We note that automatic repair validation is a fairly large problem with many different aspects (one of which we study here). In particular, it is important to test whether the self–healing fix actually stops the input or exploit data that caused the repair process to initiate. Furthermore, a complete repair validation system can check that the repair blocks similar instances of the initial exploit [6, 12, 13] or all exploits that exercise the underlying vulnerability.

[3]Even most mainstream open-source "patches" are distributed as updated versions of the program or library binary.

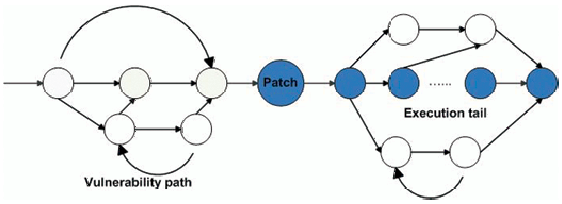

Figure 1: *Vulnerability path, patched code point, and execution tail sequence.* A vulnerability path is a sequence of machine address pairs that lead to the exercise of a particular vulnerability, the "patch point" is the set of locations in the code where the effects of the vulnerability are dealt with, and the execution tail is the sequence of address pairs that express post–repair behavior.

2 Approach: Execution Tail Model

One common approach for defining program behavior involves sequences of system calls [14, 15]. Because system calls represent the method by which processes affect the external world, these sequences are thought to provide the most tangible notion of system behavior. These models, however, are susceptible to mimicry attacks [16]; an attacker can keep the system within some delta of the "normal" patterns while executing calls of their choosing. This problem suggests that we require a more fine-grained notion of program activity[4].

Capturing program behavior in more detail than system call sequences used to be a challenging task; recent advances in binary instrumentation tools for x86 (*e.g.,* Valgrind [17], Pin [18]) have eased the burden significantly. As a result, we can capture a much finer-grained model of program control flow than system call sequences.

We are interested in observing the behavior of a program after it has exercised an area of code that was subject to a self–healing repair. This *tail* of execution represents the post–repair behavior — behavior that might be incorrect because of a broken patch or behavior that might remain under the influence of the attacker. In essence, we are proposing a form of anomaly detection by comparing these observed "execution tail" samples with a model of "normal" execution for that point in control flow.

Our insight is that for most use cases, the most common feature that patch testers and system administrators are interested in is the sequence of code that will be executed following the exercise of a fix. Essentially, a repair

[4]Note that our goal is not to make Calypso resistant to mimicry attacks or to criticize system call approaches for being susceptible to them; instead, the lesson we should learn is that relatively large amounts of work can happen "between" system calls, and it is the more precise nature of this activity that can help inform our models of program behavior.

should allow the system to deal with malicious or malformed input and continue behaving normally. Therefore, the system should largely obey the same processing path, or one of a few different default processing paths (note that we are not attempting to differentiate the semantic meaning of various paths).

```
...
tail(13) 0x804a987 -> 0x804a95f png_calculate_crc
tail(14) 0x8062b3c -> 0x8062bdd crc32
tail(15) 0x8062be0 -> 0x8062b44 crc32
tail(16) 0x8062bff -> 0x8062bec crc32
tail(17) 0x8062bff -> 0x8062bec crc32
tail(18) 0x8062bff -> 0x8062bec crc32
tail(19) 0x805b543 -> 0x805b564 png_crc_finish
tail(20) 0x805b4cc -> 0x805b514 png_crc_error
...
```

Figure 2: *Example Execution Tail From* `libpng`. This tail has a loop expressed in entries 16 to 18: control flow jumps backward from `2bff` to `2bec`, then flows forward via straight-line execution to the jump at `2bff`.

In their simplest form, execution tails are sequences containing some number of pairs of machine addresses. These pairs express a series of control flow transfers. The first machine address in the pair denotes the instruction initiating the control transfer or branch (*e.g.,* the address of a CALL instruction). The second address denotes the branch target addresses. We focus specifically on the class of branching instructions (*e.g.,* JMP, JNZ, CALL) because between branch targets or basic block entrances, control flow proceeds in sequential order. This sequential control flow is implied by the arrival at a particular branch target (see Figure 2). An execution tail is a sequence of pairs of addresses (a_j, a_k) expressing a change in control flow from instruction j to instruction k. A conjunction over these pairs expresses a particular path of dynamic control flow taken by the application in response to some input data. We believe that this feature (along with some context information) expresses the essence of application behavior.

Assume that the operation of a self–healing repair begins at some machine instruction i. The execution tail starting at instruction i is a sequence:

$$T_i = \{(a_i, a_j), (a_j, a_k), \cdots, (a_m, a_n)\} \qquad (1)$$

The set of all execution tails in a program is the union of all T_i for each instruction i in the program.

We construct a similarity measure using the distance between the test sample execution tail (the execution tail observed by our system after a patch is applied and exercised) and each normal execution tail relevant to that point. That is, we expect the behavior profile to contain a number of (similar) execution tails, or clusters of such tails. Measuring the distance between the sample and each normal tail or cluster of tails can provide an indication of the repair's fidelity to the original behavior.

Some patch or self–healing repair P applied to a program produces a sample execution tail t_{sample} for testing. Given that a program behavior model expresses N execution tail behaviors (where each behavior component could be a single execution tail training sample or a summary of a cluster of execution tail samples), the behavioral impact measure is the sum of the difference between these N behaviors and the test sample execution tail. The measure can be computed by either a weighted or unweighted sum of the differences between each behavior model component c_i and t_{sample}. We consider the unweighted option below.

$$\text{DISTANCE}(t_{sample}) = \sum_{i=1}^{N} \text{DIFF}(c_i, t_{sample}) \qquad (2)$$

where DIFF is a procedure that provides a numeric score of the distance between the representations of each relevant behavior profile component and t_{sample}. This procedure depends on the particular model in use. In our system, we use three measures for the string–based model: string equivalence, longest common substring, and longest common subsequence. For the 2-gram model, we employ Manhattan Distance between the gram positions. These measures are described more fully in Section 3.

Larger values of DISTANCE indicate less of an execution anomaly; the repair keeps the difference between post–healing behavior and the components of normal behavior small. While abnormal behavior may actually be correct, the score can help the system owner or administrator prioritize patch investigations. A sufficiently large difference may even help the system automatically roll back the patch or temporarily employ some other defensive strategy like data patches [13], vulnerability–driven execution filters [6], or Shield filters [19].

2.1 Augmenting the Model

Our model captures and measures a single important feature in determining the normality of post–patch application behavior. We acknowledge that our model may not be comprehensive enough for any imaginable use case. Organizations interested in automatically testing and validating post–repair behavior might be interested in certain aspects of application behavior that we have not considered. We believe, however, that the basic model is serviceable enough given that we are the first to look at trying to quantify this problem for a live repair system. The model is extensible; defining additional features means making Calypso capture that feature as part of its train-

ing mode[5] and modifying DIFF to take that additional feature into account. As we mention above, additional features may include context like the state of the CPU, some user token, operating system state, or file system state.

We could weight each component of the execution tail behavior model. Since some execution tails may be more frequent than others, it is more likely that these tails are "normal" behavior for the system, and thus deviation from them should be treated as a greater distance than deviation from less frequent component behaviors. We plan to explore this extension in future work.

Finally, it may be possible to use basic data compression techniques to provide a baseline indication of how similar two execution tails are. Compression naturally identifies redundancy in data and would therefore expose salient features of an execution tail. In the near future, we intend to compare our distance measure proposed above with the output of a COTS compression mechanism.

3 Implementation

Calypso operates in two modes: tail collection and tail comparison. The execution tail contains entries that record both the source and destination branch addresses as well as the name (or callsite address) of the current routine.

Calypso utilizes the Pin [18] dynamic binary rewriting framework to intercept the execution of a process and insert some basic repair instrumentation. Calypso takes advantage of Pin's ability to collect machine–level information, including branch target addresses and function callsites, to help characterize the process's control flow behavior.

Calypso intercepts each instruction, decide if the instruction was a branching instruction, and assess if the branch was actually taken. If the branch was taken, then Calypso records the source and destination addresses. This profile serves as a model of normality that we use during the tail comparison phase. The tail comparison phase starts after a self–healing repair has been applied to the application.

Since profile storage requirements are a concern, and string-based comparisons have limited tolerance for slight variations, we employ a statistical summary of the content of an execution tail as an alternative method of storing and comparing the information in the profile execution tails with the test execution tail. We collect sequences of address pairs and treat them as 2-grams (*source address*, *destination address*) and keep track of the frequency of these 2-grams for the execution tail. We calculate the difference between two models using the Manhattan Distance (the sum of the difference between the frequency at each corresponding gram).

The downside of this approach is that it loses sequence information and can miscompare tails, as the 2-gram content may change if the application is recompiled. For a strictly self–healing approach in which no code is changed (just execution artifacts like return values), we do not anticipate that such a change would cause a problem, as machine addresses will not have changed. For traditional patches, which can alter addresses, we plan to investigate other fast trace analysis techniques as well as approaches such as encoding control flow graph structure (rather than labels) with longer n-grams.

In tail collection mode, Calypso accumulates a record of execution tails from different points in program execution. This profile serves as a model of normality that we use during the tail comparison phase. There are some practical barriers to storing and loading this profile; we discuss them in Section 3.2. Calypso only enters the comparison phase if the patched or repaired code is exercised.

3.1 Execution Tail Comparison

The tail comparison phase starts after a patch or a runtime self–healing repair has been applied to the application. We use several comparison functions for sequences of address pairs. We implemented four comparison strategies, three of which are used for execution tails represented as strings and one that we use for comparing 2-gram models. For comparing strings, we use: string equivalence, longest common substring (LCS), and longest common subsequence (LCSeq). For comparing 2-gram models, we use the Manhattan Distance between the 2-gram features.

Substring Comparison Comparisons based on string equivalence, while easy to implement, are unsatisfactory because the comparison must have some amount of tolerance for normal behavior artifacts. For example, invocation of a system call might time out and be repeated. These types of artifacts manifest as slight variations in the execution tail, making straight string comparisons ineffective. We implemented LCS and LCSeq in PatchEpilogue and use them to compare the model with the sample execution tail.

n-gram Comparison Since profile storage requirements are a concern, we employ a statistical summary of the content of an execution tail as an alternative method of storing and comparing the information in the profile execution tails with the test execution tail. We collect

[5] For our implementation, additional features must be something Pin can observe or generate from observed information. Given that Pin can see many machine–level events, this condition is far from a restriction.

sequences of address pairs and treat them as 2-grams (*source address*, *destination address*) and keep track of the frequency of these 2-grams for the execution tail. This process helps merge and cluster execution tails. We calculate the difference between two models using the Manhattan Distance (the sum of the absolute value of the difference between the frequency at each corresponding gram).

3.2 Execution Tail Database

While collecting and storing large amounts of execution tail traces is relatively straightforward, we are faced with a dilemma during the comparison phase. Since we cannot anticipate when and where (*i.e.,* in what program location) a program will be patched or repaired, we do not know in advance which execution tails need to be tested against. Therefore, PatchEpilogue, when it starts up to supervise a software application, would need to read in the entire profile database because it cannot anticipate which parts of the profile it might need.

Loading such a large collection of execution tails at application startup is undesirable. First, these tails may never be used if the process is never attacked. Second, not all the tails are relevant to locations in the code (*i.e.,* vulnerabilities) that will be exercised. Third, the potentially large number of tails (depending on the depth of training) could soak up large amounts of primary memory, leaving little room for Pin to dynamically recompile the application and even less room for the application's native memory needs. Finally, it could take a potentially long time to read in the entire database.

In order to avoid this startup cost and wasted memory space, we created a PHP/MySQL web service whereby PatchEpilogue feeds execution tails acquired during training to a database for storage. When an execution tail is needed during the post–repair comparison phase, PatchEpilogue requests the set of execution tails associated with the current point in control flow. At this point, our system can proceed with the comparisons above. Having this centrally accessible service reduces the memory requirements and startup times of any particular application supervised by PatchEpilogue.

3.3 Limitations

One of our major assumptions is that the ngram model values do not change. The self–healing repairs we examine modify critical data items: they do not replace or rewrite code or instruction sequences (and thereby cause a recomputation of virtual addresses).

We do not currently impose a maximum length on an execution tail: PatchEpilogue stores all tail components it encounters. Calibrating this limit on a per-application basis is the subject of some of our ongoing efforts. We do, however, limit PatchEpilogue to observing only a few execution tails at any particular time, as we do not expect a system to handle multiple separate vulnerabilities *at the same time*. Even though we expect to repair or patch the application over time, we do not expect attackers to simultaneously exploit more than a few new and different vulnerabilities (although in an operational environment, it would be of interest to discover how well this procedure scales in terms of handling sustained and simultaneous attacks).

4 Evaluation

Our evaluation is deep rather than broad; we focus narrowly on investigating the effects that our system observes for a small set of "self–healing" cases and vulnerabilities.

Note that Calypso is not meant to demonstrate the ultimate correctness of code patches. As we explain in Section 1, such analysis is a far more difficult problem than measuring behavioral deviations. Labeling an execution trace resulting from a patch as "correct" or "incorrect" assumes a much deeper knowledge of program semantics that is difficult to achieve with even extensive human supervision and analysis. To that end, our evaluation is not designed to examine a mingled collection of some patches that work correctly and patches that contain bugs and distinguish between the two with 100% accuracy. Such a result would serve as a dramatic step forward for the field. Instead, Calypso provides a foundation for that type of research by offering one way to observe behavior deviations of an unmodified program binary as it undergoes a self–healing repair. This distinction is crucial to understanding the scope of the current work: we are not promising a decision procedure that distinguishes between arbitrary non-trivial properties of two programs.

We are interested in evaluating the process of capturing and comparing a post–patch execution tail with a database or profile of "normal" execution tails. In order to do so, we first gain an understanding of how PatchEpilogue models of the same application compare against each other as well as against a system-call based record of execution (using `strace`). Figure 3 shows the experiment space: of particular interest are comparisons between branch (3) and branch (7). For each vulnerable or patched version of an application, we observe the behavior using a sensor (either PatchEpilogue or strace) under some input (either benign or malicious).

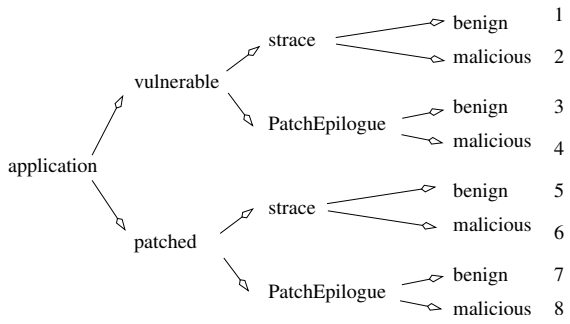

Figure 3: *Experimental Combinations.* For each application we test, there are three variables to examine: the version of the application (vulnerable or patched), what sensor we use to observe execution behavior (strace or PatchEpilogue), and what type of input we provide the program (benign or malicious). The main purpose of this paper is to provide a metric for assessing how well cases (7) and (8) compare to case (3). In particular, a BIM representing a large difference between (7) and (3) indicates that the patch or repair warrants further examination. We also expect the difference between (8) and (4) to be fairly large in cases where a successful exploit expressed in case (4) begins to execute an attacker's code.

4.1 Baseline Measures

We wanted to calibrate our expectations about the sensitivity and resilience of PatchEpilogue as a sensor. Accounting for inherent perturbations in the model can aid us in comparing a post–patch execution tail with the normal profile of pre–patch execution. We do so in two ways. First, we analyze how much variation PatchEpilogue itself introduces for normal (consumption of benign input) runs of the same application. Second, as a control for this measure, we employ the `strace` tool to collect higher–level control flow information of multiple runs of the application on the same input. Table 1 presents the results of the comparisons described below.

As mentioned in Section 3, we compare execution tails using four comparison strategies. When we treat execution tails as strings, we use equivalence, LCS, or LCSeq. Otherwise, when using the 2-gram representation, we employ Manhattan Distance. From these baseline experiments, we learn that LCS and LCSeq are much too expensive in both time and storage requirements to use on profiles of more than a few MB. The `strace` tool does not provide an output that can be structured to easily quantify the difference between two runs. We *diff* two runs and compute the percentage of system calls that are different (both the arguments of the system calls as well as the return values can differ).

For this baseline evaluation we considered a variety of applications, ranging from sorting, hashing, and compression programs to a Web server. For each program, we generated two profiles each using PatchEpilogue and strace. We do not vary the input across runs of the same application. We used the following set of input (so that our results can be easily replicated): date - N/A, echo - "Hello World!", gzip - *httpd-2.2.8.tar*, gunzip - *httpd-2.2.8.tar.gz*, httpd -GET localhost HTTP/1.0, libpng *toucan.png*, md5sum *httpd-2.2.8.tar*, sha1sum - *httpd-2.2.8.tar*, sort - *httpd.conf* (the unmodified config file for httpd-2.2.8). Given that we assess `libpng` in Section 4.3, we also observe its operation on a benign image file for both pre and post patched versions.

4.2 Wilander Testbed

The Wilander Testbed [9] consists of a variety of buffer overflows that attack the process memory space through manipulation of stack data and function pointers; it is meant to evaluate the efficacy of protection mechanisms like StackGuard and Propolice. Our purpose in using this testbed is to demonstrate the basics of creating post–repair execution tails for real vulnerability types. We selected six of the 18 cases to demonstrate the feasibility of capturing and comparing post–repair execution tails. We run these test cases in PatchEpilogue with a basic self–healing fix that automatically repairs the corrupted stack frame. We observe the behavior of these test cases in three situations:

1. without any repair or attack to acquire the "normal" profile
2. in the presence of an attack, but without any repair
3. in the presence of an attack, but with a repair

We first verified that these cases could be exploited; we marked the testbed executable as needing an executable stack and switched off address space randomization on our experiment environment. All six cases were successfully exploited; they drop a shell. We then observe the standard execution of the test cases. Next, we observe the attack profile with no automated repair in place. Finally, PatchEpilogue observes the test case's behavior after PatchEpilogue enacts the repair.

The testbed is somewhat of a special case in that it is designed to exploit a series of built-in vulnerabilities. The only benign execution behavior is when the software prints out a usage and help message. Nevertheless, a successful exploit involves forking a shell whereas the benign case involves screen output and program termination. With the addition of our self–healing mechanism, the testbed can gracefully recover from the overwrites. Doing so results in a post–patch behavior profile that we compare to the unprotected case and the benign case, as shown in Table 2.

Table 1: *Survey of Execution Tail Variation.* The table lists the BIM value for the LCS and LCSeq string models and the Manhattan Distance for the 2-gram model. We can see that, in our calibration runs, most behavior measurements remain identical or near–identical: a result we expect from running the same program on the same input. Some variation, however, must be expected — and this variation is present in measurements made by a third–party tool, `strace`, as well as PatchEpilogue.

Application	SE	LCS BIM	LCSeq BIM	ManDis BIM	`strace` diff
date	identical	∞	∞	∞	25.64%
echo	identical	∞	∞	∞	36.36%
gzip	identical	∞	∞	∞	0.65%
gunzip	identical	∞	∞	∞	0.93%
httpd	different	N/A	N/A	4.29e-8	35.66%
libpng$_{pre}$	different	N/A	N/A	0.05	36.28%
libpng$_{post}$	different	N/A	N/A	0.02	20.72%
md5sum	identical	∞	∞	∞	0.18%
sha1sum	identical	∞	∞	∞	0.18%
sort	identical	∞	∞	∞	5.07%

Table 2: *Comparing a Wilander Test Case.* Because our self–healing mechanism only repairs execution when an attack occurs, no case (7) exists; this situation differs from that of a traditional patch as shown in Table 3. Here, we are interested in how the post–repair execution matches the profile collected during the exercise of the vulnerability (4) as well as a benign execution trace (3).

Scenario	Manhattan Distance	BIM
(8,4)	6491	1.54e-4
(8,3)	8093	1.24e-4

4.3 The `libpng` Vulnerability

Older versions of `libpng` contain a major vulnerability [20]. We assessed the behavior of the PNG library both before and after we applied a patch for this vulnerability. We linked a small image viewer against a vulnerable version of the library (1.2.5) and as well as a patched version and two later versions (1.2.6 and 1.2.8). We ran the various versions of this image viewer in PatchEpilogue on both a benign PNG image as well as an image designed to trigger the vulnerability. The patch for this issue logs an error message and halts execution. Using the 2-gram model approach, we calculated the Manhattan Distance and BIM for the four relationships shown in Table 3.

Interestingly, the profiles shown in Table 3 for case (8) and case (4) are similar because the actual exploit is a proof of concept that crashes the program shortly after the exploit succeeds. Likewise, the patch, when exercised by malicious input in case (8), brings execution to an end by printing an error message and version information and then cleanly terminating. Thus, these two models have a common preamble that dominates most of their 2-gram model. In the case where the exploit was attack code that began to download a Trojan or engage in other nefarious activity, we expect the models to differ sharply. We observe that the "patched" normal behavior (expressed in (7)) is closer to (3) (normal behavior) than (8) is. The patched normal behavior is also closer to (3) than it is to (8).

Table 3: *Comparing libpng-1.2.5 with libpng-patched.*

Scenario	Manhattan Distance	BIM
(8,4)	2,722	3.67e-4
(8,3)	475,688	2.10e-6
(7,3)	*386,996*	*2.58e-6*
(7,8)	476,702	2.09e-6

It would be useful to have some sense of scale for the BIM, even though our primary purpose is to achieve a rank ordering rather than a strictly relative scale. To help accomplish this, we wanted to see what the BIM would be for later versions of the library. To this end, we retested using versions 1.2.6 and 1.2.8 of `libpng` rather than a hand–patched 1.2.5. Our results appear in Table 4.

5 Discussion

A patch can affect a behavioral model by changing control and data flow. We performed a review of some security–related patches; the collection that we studied

Table 4: *Comparing libpng-1.2.6 and libpng-1.2.8.* We can see that the behavior diverges even more, presumably because these library versions included changes and patches beyond the patch for the vulnerability in 1.2.5. Compare the BIM relationship between case (3) and (7) here with that in Table 3.

Version	Scenario	Manhattan Distance	BIM
1.2.8	(8,4)	2986	3.34e-4
	(8,3)	475,798	2.10e-6
	(7,3)	*626,449*	*1.59e-6*
	(7,8)	484,446	2.06e-6
1.2.6	(8,4)	2973	3.3e-4
	(8,3)	475,789	2.10e-6
	(7,3)	*625,728*	*1.59e-6*
	(7,8)	460,397	2.17e-6

cause only small perturbations in program behavior. As a result, we can expect that security–related patches should cause only small, localized changes in control flow. It is precisely this "local" information that execution tails encode.

Briefly, we observe that the goal of many security–related patches is to support the availability of the system by refusing to process input aimed at subverting it. Given that such behavior is similar to existing program code, most patches simply redirect execution along one of these paths (for example, calling an internal logging routine and exiting the current function with an error code). It is, however, possible for patches to drastically change the behavior of the program, making it difficult to differentiate between normal post–patch behavior and anomalous post–patch behavior. This paper provides a model and mechanism for detecting such changes, whether they are sanctioned or not. The current state of the art consists of manually–driven testing of patched versions of the application in simulated environments with little or no widely–accepted measures of behavioral deviation.

5.1 Security Patch Survey

Security–related patches cause only small perturbations in program behavior. As a result, we can expect that security–related patches should cause only small, localized changes in control flow. It is precisely this "local" information that execution tails encode.

A patch can affect a behavioral model by changing either or both the control and data flow. Examples of changes in control flow include updating, removing, or introducing new decision control structures; introducing a new child function; or inserting a new parent function (*e.g.,* a sanity check on input parameters). Changes in data flow include adding new variables or symbolic values; adding or removing arguments or function parameters; and modifications to the set of possible return values. We note that our examination in this section is strictly static: it does not attempt to execute the patches or otherwise determine if the code contained in them is ever actually executed. In addition, we make no distinction between macros and function calls. We also do not investigate changes made to global state as part of newly introduced functions.

Table 5 lists our results for a variety of applications, including stunnel [21], some web servers [22, 23, 24], linux [25], cvs [26] and fetchmail [27], as well as various vulnerabilities in libpng [20], Firefox [28], and Samba [29]. Most of the control flow changes we observed result from invocations of new functions as well as the insertion of new `if` statements or updates of `if` conditions. Most data flow changes involve new arguments to function calls, or new ways of wrapping those arguments, as well as new `return` statements that introduce new values. A majority of the patches we examined made very minor changes; for example, the patch to ghttpd substitutes the use of a "safe" library function and derives the value of a new argument for that call. The patch for nullhttpd introduces a new `if` statement and condition with a call to an application function to log an error (presumably, the dynamic behavior also involves the invocation of the library `printf()` family of functions and the `write()` system call).

6 Related Work

Although patches are typically vetted offline before they are applied to production systems, the problem of automating such a task is currently unaddressed in the research literature, mostly because the very concept of online patching or software self–healing is a relatively new research area. Most existing systems only consider rough tests of survivability or "liveness" (*e.g.,* does the machine respond to an ICMP request) and whether the application has not crashed. The closest work to ours is the paper on Delta Execution [30]. In contrast to this work, we focus exclusively on the self-healing repair validation problem, and we employ a form of anomaly detection rather than *partial replication*.

Other efforts focus on identifying, measuring, and controlling the path leading up to a vulnerability (although recent work has also examined how to use patches to drive the automatic creation of exploits [11]). In particular, efforts at dynamic taint analysis [7, 31] and vulnerability–specific alerts [12] keep track of the se-

Table 5: *Survey of Patches*. We list the vulnerable version of an application, the size of a patch in lines (including comments), and the changes in data and control flow introduced by the patch, as listed above. The magnitude of the difference between the changes and the application's total size supports the notion that patches introduce relatively confined model updates.

Application	Patch Size (lines)	control flow Δ	data flow Δ
Linux-2.4.19	20	3	1
ghttpd-1.4	16	4	5
nullhttpd-0.5.0	12	2	1
stunnel-3.21	29	0	3
libpng-1.2.5	98	10	12
cvs-1.11.15	81	1	2
Apache-1.3.24	11	0	1
fetchmail-6.2.0	183	1	5
Samba (CVE-2004-0882)	65	0	7
Samba (CVE-2004-0930)	386	99	39
Firefox-2.0.0.3	22	8	0

quence(s) of instructions that propagate taint and eventually lead to the exercise of a vulnerability. Our execution tail model mirrors these vulnerability descriptions: whereas the pre-patch control flow path models the vulnerability, the post-patch control flow defines the execution tail (these systems concern themselves only with detection, not with post–detection behavior or automated repair). Similarly, whereas anomaly-based intrusion detection [32, 33] seeks to identify anomalies that predict the presence of an attack, Calypso seeks to identify deviations from normal behavior *after* a defense to an attack has been enacted.

Execution trace analysis and measuring code similarity and execution features is a resurgent area of work. In particular, Ganapathy *et al.* [34] describe a procedure for fingerprinting source–level artifacts to identify similar source code locations that might be missing an authorization check. While profiling application behavior at the system call level [14, 35, 15, 32, 36, 37] is a somewhat saturated field, other work considers how to extract behavioral features of arbitrary malcode samples [38], and very recent work provides a way for extracting security–related properties from source code by correlating known security checks with the contexts they are used in to detect contexts that are missing such a check. In contrast, Calypso does not aim to understand the semantics of an arbitrary piece of malware, nor do we perform symbolic execution or program slicing. Instead, our system is meant to address a deceptively simple task: measuring the similarity of two execution traces to quantify the impact of the repair mechanism that produced one of the traces.

6.1 Self–Healing

Software self-healing and survivable systems is an active area of research. Rinard *et al.* [39] developed compiler extensions that 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). They employ this system for *failure oblivious computing* to execute through such faults [3]. The Reactive Immune System [1] aims at roughly the same concept: process execution can be forced through a fault or exploited vulnerability by "slicing off" the corrupted function and returning an error code. Demsky [40] discusses mechanisms for detecting corrupted data structures and fixing them to match specified constraints. The pH system [41], while not strictly self–healing, is an active response mechanism that frustrates an attacker by using system call interposition to slow down an attacker's code. This system negotiates a fine line between providing semantically correct continued execution and making the target less attractive to an attacker.

Instead of attempting to force execution through an exploited vulnerability, a significant body of work attempts to rewind execution to a pre-fault or otherwise uncorrupted state [42, 43, 44]. In particular, the Rx system [2] checkpoints an application in anticipation of errors, faults, or attacks. Rx uses a series of heuristics to explore safe alterations of program state. If Rx finds such a semantically safe change, then execution proceeds. If not, then the system falls back to crashing. PatchEpilogue focuses on creating a measure of post–repair similarity to validate the operation of these self–healing systems and can be applied to both "rewind" and "execute through" approaches.

ASSURE [45] attempts to minimize the likelihood of

a semantically incorrect response to an attack by using *error virtualization rescue points*. A rescue point is an existing program location that is known — or at least conjectured — to successfully propagate errors and recover execution. The key insight is that a program will respond to malformed input differently than legal input; during an offline training phase, the system learns about locations in the code that successfully handle these sorts of anticipated input "faults." These locations serve as 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 (rather than statically determined by a programmer).

7 Summary

The limits of detection technology have historically mandated that researchers address the shortcomings of intrusion detection before automated repair mechanisms are considered — an attack must be detected before a response can be mounted. As computing infrastructures increase in size and complexity, the use of abstraction as a design principle begins to interfere with the ability to rapidly diagnose and repair errors and exploited vulnerabilities. Providing continued availability of key services demands an active response mechanism. Unfortunately, such an active response or self–healing mechanism introduces the possibility for further and greater instability in the system.

In this paper, we focus on creating a system that can assist administrators as they begin to analyze the effects that automated repair might have on their systems. In essence, we advocate performing anomaly detection on *post-attack* program behavior. Our behavior model is based on a simple, space-efficient n-gram representation of intra-procedural control flow. Our analysis of our system's operation on a testbed of synthetic vulnerabilities and a real vulnerability indicates that such control flow is a sensitive and useful feature for observing behavior deviations due to self–healing repairs that modify program state (as opposed to patches that modify program code locations). Our analysis also indicates that a better model should preserve more of the meta-structure of the control flow, perhaps by using some existing trace analysis [46] techniques.

References

[1] 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*, April 2005, pp. 149–161.

[2] 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.

[3] 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.

[4] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis, "From STEM to SEAD: Speculative Execution for Automatic Defense," in *Proceedings of the USENIX Annual Technical Conference*, June 2007, pp. 219–232.

[5] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly," in *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[6] J. Newsome, D. Brumley, and D. Song, "Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software." in *Proceedings of the 13^{th} Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[7] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the 12^{th} Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[8] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-flow Integrity," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[9] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2003.

[10] M. E. Locasto, G. F. Cretu, S. Hershkop, and A. Stavrou, "Post-Patch Retraining for Host-Based Anomaly Detection," Columbia University, Tech. Rep. CUCS-035-07, 2007.

[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[12] 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.

[13] W. Cui, M. Peinado, H. J. Wang, and M. 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.

[14] S. A. Hofmeyr, A. Somayaji, and S. Forrest, "Intrusion Detection System Using Sequences of System Calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[15] H. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[16] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2002.

[17] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, June 2007.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[19] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," in *Proceedings of the ACM SIGCOMM*, August 2004.

[20] http://www.us-cert.gov/cas/techalerts/TA04-217A.html.

[21] http://www.securityfocus.com/bid/3748.

[22] http://www.securityfocus.com/bid/5960.

[23] http://www.securityfocus.com/bid/5774.

[24] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747.

[25] http://www.sfu.ca/~siegert/linux-security/msg00047.html.

[26] http://www.us-cert.gov/cas/techalerts/TA04-147A.html.

[27] http://fetchmail.berlios.de/fetchmail-SA-2005-01.txt.

[28] http://www.mozilla.org/projects/security/known-vulnerabilities.html.

[29] http://us4.samba.org/samba/history/security.html.

[30] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek, "Delta Execution for Software Reliability," in *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

[31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution Via Dynamic Information Flow Tracking," *SIGOPS Oper. Syst. Rev.*, vol. 38, no. 5, pp. 85–96, 2004.

[32] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-Sensitive Intrusion Detection," in *Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[33] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting Execution Context for the Detection of Anomalous System Calls," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[34] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting Legacy Code for Authorization Policy Enforcement," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[35] D. Gao, M. K. Reiter, and D. Song, "Gray-Box Extraction of Execution Graphs for Anomaly Detection," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[36] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Transactions on Information and System Security*, vol. 9, no. 1, pp. 61–93, February 2006.

[37] S. N. Chari and P.-C. Cheng, "BlueBoX: A Policy–driven, Host–Based Intrusion Detection System," in *Proceedings of the 9^{th} Symposium on Network and Distributed Systems Security (NDSS 2002)*, 2002.

[38] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and Understanding Malware Hooking Behaviors," in *Proceedings of the 15^{th} Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.

[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 20^{th} Annual Computer Security Applications Conference (ACSAC) 2004*, December 2004.

[40] B. Demsky and M. C. Rinard, "Automatic Detection and Repair of Errors in Data Structures," in *Proceedings of the 18^{th} Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[41] A. Somayaji and S. Forrest, "Automated Response Using System-Call Delays," in *Proceedings of the 9^{th} USENIX Security Symposium*, August 2000.

[42] A. Brown and D. A. Patterson, "Rewind, Repair, Replay: Three R's to dependability," in *10^{th} ACM SIGOPS European Workshop*, Saint-Emilion, France, Sep. 2002.

[43] G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay," in *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, February 2002.

[44] S. T. King and P. M. Chen, "Backtracking Intrusions," in *19^{th} ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[45] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using Rescue Points to Navigate Software Recovery (Short Paper)," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[46] G. Ammons and J. Larus, "Improving Data-flow Analysis with Path Profiles," in *PLDI*, June 1998.