Software Self-healing Using Error Virtualization

Stylianos Sidiroglou

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2008

ABSTRACT

Stylianos Sidiroglou

Software Self-healing Using Error Virtualization

Despite considerable efforts in both research and development strategies, software errors and subsequent security vulnerabilities continue to be a significant problem for computer systems. The accepted wisdom is to approach the problem with a multitude of tools such as diligent software development strategies, dynamic bug finders and static analysis tools in an attempt to eliminate as many bugs as possible. Unfortunately, history has shown that it is very hard to achieve bug-free software. The situation is further exacerbated by the exorbitant cost of system down-time which some estimates place at six million dollars per hour. In the absence of perfect software, retrofitting error toleration and recovery techniques, in systems not designed to deal with failures, becomes a necessary complement to proactive approaches.

Towards this goal, this dissertation introduces and evaluates a set of techniques for recovering program execution in the presence of faults by effectively retrofitting legacy applications with exception handling techniques, *Error Virtualization* and *ASSURE* . The main premise of the approach is that there is a mapping between faults that may occur during program execution and a finite set of errors that are explicitly handled by the program's code. Experimental results are presented to support our hypothesis. The results demonstrate that our techniques can recover program execution in the case of failures in 80 to 90% of the examined cases, based on the technique used. Furthermore, the results illustrate that the performance overhead induced by

the techniques to protect against a specific fault can be minimized to under 10%. This dissertation also describes two deployment mechanisms, *Shadow Honeypots* and *Application Communities*, that can reduce the cost of monitoring the application and, in turn, enable efficient deployment strategies for error virtualization systems.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to briefly thank some of the people who supported this odyssey of a PhD.

I would like to thank Angelos D. Keromytis for his help and guidance in this work, and for being a great adviser, mentor and friend. He helped harness my energy towards something other than whining and I thank him for that.

I am also grateful to Jason Nieh for teaching me how to ask questions without underlying polemics and about research in general.

I would also like to thank the rest of my committee, Steve, Dan, Gail, and Nasir for their valuable insight and feedback.

I am thankful to all member of the Network Security Lab for their valuable time, advice, friendship and support.

I am grateful to Ali for her undying support and encouragement. I also thank her for being visible despite my myopia.

Finally, I would like to thanks my parents for their desire to procreate and pass genetic and memetic information to their offspring. Their unconditional support was sustaining.

Chapter 1

Introduction

Software errors and subsequent security vulnerabilities continue to be a thorn in the side of computer systems despite considerable efforts in both research and development strategies. They are of particular concern to high-availability systems where outages have been estimated to cost businesses billions of dollars each year [103]. Many traditional proactive approaches have been employed to attempt to address this problem by trying to make code as dependable as possible, through a combination of safe languages, libraries, compilers, code analysis tools, and development methodologies. Unfortunately, experience has shown [11] that it is very hard to achieve bug-free software.

Further exacerbating the problem is the fact that for existing proactive mechanisms the only available action upon detection of a fault is program termination, effectively a denial-of-service attack on the application. This situation is particularly troublesome for server applications that need to maintain high availability in the face of remote attacks, high-volume events (such as fast-spreading worms like Slammer and Blaster) that may trigger unrelated and possible non-exploitable bugs, or simple application-level denial of service attacks. Given these problems, we posit that in

the absence of perfect software, error toleration and recovery techniques become a necessary complement to proactive approaches.

Towards this goal, this dissertation describes the development of a general autonomic self-healing framework for handling a wide variety of software failures, ranging from remotely exploitable vulnerabilities to more mundane bugs that cause abnormal program termination (e.g., illegal memory dereference) or other recognizable bad behavior (e.g., computational denial of service). Briefly, the approach of this work to self-healing systems has been to use lightweight monitoring mechanisms to observe and analyze failures (including software vulnerabilities, but also bugs leading to system crashes), develop candidate fixes by modifying the software or its environment, and validate these fixes through a combination of static and dynamic analysis, and automated testing. The goal is to automate as much of the reaction ("self-healing") process as possible, toward building a completely autonomous system-healing mechanism.

The major contribution of this thesis is the introduction of the concept of *error virtualization* [111, 112, 108], a mechanism for program execution recovery. Briefly, error virtualization operates under the assumption that there exists a mapping between the set of errors that *could* occur during a program's execution (*e.g.,* a caught buffer overflow attack, or an illegal memory reference exception) and the limited set of errors that are explicitly handled by the program's code. Thus, a failure that would cause the program to crash is translated into a return with an error code from the function in which the fault occurred (or from one of its ancestors in the stack). Conceptually, error virtualization is a mechanism that retrofits exception-handling capabilities to legacy software. The main premise of error virtualization is that inside every complex software system there exists a well-tested core in which the system has been observed to behave acceptably. If the code inside this well-tested core can be

harnessed to handle errors that occur outside this space, as shown in Figure 2.1, then error virtulization can help an application handle the failure and call upon existing code to help with state cleanup and execution recovery.

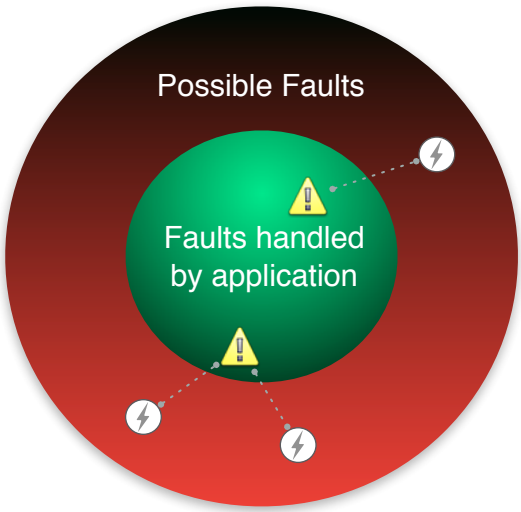


Figure 1.1: Map unexpected faults to well-tested application core

In support of this notion this dissertation describes the design, development and evaluation of three prototype systems that implement, refine and evaluate the concept of error virtualization: DYBOC [108], STEM [112] and ASSURE [111].

DYBOC is a mechanism that automatically instruments all statically and dynamically allocated buffers in an application so that any buffer overflow or underflow attack will cause transfer of the execution flow to a specified location in the code, from which the application can resume execution. Our hypothesis is that function calls can be treated as transactions that can be aborted when a buffer overflow is detected, without impacting the application's ability to execute correctly. Nested function calls are treated as sub-transactions, whose failure is handled independently. Our mechanism takes advantage of standard memory-protection features available in all modern operating systems and is highly portable. DYBOC is a stand-alone tool,

which simply needs to be run against the source code of the target application.

STEM is a binary supervision framework that is implemented as an instruction-level emulator, that can be selectively invoked for arbitrary segments of code. This tool permits the execution of emulated and non- emulated code inside the same process. The emulator is implemented as a C library that defines special tags (a combination of macros and function calls) that mark the beginning and the end of selective emulation. To use the emulator, we can either link it with an application in advance, or compile it (or dynamically inject it) in the code in response to a detected failure.

ASSURE presents the logical progression of Error Virtualization. ASSURE is a general software fault-recovery mechanism that uses operating system virtualization techniques to provide "rescue points" that an application can use to recover execution to, in the presence of faults. When a fault occurs at an arbitrary location in the program, we restore program execution to a "rescue point" and imitate its observed behavior to propagate errors and recover execution. Towards this goal, we have implemented an initial prototype that includes the checkpoint mechanism, application profiling, rescue point selection and recovery using popular error return values.

This dissertation evaluates the effectiveness of error virtualization as an enabling component of a general self-healing architecture that is able to deal with a variety of software failures with acceptable performance overhead. In the next Section, we describe two deployment architecture that help ameliorate the performance cost of application instrumentation.

Deployment Architectures Monitoring applications for faults using instrumentation can be expensive in terms of performance. For this reason, this work explored mechanisms that can reduce the cost of monitoring the application and, in turn, enable efficient deployment strategies for error virtualization systems.

This dissertation introduces an alternative deployment technique with Shadow Honeypots [17]. The main difference with the traditional honeypot approach is that the shadow honeypot shares state with the protected application and can therefore be used to detect targeted attacks (*e.g.,* attacks that exploit specific application state) against the application. The most significant advantage of the shadow honeypot approach is that it can ameliorate the cost of monitoring the application by deciding at run-time whether to service a request using extensive instrumentation or to run it natively. The decision to use the more expensive instrumented version of the application is arbitrated by an anomaly detector component that examines incoming requests for any signs of foul play.

Another alternative for minimizing the cost of protecting an application is to spread the monitoring cost across a number of hosts. In support of this notion, this thesis demonstrates the concept of Application Communities [76]. Application Communities (AC) are a collection of almost-identical instances of the same application running autonomously across a wide area network. Members of an AC collaborate in identifying previously unknown (zero day) flaws/attacks and exchange information so that such failures are prevented from re-occurring. While individual members may succumb to new flaws, over time the AC should converge to a state of immunity against that specific fault. The system learns from new faults and adapts to them, exploiting the AC size to achieve both coverage (in detecting faults) and fairness (in distributing the monitoring task).

This chapter begins with a brief overview of prior work in the area of execution recovery. We discuss the evolution of error virtualization from using source-to-source transformations and heuristics to using binary-level emulation and rescue-points. Finally, we motivate the use of different deployment mechanisms for alleviating performance concerns.

1.1 Background: Exploiting Software Elasticity

In trying to improve the safety and reliability of software systems, a panoply of techniques have been proposed, including error toleration and recovery. Recent research [97, 112, 111], including the concepts described in this dissertation, has revealed a new class of techniques for recovering from software faults, which exploit the concept of *software elasticity:* the ability of regular code to recover from certain types of failures when low-level faults are masked by the operating system (OS) or by appropriate instrumentation. The primary limitation of these techniques revolves around their lack of guarantees, in terms of altering program semantics, that can be provided. Masking the occurrence of faults will always carry this stigma since it forces programs down unexpected execution paths. However, we believe that the basic premise of masking failures to permit continued program execution is promising, and our goal in this work is to minimize the likelihood of undesirable side-effects.

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. This is an issue that is present in the majority of recovery efforts. A fault-recovery mechanism must evaluate and choose a response from a wide array of choices. In other approaches, when encountering a fault, a system can pick from the following options: crash [52], crash and be restarted by a monitor [34, 36], return arbitrary values [96, 97], change environment and replay [93], or slice off the functionality [108, 112]. This dissertation introduced the notion of "slicing off" functionality which was later refined with ASSURE. In contrast to previous solutions, ASSURE jumps to a safe (rescue) point and forces an error.

A problem that is inherent with all techniques that try to be oblivious to the fact that an error has occurred is the ability to guarantee session semantics. Altering

the functionality of the memory manager often leads to the uncovering of latent bugs in the code [29]. Rx [93] and failure oblivious computing [97] suffer from this limitation. Our approach explicitly triggers an application's fault recovery mechanism in an attempt to avoid precipitating post recovery symptoms.

For the remainder of this dissertation, we discuss how error virtualization can be applied to self-healing software systems as a mechanism that enables execution recovery in the presence of faults.

1.2 Contributions

My thesis is:

Error virtualization: the first technique that tries to recover program execution by mapping faults into return values used by the application code.

We validate this claim by implementing systems that allow server applications to recover from software failures. We evaluate these systems using both synthetic fault injection and real software failures. We also examine the performance implications of our system by measuring the overhead that they impose on several server applications.

Our contributions are:

- Introducing the idea of using existing application code to facilitate program execution recovery from software failures.
  - Design and implementation of two systems that leverage the notion: DYBOC, STEM.
- Introducing the notion of recovery using rescue points and building a system that evaluates the idea.
  - Design and implementation of rescue points: ASSURE

- Introducing the notion of Shadow Honeypots and building the first system that implements the concept.
- Developing the first system that implements the concept of Application Communities.

1.3 Dissertation Roadmap

In this dissertation, we discuss how error virtualization can be used to facilitate program execution recovery in the presence of faults and , in turn, enable software self-healing.

In Chapter 2, we introduce the core recovery techniques of this dissertation: error virtualization and error virtualization using rescue points.

In Chapter 3, we discuss related work.

In Chapter 4, we discuss DYBOC, the first tool we developed in support of error virtualization. We explain how DYBOC can be used to protect legacy application written in C/C++ from memory violations and how DYBOC can be selectively employed to reduce performance overhead.

In Chapter 5, we discuss STEM, a binary supervision framework that enables selective emulation (combination of native and emulated execution). We explain how selective emulation can be used to implement transactional semantics in legacy applications.

In Chapter 6, we introduce ASSURE, the first system to support error virtualization using resue points. We introduce rescue points and discuss how they can be used to restore program execution by imitating observed behavior. We examine how a process-level checkpoint/restart mechanism can be used to efficiently provide transactional semantics in source-available and binary environments.

Chapter 7, evaluates error virtualization's ability to recover program Execution. We examine error virtualization and error virtualization using rescue points using comprehensive synthetic fault injection and real bugs.

Chapter 8 discusses two deployment strategies that reduce the performance overhead of application instrumentation: Shadow Honeypots and Application Communities.

Finally, the dissertation concludes with Chapter 9.

Chapter 2

Error Virtualization

This Chapter serves as the introduction to error virtualization, and error virtualization using rescue points, as a recovery mechanism for self-healing software systems. We start the introduction of the concept of error virtualization and expand with with its logical progression: error virtualization using rescue points.

2.1 Error Virtualization

In the absence of a recovery mechanism, the only available action, upon detection of a fault, is program termination, effectively a denial-of-service on the application. While program termination in the presence of failures represents the prudent course-of-action for most systems, it is an undesirable solution for systems that need to maintain availability.

The major contribution of this dissertation is the introduction of the concept of *error virtualization* [111, 112, 108], a mechanism that enables software self-healing by enabling program execution recovery in the presences of failures.

Error virtualization (EV) operates under the assumption that there exists a map-

ping between the set of errors that *could* occur during a program's execution (*e.g.,* a caught buffer overflow attack, or an illegal memory reference exception) and the limited set of errors that are explicitly handled by the program's code. Thus, a failure that would cause the program to crash is translated into a return with an error code from the function in which the fault occurred (or from one of its ancestors in the stack). Conceptually, error virtualization is a mechanism that retrofits exception-handling capabilities to legacy software.

Error Virtualization along with a few other techniques [97, 98] forms the set of of novel techniques that exploit the concept of *software elasticity:* the ability of regular code to recover from certain types of failures when low-level faults are masked by the operating system (OS) or by appropriate instrumentation. Martin Rinard further expands on the concept with the description of the *comfort zone.* A program's comfort zone is represented by the tested region of a system's input space within which is has been observed to behave acceptably [98].

Error Virtualization (and EV using rescue-points) maps previously unseen errors into error codes that the existing application is expected to handle correctly (in the application's comfort zone). The basic premise is that although programmers might not be capable of handling the infinite space of errors that might occur in a program (especially with the addition of new features), they are capable of handling the core set of faults associated with the basic program operation. If previously unseen (or unexpected) errors can be mapped to existing and tested error handling code, the application will have a higher chance of surviving the failure (and correctly clean up after it self).

In this work, the mapping of faults into errors that are handled by the application is implemented through function return values. Other possible locations include mapping into existing exception handling techniques, basic blocks or even individual

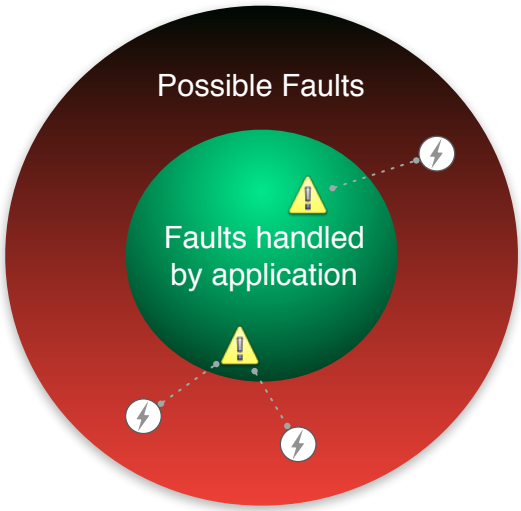


Figure 2.1: Map unexpected faults to well tested application core

instructions. We focus on function boundaries as our empirical examination shows that programmers tend to use unified and standard error semantics during program development at function granularity. For example, most functions in the Unix environment use a return value of `-1` to indicate an error and `0` for success. Thus, a natural point to initiate error handling code is at function invocation return points.

```
int vulnerable_function()
{
    if ( error )
        return -1;  /* return error */
    else
        return 0;   /* return success */
}
int main( int argc, char *argv[] )
{
    if ( vulnerable_function() < 0 )
        return -1;  /* do some error handling */
    else
        return 0;   /* computation was correct */
}
```

Backtracking to the exception handling analogy, error virtualization represents

a `try-catch` block that speculatively executes a portion of the application's code. If an error is detected during this speculative execution, computation is rolled back to the beginning of the "transaction". At that point, the failure is manifested as a return value from the function encompasses the execution. In the next section we describe the concept of execution transactions, which relies on a set of heuristics for mapping return values. Finally, we describe ASSURE or error virtualization using rescue points that replaces return value heuristics with observed (or mined from the application code) return codes.

2.1.1 Execution Transactions

The initial motivation for error virtualization was to produce a mechanism that allowed for the detection and recovery from memory violation failures such as buffer overflows.

The hypothesis is that function calls can be treated as transactions that can be aborted when a buffer overflow is detected, without impacting the application's ability to execute correctly. Nested function calls are treated as sub-transactions, whose failure is handled independently.

In determining how to recover from such exceptions, we introduce the hypothesis of an **execution transaction.** Very simply, we posit that for the majority of code (and for the purposes of defending against buffer overflow attacks), we can treat each function execution as a transaction (in a manner similar to a sequence of operations in a database) that can be aborted without adversely affecting the graceful termination of the computation. Each function call from inside that function can itself be treated as a transaction, whose success (or failure) does not contribute to the success or failure of its enclosing transaction. Under this hypothesis, it is sufficient to snapshot

the state of the program execution when a new transaction begins, detect a failure per our previous discussion, and recover by aborting this transaction and continuing the execution of its enclosing transaction.

2.1.2 Recovery: Forcing Error Returns

Upon detecting a fault, our recovery mechanism undoes memory changes and forces an error return from the currently executing function.

In our introductory work, source-to-source transformations were use to create "transactions" around vulnerable code segmenents as described in Chapter 4. In that work, transactions are implemented by adding calls to `sigsetjmp` and `siglongjmp` around vulnerable functions. This mechanism provides an efficient way to save and restore current execution environments but does not completely restore memory state, such as global variables (we describe work-around mechanisms in Chapter 4). A more complete solution for restoring all memory changes is described in Chapter 5 where we use an instruction-level emulator to track and restore all memory updates during the execution of a transaction. For the remainder of this section, we assume the use of the emulator for the implementation of the transactional semantics.

As previously mentioned, when a failure in execution is detected, the recovery mechanism restores any memory updates that might have occured during the execution of the transaction and forces the function that originated the transaction to return with an error value. To determine the appropriate error return value, we analyze the declared type of the function. Depending on the return type of the emulated function, the system returns an "appropriate" value. This value is determined based on heuristics and is placed in the stack frame of the returning function. The emulator then transfers control back to the calling function. For example, if the return type is

an *int,* a value of -1 is returned; if the value is *unsigned int* the system returns 0, *etc.* A special case is used when the function returns a pointer. Instead of blindly returning a *NULL*, we examine if the returned pointer is further dereferenced by the parent function. If so, we expand the scope of the emulation to include the parent function. We handle value-return function arguments similarly. There are some contexts where this heuristic may not work well; however, as a first approach these heuristics worked extremely well in our experiments (see Section 5.9).

A more detailed description of our algorithm is described in Algorithm 1.

Algorithm 1: Return-value mapping

```
1  cfg ← get_stack_trace()
2  foreach node in cfg do
3      return_type ← get_ret_type(function)
4      if return_type is int then
5          return_value ← -1
6          break
7      else if return_type is unsigned int then
8          return_value ← 0
9          break
10     else if return_type is char then
11         return_value ← 0
12         break
13     else if return_type is void then
14         return_value ← return
15         break
16     else
17         continue
18     end
19 end
```

In future work, we plan to use more aggressive source code analysis techniques to determine the return values that are appropriate for a function. Specifically, we plan to examine how return values are used in functions (i.e. what return values lead to a break in execution).

Since in many cases a common error-code convention is used in large applications or modules, it may be possible to ask the programmer to provide a short description of this convention as input to our system either through code annotations or as separate input. A similar approach can be used to mark functions that must be fail-safe and should return a specific value when an error return is forced, *e.g.,* code that checks user permissions.

2.2 Error Virtualization Using Rescue Points

We now describe the basic concept of error virtualization using rescue points, which combined with the checkpoint-rollback mechanism, forms the core of the remediation component. We describe the process in a top-down approach, starting with error virtualization using rescue points, followed by a detailed description of individual components.

Error virtualization using rescue points is the primary mechanism employed by ASSURE to recover from software faults. It is the end-product of ASSURE's operation, and its purpose is to protect against the manifestation of an already analyzed fault at the production system. Its operation can be summarized by the following steps: (a) checkpoint application state at the rescue point; (b) monitor application for faults in the protected code; (c) when a fault occurs, roll back application state to the rescue point; (d) after the rollback to the rescue point, force return with an error using an observed value from the application profiler.

Figure 6.2 illustrates ASSURE's self healing on a real bug in the Apache web server. A detailed description of the bug is provided in Appendix A.0.2. The scenario involved the execution of three functions: `ap_proxy_ftp_handler()`, `ap_pass_brigade()` and `ap_proxy_send_dir_filter()`. Due to bad input, the bug manifests in `ap_proxy_send_dir_filt`

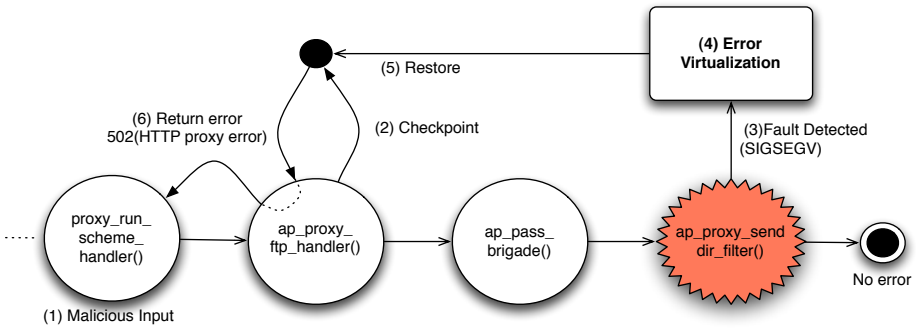


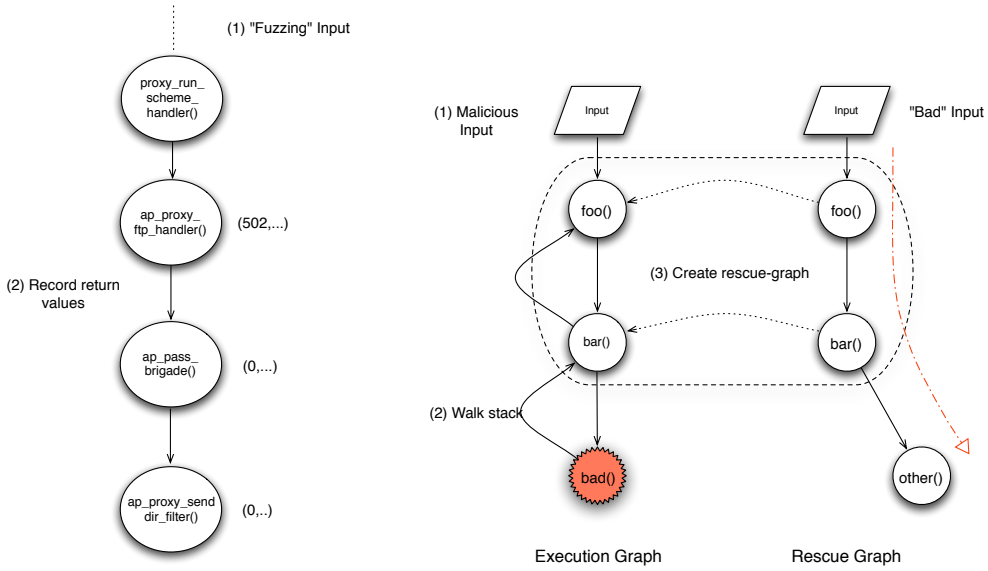
Figure 2.2: Error virtualization (EV) using rescue points: (1) Fault-triggering input is supplied to the application. (2) The application checkpoints its state at a rescue point. (3) A fault detector monitors protected functions for faults. When a fault occurs in a protected function, (4) the EV decides on a recovery strategy and (5) rolls back application state to the rescue point. (6) Once application state is restored, the EV forces an error return using the determined value.

and results in a memory fault (SIGSEGV). ASSURE intercepted the first occurrence of the fault, identified `ap_proxy_ftp_handler()` as a suitable rescue point, and instrumented the Apache server as follows. Whenever the patched server enters `ap_proxy_ftp_handler()`, ASSURE takes a checkpoint of the server state, and allows the server to continue execution. If the same (or similar) fault occurs in `ap_proxy_send_dir_filter()`, the fault detection component in the patched server detects the error and notifies the error virtualization (EV) component. The EV component analyzes the fault information and rolls back the server state back to the rescue point in `ap_proxy_ftp_handler()`. Furthermore, instead of allowing execution to proceed down the same path that caused the fault to manifest, ASSURE uses error virtualization to force `ap_proxy_ftp_handler()` to return with an error value identified during the application profiling stage, namely 502 (HTTP "Proxy Error").

Alternatively, programmers can design software with error virtualization in mind, where specific locations in the code can be assigned, a priori, as rescue points that propagate faults gracefully. Programmer insight is difficult to replicate with auto-

mated techniques, especially when dealing with code cleanup and efficiency. We envision that programming with error virtualization will prove easier than dealing with language specific constructs, such as exception handling, since attention can be focused on a few select program points. However, in this paper we focus on fully automated techniques for every aspect of our system, to determine the limits of such an approach.

In the remainder of this section we describe how ASSURE discovers, selects, creates, tests, and deploys rescue points.



(a) A section of the rescue-trace

(b) Generating the rescue-graph

Figure 2.3: Figure (a) shows the process of creating rescue-traces. (1) The application is bombarded with "bad" input. (2) Its behavior is monitored and error return values are recorded. Figure (b) shows how rescue-graphs are extracted to identify candidate rescue points for a particular fault: (1) When an fault-detection monitor diagnoses a fault (2) it examines that call-graph that lead to the fault. (3) The call-graph is compared with the rescue-trace to find common nodes that, in turn, form the rescue-graph: the set of candidate rescue points along with the suggested error return value.

2.2.1 Rescue Point Discovery

Determining a suitable recovery point is of pivotal importance to error virtualization. It determines, to a large extent, the likelihood that the application will survive a fault. During the application profiling stage, ASSURE uses dynamic analysis with fuzzing to discover suitable rescue points. The goal is to learn how an application responds to "bad" input, under controlled conditions, and use this knowledge in the future to map previously unseen faults to a set of observed fault behaviors. For example, we would like to see how a program normally handles errors when stress-tested by quality assurance tests. Learning from "bad" behavior has been used in machine learning and subsequently intrusion detection systems, but to the best of our knowledge has not been used to recover from software faults.

Specifically, ASSURE instruments applications by inserting monitoring code at every function's entry and exit points using the run-time injection capabilities of Dyninst [31], a runtime binary injection tool. The instrumentation records return values, function parameters, and return types (the latter two are available only when the binary is not stripped) while the application is bombarded with faults (through fault injection) and fuzzed inputs (*e.g.,* malformed protocol requests). From these traces, ASSURE extracts function call-graphs along with the history of return values used at each point in the graph. We call these graphs the *rescue-traces*.

Figure 6.3a illustrates part of the rescue-trace generated using the Apache bug example introduced earlier. The figure shows a summarized execution trace that includes `ap_proxy_ftp_handler()`, `ap_pass_brigade()` and `ap_proxy_send_dir_filter()`, as well as the observed error values that are associated with each function, which are 502, 0 and 0 respectively.

Algorithm 2: Rescue-point selection algorithm

```
1  rg ← get_rescue_graph(function)
2  cfg ← get_stack_trace()
3  rescue_set ← rg ∩ cfg
4  foreach node in rescue_set do
5      create_rescue_point(node)
6      replay_failure(ckpt)
7      if is_server_ok() and is_output_correct() then
8          create_rescue_point()
           /* Found rescue point                                    */
9          break
10     else
11         continue
12     end
13 end
```

2.2.2 Rescue Point Selection

Algorithm 2 illustrates the rescue-point selection algorithm. In detail, when a fault is detected in a specific code region for the first time, the call-stack is examined to derive the sequence of functions that led to the fault. At that point, ASSURE compares the call-stack with the rescue-trace from the discovery phase to derive common nodes. The common nodes form the set of candidate rescue points, or the *rescue-graph*.

Once candidate rescue points are identified, ASSURE attempts to determine their return type. If debugging information is available, function return types can be extracted directly from the binary. In the case of stripped binaries, as is the case with most commercial off-the-shelf (COTS) applications, ASSURE estimates the actual return type of the function through a set of heuristics that work on the observed return values found in profiling traces and through binary analysis [59]. Candidate rescue points are filtered according to heuristics that consider both the return type (if available) and the observed return values. Currently, candidate rescue points are functions with non-pointer return types, or functions that return pointers but

the observed return value is `NULL`. Functions that return pointers require a deeper inspection of the data structures to ascertain the values of their return types beyond the simple case of returning a `NULL`. Preliminary empirical examination shows that the examined *C* programs favor the use of integer return types as failure indicators.

Next, ASSURE examines the return value distribution that was found at each candidate rescue point. The objective is to find a value that the error virtualization component can use to trigger error-handling code. The obvious strategy is to use the most frequently used return value, given that the profiling runs consist of execution traces that propagate errors. This is especially true in the absence of source code, where ASSURE cannot verify how the actual code handles errors.

Figure 6.3b illustrates how candidate rescue points are identified for a particular fault. When a fault is detected in `ap_proxy_send_dir_filter()`, the call-stack is examined to determine the execution path that had lead to the observed failure. This path is compared against the rescue-trace to determine overlapping functions that form the rescue-graph. Using the same example from the figure, functions `ap_proxy_ftp_handler()` and `ap_pass_brigade()` form the rescue graph for the particular fault instance.

The rescue points in the rescue-trace can be sorted using different selection strategies. We chose perhaps the simplest: sort the rescue points by shortest distance to the faulty code that represents an active function on the call graph. The idea is that suitable rescue points that reside closer to the fault will minimize the performance overhead that rescue points incur (due to checkpointing and monitoring for the specific fault), since they might avoid critical application paths that get invoked on each request. ASSURE follows this ordering to instantiate and test rescue points, seeking one that enables recovery from the given fault. If the closest rescue point fails during testing, ASSURE chooses the nearest active ancestor and repeats.

2.2.3 Rescue Point Creation

Having determined a set of candidate rescue points, ASSURE can now activate and test rescue points. To activate a rescue point, ASSURE needs to insert code into the application running in the testing environment. This is done using the same mechanism for deploying the rescue point on the production server as described in Section 6.3.4. Using this mechanism, ASSURE activates a rescue point by inserting at the function designated as the rescue point a call to `int rescue_capture(id,fault)`. The parameter `id` uniquely identifies a rescue point; `fault` is a structure that contains all additional information pertaining to the rescue point, including the error virtualization code to be used to force an early return.

The `rescue_capture()` function is responsible for capturing the state of the application as it executes through the rescue point by performing a checkpoint. Checkpoints are kept entirely in memory using standard copy-on-write semantics and are indexed by their corresponding identifiers. `rescue_capture()` returns the rescue point identifier upon a successful checkpoint, or zero when it returns following a rollback of the application state. A typical calling sequence is given in the following code snippet. Similar to `fork()` semantics, the return value of the function `rescue_capture()` directs the execution context.

```
int rescue_point( int id, fault_t fault ) {
    int ret = rescue_capture(id, fault);
    if (ret < 0)
        handle_error(id);  /* rescue point error */
    else if (ret == 0)
        return get_rescue_return_value(fault);
    /* all ok */
    ...
}
```

The handling of rescue points becomes more complicated in the presence of multi-

process or multi-threaded applications. The key issue with rescue points is that the checkpoints are always initiated by the application, since they must occur at designated safe locations. With multiple (cooperative) processes, the state must be obtained not only in a globally consistent manner, but also with all the participating processes stationed at some suitable point. By "cooperative" we refer to processes that share resources, such as shared memory, or some portion of their state, and therefore must agree on that state at any point in time. On the other hand, if the processes are independent, then they can be handled independently one by one as in the single-process case.

ASSURE ensures global consistency of checkpoints by using coordinated checkpoints of cooperative process groups. When a process in a group reaches a rescue point, it must wait until the remaining processes are ready in order to take the checkpoint synchronously. While processes clearly become ready when they reach a rescue point, the implied synchronization can adversely impact performance. This is because there are no hard guarantees that the other processes will reach a safe point soon. To mitigate this, ASSURE can checkpoint processes when they enter or exit the kernel due to system calls or signal handling. Thus, when a process initiates a checkpoint, ASSURE marks other processes in the group processes, telling them to request a checkpoint as soon as possible. If other processes are blocked on I/O, signals are used to force them out of blocking calls.

2.2.4 Rescue Point Testing

Once a candidate rescue point has been elected, ASSURE proceeds to verify the efficacy of the proposed fix, by testing the rescue-enabled version of the application. To accomplish this, ASSURE restarts the application from the most recent checkpoint

image available in a separate testing environment, and then feeds it with the specific input that caused the fault to manifest. If the offending input cannot be identified easily, ASSURE replays longer input sequences recorded since that checkpoint. Eventually the fault occurs and triggers a rollback to the selected rescue point. If the application crashes, fails to maintain service availability, or is not semantically equivalent, a new fix is created using the next available candidate rescue point and the testing and analysis phase is repeated.

If the fix does not introduce any faults that cause the application to crash, the application is examined for semantic bugs using a set of user-supplied tests. The purpose of these tests is to increase confidence about the semantic correctness of the generated fix. For example, an online vendor could run tests that verify that client orders can be submitted and processed by the system.

For our initial approach, we are primarily concerned with failures where there is a one-to-one correspondence between inputs and failures, and not with those that are caused by a combination of inputs. Note, however, that many of the latter types of failures are in fact addressed by our system, because the last input (and the code leading to a failure) will be recognized as "problematic" and handled as we have discussed.

2.2.5 Rescue Point (Patch) Deployment

Swift patch deployment is of foremost importance in "reactive" systems. First, it reduces system downtime and subsequently improves system availability. Second, it allows for the deployment of critical fixes that could curtail the spread of large-scale epidemics such as in the case of worms. Previous work has relied on a traditional software development cycle of making changes to source code (albeit automatically

through source-to-source transformations), compiling, linking, testing and then instantiating the new version of the application. For this work, we examined binary instrumentation as a mechanism for deploying patches. Specifically, we chose to use runtime injection and Dyninst [31] in particular due to its low runtime overhead and its ability to attach and detach from already running processes. The Dyninst API provides a rich interface that facilitates code steering through injection and call-site replacement within functions. Note that in addition to being used for the final rescue point patch deployment on the production server, the same runtime injection mechanism is also used to insert rescue points into the shadow deployment of the application during rescue-point testing, and to inject the fault monitoring mechanism into the production server.

Chapter 3

Related Work

To the best of our knowledge, Error Virtualization is the first technique that generalizes software faults into return values that can be used by an application to aid in recovery. The work described in this dissertation is the first to show the benefit server applications can derive from error virtualization when it comes to surviving from software faults. This work is also first to introduce the notion of shadow honeypots and first to produce a working system for the concept of Application Communities.

The design and implementation of this dissertation has benefited greatly from prior work. In Section 3.2, we discuss related work in the area of self-healing software systems. In Section 3.1, we examine related work on systems that explore the notion of software elasticity. The remainder of the chapter discusses prior work on transactional processing, protection mechanisms and automated testing.

3.1 Software Elasticity and Error Recovery

The acceptability envelope, a region of imperfect but acceptable software systems that surround a perfect system, as introduced by Rinard [95] promotes the idea that cur-

rent software development efforts might be misdirected. Rinard explains that certain regions of a program can be neglected without adversely affecting the overall availability of the system. To support these claims, a number of case studies are presented where introducing faults such as an off-by-one error does not produce unacceptable behavior. This work supports our claim that most complex systems contain the necessary framework to propagate faults gracefully and the error toleration allowed by our system expands the acceptability envelope of a given application.

In the same motif, Rinard *et al.* [96] have developed a compiler that inserts code to deal with writes to unallocated memory by virtually expanding the target buffer. Such a capability aims toward the same goal our system does: provide a more robust fault response rather than simply crashing. The technique presented in [96] is modified in [97] and introduced as *failure-oblivious computing*. Because the program code is extensively re-written to include the necessary checks for *every* memory access, their system incurs overheads ranging from 80% up to 500% for a variety of different applications. Another issue with this work is the lack of a thorough examination on the side effects of the proposed technique. The authors test their approach on a few applications with specific memory error vulnerabilities but do not provide any further insight into to how their technique would deal with errors and unexpected execution paths. Finally, *failure-oblivious computing* is only applicable to memory errors where our technique can be applied to a variety faults.

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. An important contribution in this area is [49], which discusses mechanisms for detecting corrupted data structures and fixing them to match some pre-specified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate when faults were randomly

injected. Similar results are shown in [133] when program execution is forced to take the "wrong" path at a branch instruction, program behavior remains the same in over half the times.

Rx [93] presents an alternative approach for recovering program execution where applications are periodically checkpointed and continuously monitored for faults. When an error occurs, the process state is rolled back and replayed in a new "environment". If the changes in the environment do not cause the bug to manifest, the program will have survived that specific software failure. The issue with this approach is that it is primarily focused on non-malicious bugs, thus it is open to gaming attacks that can effectively lead to a whole program restart, effectively inducing a denial-of-service attack on the application. It is unclear, at least from the current implementation, that memory attacks can be correctly handled. Specifically, the buffer overflow cases examined in the paper, in contrast to what is reported, could not be recovered using the said techniques; the requests had to be dropped. In order to replay execution after a checkpoint and to mask the manifestation of faults to the client, Rx needs to employ a protocol-aware application proxy that must be capable of filtering out information such as time stamps that would confuse the client program.

The results presented by Chandra and Chen [38, 39] make an interesting case for the use of application-specific recovery mechanisms, such as the ones described in this proposal. Specifically, they examine a set of bugs found in three large open-source application and classify them according to their dependence on the operating system environment. Surprisingly, their findings show that 72-87% of the faults examined are deterministic. This is a very interesting result for the following reasons: it shows that the majority of the bugs are likely persist on retry (non-transient), and that application-specific mechanisms are essential for program recovery. The deterministic nature of these bugs means that techniques, such as Rx, that make changes to the

underlying environment and replay the input, are unlikely to be successful in program recovery for the majority of the cases. In fact, deterministic faults require some form of application-specific recovery mechanism. The mechanisms presented in this proposal are a hybrid approach of application-specific and -generic recovery mechanisms so this result reinforces our intuition.

One approach that can be employed by error virtualization techniques is where function-level profiles are constructed during a training phase that can, in turn, be used to predict function return values [77]. While this technique is useful for predicting appropriate return values, especially in the absence of return type information, it suffers from the same problems as error virtualization, *i.e.,* it is oblivious to errors. Of further interest is the work proposed in Triage [129]. Triage can automatically provide failure diagnosis using checkpoint/reexecution mechanism and an online version of the delta debugging [141]. The diagnosis information reported by Triage can be used to help better predict rescue point locations and recovery success.

A problem that is inherent with all techniques that try to be oblivious to the fact that an error has occurred is the ability to guarantee session semantics. Altering the functionality of the memory manager often leads to the uncovering of latent bugs in the code [29]. Rx and failure oblivious computing suffer from this limitation. Our approach, explicitly triggers an application's fault recovery mechanism in an attempt to avoid precipitating post recovery symptoms.

3.2 Self-healing Systems

Our work on end-to-end self-healing software systems is most related to a recent Microsoft research effort, Vigilante [43]. Vigilante, similarly to our work [109], proposes a collaborative end-host based worm defense mechanism. The main advancement over

our work is the use of self-certifying alerts (SCAs). SCAs are used to communicate the existence of a vulnerability that can be verified by each host independently at a low cost. At a technical level, there are several differences between the two works: (a) vigilante does not provide a mechanism for recovering program execution upon detection of fault, (b) it cannot protect against attacks that target specific application state, (c) does not provide a mechanism to ameliorate the cost of instrumentation and (d) employs a filter-based towards detecting attacks which, can lead to false positives.

Shield [131] and VSEF [30] generate vulnerability-specific signatures instead of input-specific signatures. They provide the ability to handle specific application state and encrypted traffic and have many fewer false positives than network- or host-based input filtering. However, the only option available upon detection of a malicious input is to terminate execution. They address a broader class of vulnerabilities than buffer overflow protection mechanisms, but like those mechanisms, they only protect system integrity: they do not help with improving system availability. Shield requires manual creation of vulnerability-specific signatures, while VSEF automatically generates and injects them into application binaries. However, VSEF has not been demonstrated to work on any real applications except ATPhttpd, an old and minimalistic webserver.

Sweeper [130] combines the Rx checkpoint-restart mechanism and proxy with VSEF. VSEF is used to more accurately detect bug-inducing faults. If a fault occurs, Sweeper uses taint analysis and backward slicing to identify the input that led to the failure, generates an input filter to drop this and similar future requests, then rolls back to a previous checkpoint and replays the input *sans* the bad request. Since Sweeper reduces VSEF to being used for input signature generation, it suffers from the same input filtering limitations described earlier (polymorphism and encrypted traffic). Furthermore, the expensive analysis used to generate the filters can take over half a minute, resulting in large delays as the proxy also must replay over half a

minute of traffic. Since Sweeper uses VSEF and the Rx proxy and checkpoint-restart mechanism, it shares the combined limitations of both mechanisms. Thus, it is not surprising that empirical evidence indicating whether Sweeper can recover from bugs in realistic application deployments was even more limited than just using Rx.

3.3 Protection Mechanisms

There are a number of available fault detection components that can detect memory errors [82, 52, 107, 44, 46, 45, 20, 85] and some that detect violations to underlying security policies [91, 68, 12].

Recently, program information flow or "tainting" has been used as a technique for thwarting a wide range of control-hijacking attacks [122, 115, 83]. Suh *et al.* [122], propose a hardware based solution that can be used to thwart control-transfer attacks and restrict executable instructions by monitoring "tainted" input data. In order to identify "tainted" data, they rely on the operating system. If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception that can be handled by the operating system. The authors do not address the issue of recovering program execution and suggest the immediate termination of the offending process. DIRA [115] is a technique for automatic detection, identification and repair of control-hijacking attacks. This solution is implemented as a GCC compiler extension that transforms a program's source code adding heavy instrumentation so that the resulting program can perform these tasks. The use of checkpoints throughout the program ensures that corruption of state can be detected if control sensitive data structures are overwritten. Unfortunately, the performance implications of the system make it unusable as a front line defense mechanism. Song and Newsome [83] propose dynamic taint analysis for automatic detection of over-

write 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 but suffers from severe performance degradation.

Starting with the technique of *program shepherding* [69], the idea of enforcing the integrity of control flow has been increasingly researched. Program shepherding validates branch instructions to prevent transfer of control to injected code and to make sure that calls into native libraries originate from valid sources. Control flow is often corrupted because input is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call.

Observing that high-level programming often assumes properties of control flow that are not enforced at the machine level, Control Flow Integrity (CFI) [13] provides a way to statically verify that execution proceeds within a given control-flow graph. The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. To improve software reliability, hardware thread-level speculation [88] executes an application's monitoring code in parallel with the primary computation and rolls back the computation "transaction" depending on the results of the monitoring code.

Other promising work in this space [61] uses virtual machines combined with vulnerability-specific predicates that are executed in the context of the vulnerable process to test whether a flaw is being exercised (or was exercised at some point in the past, if such logs are available). However, these predicates must be generated manually, and require intimate knowledge of the application and the patch (or the vulnerability).

3.4 Transactional Processing

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (specifically, Java) by Wallach et al [101, 100]. That work focused on safely terminating misbehaving threads, introducing the concept of "soft termination". Soft termination allows threads to be terminated while preserving the stability of the language runtime, without imposing unreasonable performance overheads. In that approach, threads (or *codelets*) are each executed in their own transaction, applying standard ACID semantics. This allows changes to the runtime's (and other threads') state made by the terminated codelet to be rolled back. The performance overhead of their system can range from 200% up to 2,300%. Relative to that work, our contribution is twofold. First, we apply the transactional model to an unsafe language such as *C*, addressing several (but not all) challenges presented by that environment. Second, by selectively applying transactional processing, we substantially reduce the performance overhead of the application. However, there is no free lunch: this reduction comes at the cost of allowing failures to occur. Our system aims to automatically evolve a piece of code such that it *eventually* (*i.e.*, after an attack has been observed) does not succumb to attacks.

3.5 Automated Testing

Manually testing applications for software errors is a daunting undertaking even for the simplest of systems. Random testing has been proposed as a complementary mechanism to traditional testing approaches. Generating random input can help uncover software errors that are not easily detectable using manual techniques alone [79]. Unfortunately, true random testing is very inefficient at covering corner cases

and at generating input that has structure. Recently, some interesting work has been done in trying to automate the unit testing of software [55, 14, 33]. These techniques use symbolic execution in conjunction with dynamic analysis to help direct program execution down untested paths. DART [55] and its more complete extension EXE [14], completely automate the process of unit testing applications by generating test-driver, harness code and continuously running the application until sufficient coverage is achieved. We believe that such techniques can be of great use to our system. Specifically, the ability to test the effectiveness of our recovery technique with highly esoteric input handling cases would greatly help with the discovery of highly robust rescue points.

Another popular line of testing is that of software-implemented fault injection (SWIFI). Injecting faults in component code has been shown to produce realistic failures [140]. Deciding what types of faults to inject is an interesting sub-problem. Studies have identified representative programming errors common to operating system code [123, 41]. In our examination, we were initially concerned with the examination of fault propagation at particular function boundaries. The techniques described above randomly inject errors without any guaranties as to where they will manifest. In the future, we plan to expand our fault injection study with one of the aforementioned injection techniques [123, 41].

Chapter 4

DYBOC: DYnamic Buffer Overflow Containment

This Chapter demonstrates the use of light-weight operating system checkpoint/restart mechanisms along with source-to-source transformations for enabling self-healing software on applications written in C/C++. In this Chapter we introduce the notion of execution transactions where functions execution is treated as transaction that can be aborted without adversely affecting graceful termination of computation.

4.1 Introduction

The motivation behind our fist software self-healing tool, DYBOC (Dynamic Buffer Overflow Containment) was to create a system that allows system developers to better handle the trade-off between security and availability. As suggested by the name of our tool, the goal was to create a mechanism that allowed systems to protect against buffer overflow attacks without severely impacting application performance.

Protecting against buffer overflow attacks (removing, containing or mitigating)

has been the focus of a considerable body of work as evidenced by the literature [46, 53]. These techniques suffer from at least on the following problems:

- There is a poor trade-off between security and availability: once an attack has been detected, the only option available is to terminate program execution [46, 53], since the stack has already been overwritten. Although this is arguably better than allowing arbitrary code to execute, program termination is not always a desirable alternative (particularly for critical services). *Automated, high-volume attacks, e.g., a worm outbreak, can exacerbate the problem by suppressing a server that is safe from infection but is being constantly probed and thus crashes.*

- Severe impact in the performance of the protected application: dynamic techniques that seek to detect and avoid buffer overflow attacks during program execution by instrumenting memory accesses, the performance degradation can be significant. Hardware features such as the NoExecute (NX) flag in recent Pentium-class processors [53] address the performance issue, but cover a subset of exploitation methods (*e.g.,* jump-into-libc attacks remain possible).

- Ease of use: especially as it applies to translating applications to a safe language such as Java or using a new library that implements safe versions of commonly abused routines.

An ideal solution uses a comprehensive, perhaps "expensive" protection mechanism only where needed and allows applications to gracefully recover from such attacks, in conjunction with a low-impact protection mechanism that prevents intrusions at the expense of service disruption.

4.1.1 Our Contribution

We have developed such a mechanism that automatically instruments all statically and dynamically allocated buffers in an application so that any buffer overflow or underflow attack will cause transfer of the execution flow to a specified location in the code, from which the application can resume execution. *Our hypothesis is that function calls can be treated as transactions that can be aborted when a buffer overflow is detected, without impacting the application's ability to execute correctly.* Nested function calls are treated as sub-transactions, whose failure is handled independently. Our mechanism takes advantage of standard memory-protection features available in all modern operating systems and is highly portable. We implement our scheme as a stand-alone tool, named DYBOC (DYnamic Buffer Overflow Containment), which simply needs to be run against the source code of the target application. Previous research [100, 101] has applied a similar idea in the context of a safe language runtime (Java); we extend and modify that approach for use with unsafe languages, focusing on single-threaded applications. Because we instrumented memory regions and not accesses to these, our approach does not run into any problems with pointer aliasing, as is common with static analysis and some dynamic code instrumentation techniques.

We applied DYBOC to 17 open-source applications (taken from the CoSak dataset) with known buffer overflow exploits. DYBOC, enabled the applications to continue executing, despite the triggered buffer overflow attack, for 14 of the examined applications. In the remaining 3 cases, the program terminated; in no case did the attack succeed. Although a contrived micro-benchmark showed a performance degradation of up to 440%, measuring the ability of an instrumented instance of the Apache web server indicated a performance penalty of only 20%. We provide some preliminary experimental validation of our hypothesis on the recovery of execution transactions

by examining its effects on program execution on the Apache web server. We showed that when each of the 154 potentially vulnerable routines are forced to fail, 139 result in correct behavior, with similar results for *sshd* and Bind. Our approach can also protect against heap overflows.

Although we believe this performance penalty (as the price for security and service availability) to be generally acceptable, we provide further extensions to our scheme to protect only against specific exploits that are detected dynamically. This approach lends itself well to defending against scanning worms. Briefly, we use an instrumented version of the application (*e.g.,* web server) in a sandboxed environment, with all protection checks enabled. This environment operates *in parallel with* the production servers, but is not used to serve actual requests nor are requests delayed. Rather, it is used to detect "blind" attacks, such as when a worm or an attacker is randomly scanning and attacking IP addresses. We use this environment as a "clean room" to test the effects of "suspicious" requests, such as potential worm infection vectors. A request that causes a buffer overflow on the production server will have the same effect on the sandboxed version of the application. The instrumentation allows us to determine the buffers and functions involved in a buffer overflow attack. This information is then passed on to the production server, which enables that subset of the defenses that is necessary to protect against the detected exploit. In contrast with our previous work, where patches were dynamically generated "on the fly" [109, 110], DYBOC allows administrators to test the functionality and performance of the software with all protection components enabled. Even by itself, the honeypot mode of operation can significantly accelerate the identification of new attacks and the generation of patches or the invocation of other protection mechanisms, improving on the current state-of-the-art in attack detection [92, 60].

4.2 Approach

The core of our approach is to automatically instrument parts of the application source code[1] that may be vulnerable to buffer overflow attacks (*i.e.,* buffers declared in the stack or the heap) such that overflow or underflow attacks cause an exception. We then catch these exceptions and recover the program execution from a suitable location.

This description raises several questions: Which buffers are instrumented? What is the nature of the instrumentation? How can we recover from an attack, once it has been detected? Can all this be done efficiently and effectively? In the following subsections we answer these questions and describe the main components of our system. The question of efficiency and effectiveness is addressed in the next section.

4.2.1 Instrumentation

Since our goal is to contain buffer overflow attacks, our system instruments all statically and dynamically allocated buffers, and all read and writes to these buffers. In principle, we could combine our system with a static analysis tool to identify those buffers (and uses of buffers) that are provably safe from exploitation. Although such an approach would be an integral part of a complete system, we do not examine it further here; we focus on the details of the dynamic protection mechanism. Likewise, we expect that our system would be used in conjunction with a mechanism like StackGuard [46] or ProPolice to prevent successful intrusions against attacks we are not yet aware of; following such an attack, we can enable the dynamic protection mechanism to prevent service disruption. We should also note the "prove and check" approach has been used in the context of software security in the past, most notably

[1] Binary rewriting techniques may be applicable, but we do not further consider them due to their significant complexity.

in CCured [82]. In the remainder of this paper, we will focus on stack-based attacks, although our technique can equally easily defend against heap-based ones.

For the code transformations we use TXL [62], a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Naur (BNF). Several parsing strategies are supported by TXL making it comfortable with ambiguous grammars allowing for more "natural" user-oriented grammars, circumventing the need for strict compiler-style "implementation" grammars. In our system, we use TXL for *C*-to-*C* transformations using the GCC *C* front-end.

The instrumentation is fairly straightforward: we move static buffers to the heap, by dynamically allocating the buffer upon entering the function in which it was previously declared; we de-allocate these buffers upon exiting the function, whether implicitly (by reaching the end of the function body) or explicitly (through a *return* statement).

Original code

```
int func()
{
    char buf[100];
...
    other_func(buf);
...
    return 0;
}
```

Modified code

```
int func()
{
    char *buf = pmalloc(100);
...
    other_func(buf);
...
    pfree(buf); return 0;
}
```

Figure 4.1: First-stage transformation, moving buffers from the stack to the heap with *pmalloc()*.

For memory allocation we use *pmalloc(),* our own version of *malloc()*, which allocates two zero-filled, write-protected pages surrounding the requested buffer.

The guard pages are *mmap()*'ed from */dev/zero* as read-only. As *mmap()* operates at memory page granularity, every memory request is rounded up to the nearest page. The pointer that is returned by *pmalloc()* can be adjusted to immediately catch any buffer overflow or underflow depending on where attention is focused. This functionality is similar to that offered by the *ElectricFence* memory-debugging library, the difference being that *pmalloc()* catches both buffer overflow and underflow attacks. Because we *mmap()* pages from */dev/zero*, we do not waste physical memory for the guards (just page-table entries). Some memory is wasted, however, for each allocated buffer, since we round to the next closest page. While this could lead to considerable memory waste, we note that in our experiments the overhead has proven manageable.

Figure 8.6 shows an example of such a translation. Buffers that are already allocated via *malloc()* are simply switched to *pmalloc()*. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the *C* grammar to free the variables before the function returns. After making changes to the standard ANSI *C* grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial. For single-threaded, non-reentrant code, it is possible to use *pmalloc()* once for each previously-allocated static buffer. Generally, however, this allocation needs to be done each time the function is invoked.

Any overflow or underflow attack to a *pmalloc()*-allocated buffer will cause the process to receive a Segmentation Violation (SEGV) signal, which is caught by a signal handler we have added to the source code. It is then the responsibility of the signal handler to recover from such failures.

4.2.2 Recovery: Execution Transactions

In determining how to recover from such exception, we introduce the hypothesis of an **execution transaction.** Very simply, we posit that for the majority of code (and for the purposes of defending against buffer overflow attacks), we can treat each function execution as a transaction (in a manner similar to a sequence of operations in a database) that can be aborted without adversely affecting the graceful termination of the computation. Each function call from inside that function can itself be treated as a transaction, whose success (or failure) does not contribute to the success or failure of its enclosing transaction. Under this hypothesis, it is sufficient to snapshot the state of the program execution when a new transaction begins, detect a failure per our previous discussion, and recover by aborting this transaction and continuing the execution of its enclosing transaction. Currently, we focus our efforts inside the process address space, and do not deal with rolling back I/O. For this purpose, a virtual file system approach can be employed to roll back any I/O that is associated with a process. We plan to address this further in future work, by adopting the techniques described in One-way Isolation [124]. However, there are limitations to what can be done, *e.g.,* network traffic.

Note that our hypothesis does not imply anything about the correctness of the resulting computation, when a failure occurs. Rather, it merely states that if a function is prevented from overflowing a buffer, it is sufficient to continue execution at its enclosing function, "pretending" the aborted function returned an error. Depending on the return type of the function, a set of heuristics are employed so as to determine an appropriate error return value that is, in turn, used by the program to handle error conditions. Our underlying assumption is that the remainder of the program can handle truncated data in a buffer in a graceful manner. For example, consider the

case of a buffer overflow vulnerability in a web server, whereby extremely long URLs cause the server to be subverted: when DYBOC catches such an overflow, the web server will simply try to process the truncated URL (which may simply be garbage, or may point to a legitimate page).

A secondary assumption is that most functions that are thusly aborted do not have other side effects (*e.g.,* touch global state), or that such side effects can be ignored. *Obviously, neither of these two conditions can be proven, and examples where they do not hold can be trivially constructed, e.g., an mmap()'ed file shared with other applications.* Since we are interested in the actual behavior of real software, we experimentally evaluate our hypothesis in Section 4.3. Note that, in principle, we could checkpoint and recover from each instruction (line of code) that "touches" a buffer; doing so, however, would be prohibitively expensive.

To implement recovery we use *sigsetjmp()* to snapshot the location to which we want to return once an attack has been detected. The effect of this operation is to save the stack pointers, registers, and program counter, such that the program can later restore their state. We also inject a signal handler (initialized early in *main()*) that catches SIGSEGV[2] and calls *siglongjmp()*, restoring the stack pointers and registers (including the program counter) to their values prior to the call of the offending function (in fact, they are restored to their values as of the call to *sigsetjmp()*):

```
void sigsegv_handler() {
    /* transaction(TRANS_ABORT); */
    siglongjmp(global_env, 1);
}
```

(We explain the meaning of the *transaction()* call later in this section.) The

[2]Care must be taken to avoid an endless loop on the signal handler if another such signal is raised while in the handler. We apply our approach on OpenBSD and Linux RedHat.

program will then re-evaluate the injected conditional statement that includes the *sigsetjmp()* call. This time, however, the return value will cause the conditional to evaluate to false, thereby skipping execution of the offending function. Note that the targeted buffer will contain exactly the amount of data (infection vector) it would if the offending function performed correct data-truncation. In our example, after a fault, execution will return to the conditional statement just prior to the call to *other_func()*, which will cause execution to skip another invocation of *other_func()*. If *other_func()* is a function such as *strcpy(), or sprintf()* (*i.e.,* code with no side effects), the result is similar to a situation where these functions correctly handled array-bounds checking.

There are two benefits to this approach. First, objects in the heap are protected from being overwritten by an attack on the specified variable since there is a signal violation when data is written beyond the allocated space. Second, we can recover gracefully from an overflow attempt, since we can recover the stack context environment prior to the offending function's call, and effectively *siglongjmp()* to the code immediately following the routine that caused the overflow or underflow. While the contents of the stack can be recovered by restoring the stack pointer, special care must be placed in handling the state of the heap. To deal with data corruption in the heap, we can employ data structure consistency constraints, as described by Demsky et al [50], to detect and recover from such errors. Thus, the code in our example from Figure 8.6 will be transformed as shown in Figure 4.2.

To accommodate multiple functions checkpointing different locations during program execution, a globally defined *sigjmp_buf* structure always points to the latest snapshot to recover from. Each function is responsible for saving and restoring this information before and after invoking a subroutine respectively, as shown in Figure 4.3.

Functions may also refer to global variables; ideally, we would like to unroll any

```
int func()
{
  char *buf;
  buf = pmalloc(100);
...
  if (sigsetjmp(global_env, 1) == 0) {
     other_func(buf); /* Indented */
  }
...
  pfree(buf);
  return 0;
}

/* Global definitions */
sigjmp_buf global_env;
```

Figure 4.2: **Saving state for recovery.**

changes made to them by an aborted transaction. The use of such variables can be determined fairly easily via lexical analysis of the instrumented function: any *l-values* not defined in the function are assumed to be global variables (globals used as *r-values* do not cause any changes to their values, and can thus be safely ignored). Once the name of the global variable has been determined, we scan the code to determine its type. If it is a basic type (*e.g.,* integer, float, character), a fixed-size array of a basic type, or a statically allocated structure, we define a temporary variable of the same type in the enclosing function and save/restore its original value as needed. In the example shown in Figure 4.4, variable "global" is used in *other_func()*.

Unfortunately, dynamically allocated global data structures (such as hash tables or linked lists) are not as straightforward to handle in this manner, since their size may be determined at run time and thus be indeterminate to a static lexical analyzer. Thus, when we cannot determine the side-effects of a function, we use a different

```
int func()
{
  char *buf;
  sigjmp_buf curr_env;
  sigjmp_buf *prev_env;
  buf = pmalloc(100);
...
  if (sigsetjmp(curr_env, 1) == 0) {
    prev_env = global_env;
    global_env = &curr_env;
    other_func(buf); /* Indented */
    global_env = prev_env;
  }
...
  pfree(buf);
  return 0;
}
```

Figure 4.3: **Saving previous recovery context.**

mechanism, assisted by the operating system: we added a new system call, named *transaction()*. This is conditionally invoked (as directed by the *dyboc_flag()* macro) at three locations in the code, as shown in Figure 4.4.

First, prior to invoking a function that may be aborted, to indicate to the operating system that a new transaction has begun. The OS makes a backup of all memory page permissions, and marks all heap memory pages as read-only. As the process executes and modifies these pages, the OS maintains a copy of the original page and allocates a new page (which is given the permissions the original page had, from the backup) for the process to use, in exactly the same way copy-on-write works in modern operating systems. Both copies of the page are kept until *transaction()* is called again. Second, after the end of a transaction (execution of a vulnerable function), to indicate to the operating system that a transaction has successfully completed. The OS then discards all original copies of memory pages that have been

```
/* Global variables */
int global;

int func()
{
    char *buf;
    sigjmp_buf curr_env;
    sigjmp_buf *prev_env;
    buf = pmalloc(100);
    int temp_dyboc_global;
...
    if (sigsetjmp(curr_env, 1) == 0) {
      temp_dyboc_global = global;
  /* OR: transaction(TRANS_START); */
      prev_env = global_env;
      global_env = &curr_env;
      other_func(buf); /* Indented */
      global_env = prev_env;
    } else {
      global = temp_dyboc_global;
  /* OR: transaction(TRANS_END); */
    }
...
    pfree(buf);
    return 0;
}
```

Figure 4.4: **Saving global variable.**

```
int func()
{
    char *buf;
    sigjmp_buf curr_env, *prev_env;
    char _buf[100];
    if (dyboc_flag(827))
      buf = pmalloc(100); /* Indented */
    else
      buf = _buf;
...
    if (dyboc_flag(1821)) {
      if (sigsetjmp(curr_env, 1) == 0) {
        prev_env = global_env;
        global_env = &curr_env;
        other_func(buf);
        global_env = prev_env;
      }
    } else {
      other_func(buf);
    }
...
    if (dyboc_flag(827)) {
      pfree(buf); /* Indented */
    }
    return 0;
}
```

Figure 4.5: **Enabling DYBOC conditionally.**

modified during processing this request. Third, in the signal handler, to indicate to the OS that an exception (attack) has been detected. The OS then discards all dirty pages by restoring the original pages.

A similar mechanism could be built around the filesystem by using a private copy of the buffer cache for the process executing in shadow mode, although we have not implemented it. The only difficulty arises when the process must itself communicate with another process while servicing a request; unless the second process is also included in the transaction definition (which may be impossible, if it is a remote process on another system), overall system state may change without the ability to

roll it back. For example, this may happen when a web server communicates with a back-end database. Our system does not currently address this, *i.e.,* we assume that any such state changes are benign or irrelevant (*e.g.,* a DNS query). Back-end databases inherently support the concept of a transaction rollback, so it is (in theory) possible to undo any changes.

The signal handler may also notify external logic to indicate that an attack associated with a particular input from a specific source has been detected. The external logic may then instantiate a filter, either based on the network source of the request or the contents of the payload.

4.2.3 Dynamic Defensive Postures

'Eternal vigilance is the price of liberty.' - Wendell Phillips, 1852

Unfortunately, when it comes to security mechanisms, vigilance takes a back seat to performance. Thus, although our mechanism can defend against all buffer overflow attacks and (as we shall see in Section 4.3) maintains service availability in the majority of cases, this comes at the cost of performance degradation. Although such degradation seems to be modest for some applications (about 20% for Apache, see Section 4.3), it is conceivable that other applications may suffer a significant performance penalty if all buffers are instrumented with our system (for example, a worst-case micro-benchmark measurement indicates a 440% slowdown). One possibility we already mentioned is the use of static analysis tools to reduce the number of buffers that need to be instrumented; however, it is very likely that a significant number of these will remain unresolved, requiring further protection.

Our scheme makes it possible to *selectively* enable or disable protection for specific buffers in functions, in response to external events (*e.g.,* an administrator command,

or an automated intrusion detection system). In the simplest case, an application may execute with all protection disabled, only to assume a more defensive posture as a result of increased network scanning and probing activity. This allows us to avoid paying the cost of instrumentation most of the time, while retaining the ability to protect against attacks quickly. Although this strategy entails some risks (exposure to a successful directed attack with no prior warning), it may be the only alternative when we wish to achieve security, availability, **and** performance.

The basic idea is to only use *pmalloc()* and *pfree()* if a flag instructs the application to do so; otherwise, the transformed buffer is made to point to a statically allocated buffer. Similarly, the *sigsetjmp()* operation is performed only when the relevant flag indicates so. This flagging mechanism is implemented through the *dyboc_flag()* macro, which takes as argument an identifier for the current allocation or checkpoint, and returns true if the appropriate action needs to be taken. Continuing with our previous example, the code will be transformed as shown in Figure 4.5. Note that there are three invocations of *dyboc_flag()*, using two different identifiers: the first and last use the same identifier, which indicates whether a particular buffer should be *pmalloc()*'ed or be statically allocated; the second invocation, with a different identifier, indicates whether a particular transaction (function call) should be checkpointed.

To implement the signaling mechanism, we use a shared memory segment of sufficient size to hold all identifiers (1 bit per flag). *dyboc_flag()* then simply tests the appropriate flag. A second process, acting as the *notification monitor* is responsible for setting the appropriate flag, when notified through a command-line tool or an automated mechanism. Turning off a flag requires manual intervention by the administrator. We not address memory leaks due to the obvious race condition (turning off the flag while a buffer is already allocated), since we currently only examine single threaded cases and we expect the administrator to restart the service under such rare

circumstances, although these can be addressed with additional checking code. Other mechanisms that can be used to address memory leaks and inconsistent data structures are recursive restartability [35] and micro-rebooting [36]. We intend to examine these in future work.

4.3 Evaluation: Execution Transactions

4.3.1 Performance Evaluation

To understand the performance implications of our protection mechanism, we run a set of performance benchmarks. We first measure the worst-case performance impact of DYBOC in a contrived program; we then run DYBOC against the Apache web server and measure the overhead of full protection.

Micro Benchmark

The first benchmark is aimed at helping us understand the performance implications of our DYBOC engine. For this purpose, we use an austere *C* program that makes an *strcpy()* call using a statically allocated buffer as the basis of our experiment.

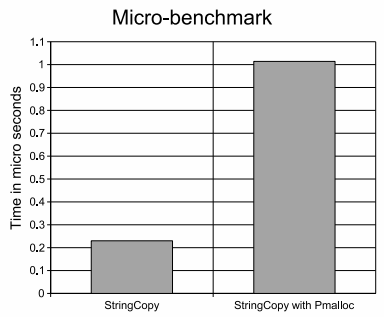


Figure 4.6: **Micro-benchmark results.**

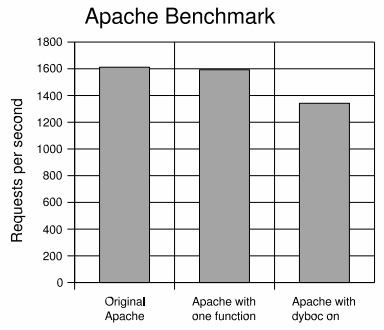


Figure 4.7: **Apache benchmark results.**

After patching the program with DYBOC, we compare the performance of the

patched version to that of the original version by examining the difference in processor cycles using the Read Time Stamp Counter (RDTSC), found in Pentium class processors. The results illustrated by Figure 4.6 indicate the mean time, in microseconds (adjusted from the processor cycles), for 100,000 iterations. The performance overhead for the patched, protected version is 440%, which is expected given the complexity of the *pmalloc()* routine relative to the simplicity of calling *strcpy()* for small strings.

We also used DYBOC on the Apache web server, version 2.0.49. Apache was chosen due to its popularity and source-code availability. Basic Apache functionality was tested, omitting additional modules. Our goal was to examine the overhead of preemptive patching of a software system. The tests were conducted on a PC with a 2GHz Intel P4 processor and 1GB of RAM, running Debian Linux (2.6.5-1 kernel).

We used ApacheBench, a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance.

Figure 8.7 illustrates the requests per second that Apache can handle. There is a 20.1% overhead for the patched version of Apache over the original, which is expected since the majority of the patched buffers belong to utility functions that are not heavily used. This result is an indication of the worst-case analysis, since all the protection flags were enabled; although the performance penalty is high, it is not outright prohibitive for some applications. For the instrumentation of a single buffer and a vulnerable function that is invoked once per HTTP transaction, the overhead is 1.18%.

Space Overheads

The line count for the server files in Apache is 226,647, while the patched version is 258,061 lines long, representing an increase of 13.86%. Note that buffers that are already being allocated with *malloc()* (and de-allocated with *free()*) are simply translated to *pmalloc()* and *pfree()* respectively, and thus do not contribute to an increase in the line count. The binary size of the original version was 2,231,922 bytes, while the patched version of the binary was 2,259,243 bytes, an increase of 1.22%. Similar results are obtained with OpenSSH 3.7.1. Thus, the impact of our approach in terms of additional required memory or disk storage is minimal.

Memory Overheads

Of further interest is the increase in memory requirements for the patched version. A naive implementation of *pmalloc()* would require two additional memory pages for each transformed buffer. Full transformation of Apache translates into 297 buffers that are allocated with *pmalloc()*, adding an overhead of 2.3MB if all of these buffers are invoked simultaneously during program execution. When protecting *malloc()*'ed buffers, the amount of required memory can skyrocket. To avoid this overhead, we use an *mmap()* based allocator. The two guard pages are *mmap*'ed write-protected from */dev/zero,* without requiring additional physical memory to be allocated. Instead, the overhead of our mechanism is 2 page-table entries (PTEs) per allocated buffer, plus one file descriptor (for */dev/zero*) per program. As most modern processors use an MMU cache for frequently used PTEs, and since the guard pages are only accessed when a fault occurs, we expect a small impact on performance.

4.3.2 Effectiveness as a worm containment strategy

In order to analyze the effectiveness of our technique as a worm containment strategy, we examine its effects on worm propagation. In particular, we model the impact of our approach on a scanning worm with propagation characteristics similar to Code Red. Worm containment is comprised of three steps: worm detection, patch generation, and patch dissemination. In this work, we address the effects of detection and patch generation on worm propagation.

Modelling worm propagation has been addressed extensively in the literature [137, 81, 134, 142] and is often described using variations on the susceptible-infected-susceptible (SIS) and susceptible-infected-recovered (SIR) models borrowed from classic epidemiology research. The SIR model is more suitable for worm modelling since it takes into account recovered (patched) systems. We model our system as a variation to the SIR model where infected hosts that are protected using DYBOC recover after the time period required to detect a vulnerability and patch a system. Specifically, we use a model similar to the one proposed by Wong et al [137] where the population size is 10000 and the probability of selecting a host out of the network's address space is 10000/65535.

As illustrated in figure 4.3.2, we examine the effects of using DYBOC on worm propagation by varying the percentage of susceptible hosts that are protected with our tool. Specifically, we vary the percentage of DYBOC protected hosts from 0% to 20% and inspect the effects on overall infected population and the rate of infection. When a worm infects a node that is protected with DYBOC, the node continues as an infected host for the time required to generate and test a patch, effectively participating in worm propagation during this period. We model this period as a random value between one and three minutes. After this period, the node joins the

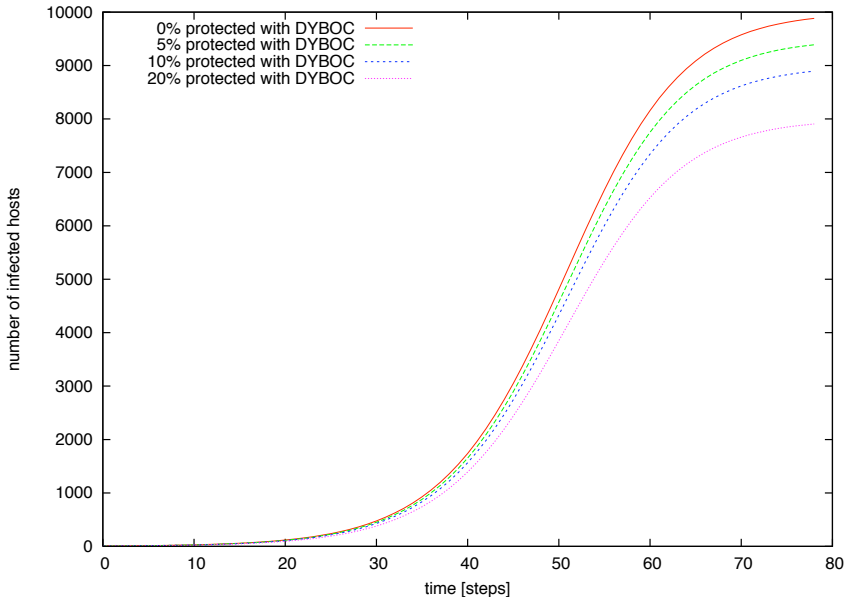


Figure 4.8: Simulating the effects of using DYBOC on worm propagation. We vary the percentage of protected hosts and examine the effects on infection rate and overall susceptible population.

group of patched (recovered) hosts.

Upon first glance, it is evident that although DYBOC mitigates a worm's effect on the susceptible population, protection of individual hosts will not suffice as an attempt at curtailing worm propagation (unless a very large portion of the population is protected) without further collaboration between hosts. Collaboration techniques may include patch dissemination and collaborative filtering. Patch dissemination techniques have been addressed in [43] and analyzed by Vojnovic and Ganesh [81] where the authors show that given significant participation and swift patch generation a worm's propagation rate can be restricted if the patch spreading rate is higher than the worm's infection rate. In more detail, they explore patch dissemination using a hierarchical system where hosts responsible for regions can also act as filters (or firewalls). Nicol et al [86] also examine models of active worm defenses; they investigate the effects of filtering in conjunction with patch dissemination. Unfortunatelly,

their analysis only holds for scanning worms. Very fast worms (flash, hitlist) tend to have infection rates that are as high, if not higher, as patch dissemination rates. Furthermore, fast scanning worms do not follow traditional epidemic models since their propagation is often shaped by the affect they have on the underlying network infrastructure.

Chapter 5

STEM: Selective Transactional EMulation

In this Chapter, we discuss the design and implementation of STEM (Selective Transactional EMulation). STEM is a binary supervision framework that holistically enables software self-healing by including a variety of protection, detection and repair strategies. By enabling selective program emulation, STEM allows speculative execution of program"slices" whose effects can be rolled back (system level undo) in case of a fault (or detected attack).

5.1 Introduction

In Chapter 4, we discussed the design and implementation of our first self-healing tool, DYBOC. DYBOC was a great first tool for exploring self-healing systems in general and error virtualization in particular. Unfortunately, it had some limitations:

- Limited to memory violations.
- Used the rather limited setjmp/longjmp mechanism for enforcing transactions.

- Required juggling in order to deal with rolling back global memory

Addressing the limitations associated with DYBOC, led to the introduction Selective Transactional EMulation (*STEM*), an instruction-level emulator that can be selectively invoked for arbitrary segments of code, allowing us to mix emulated and non-emulated execution inside the same process. The emulator allows us to (*a*) monitor for the specific type of failure prior to executing the instruction, (*b*) undo any memory changes made by the function inside which the fault occurred, by having the emulator record all memory modifications made during its execution, and (*c*) simulate an error-return from said function.

5.2 Approach

Our architecture, depicted in Figure 8.2, uses three types of components: a set of sensors that monitor an application (such as a web server) for faults; Selective Transactional EMulation (*STEM*), an instruction-level emulator that can selectively emulate "slices" (arbitrary segments) of code; and a testing environment where hypotheses about the effect of various fixes are evaluated. These components can operate without human supervision to minimize reaction time.

5.3 System Overview

When the sensor detects an error in the application's execution (such as a segmentation fault), the system instruments the portion of the application's code that immediately surrounds the faulty instruction(s), such that the code segment is emulated (the mechanics of this are explained in Section 8.4). To verify the effectiveness of the fix, the application is restarted in a test environment with the instrumentation

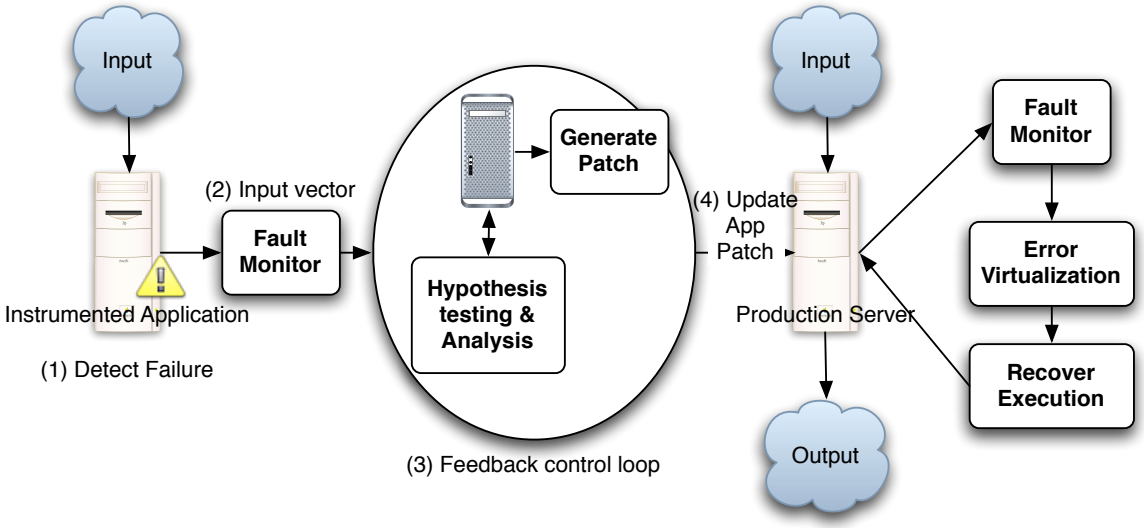


Figure 5.1: Feedback control loop: (1) a variety of sensors monitor the application for known types (but unknown instances) of faults; (2) upon recognizing a fault, we emulate the region of code where the fault occurred and test with the inputs seen before the fault occurred; (3) by varying the scope of emulation, we can determine the "narrowest" code slice we can emulate and still detect and recover from the fault; (4) we then update the production version of the server.

enabled, and is supplied with the input that caused the failure (or the N most recent inputs, if the offending one cannot be easily identified, where N is a configurable parameter). We focus on server type applications that have a transactional processing model, because it is easier to quickly correlate perceived failures with a small or finite set of inputs than with other types of applications (*e.g.,* those with a GUI).

During emulation, *STEM* maintains a record of all memory changes (including global variables or library-internal state, *e.g., libc* standard I/O structures) that the emulated code makes, along with their original values. Furthermore, *STEM* examines the operands for each machine instruction and pre-determines the side effects of the instructions it emulates. The use of an emulator allows us to circumvent the complexity of code analysis, as we only need to focus on the operation and side effects

of individual instructions independently from each other.

If the emulator determines that a fault is about to occur, the emulated execution is aborted. Specifically, all memory changes made by the emulated code are undone, and the currently executing function is "forced" to return an error. We describe how both emulation and error virtualization are accomplished in Sections 5.5 and 5.6, respectively, and we experimentally validate the error virtualization hypothesis in Section 5.9. For our initial approach, we are primarily concerned with failures where there is a one-to-one correspondence between inputs and failures, and not with those that are caused by a combination of inputs. Note, however, that many of the latter type of failures are in fact addressed by our system, because the last input (and code leading to a failure) will be recognized as "problematic" and handled as we have discussed.

In the testing and error localization phase, emulation stops after forcing the function to return. If the program crashes, the scope of the emulation is expanded to include the parent (calling) routine and the application re-executes with the same inputs. This process is repeated until the application does not terminate after we abort a function calls sequence. In the extreme case, the whole application could end up being emulated, at a significant performance cost. However, Section 5.9 shows that this failsafe measure is rarely necessary.

If the program does not crash after the forced return, we have found a "vaccine" for the fault, which we can use on the production server. Naturally, if the fault is not triggered during an emulated execution, emulation halts at the end of the vulnerable code segment, and all memory changes become permanent.

The overhead of emulation is incurred at all times (whether the fault is triggered or not). To minimize this cost, we must identify the smallest piece of code that we need emulate in order to catch and recover from the fault. We currently treat

functions as discrete entities and emulate the whole body of a function, even though the emulator allows us to start and stop emulation at arbitrary points, as described in Section 8.4. Future work will explore strategies for minimizing the scope of the emulation and balancing the tradeoff between coverage and performance.

In the remainder of this section, we describe the types of sensors we employ, give an overview of how the emulator operates (with more details on the implementation in Section 8.4), and describe how the emulator forces a function to return with an error code. We also discuss the limitations of reactive approaches in general and our system in particular.

5.4 Application Monitors

The selection of appropriate failure-detection sensors depends on both the nature of the flaws themselves and tolerance of their impact on system performance. We describe the two types of application monitors that we have experimented with.

The first approach is straightforward. The operating system forces a misbehaving application to abort and creates a core dump file that includes the type of failure and the stack trace when that failure occurred. This information is sufficient for our system to apply selective emulation, starting with the top-most function in the stack trace. Thus, we only need a watchdog process that waits until the service terminates before it invokes our system.

A second approach is to use an appropriately instrumented version of the application on a separate server as a honeypot, as we demonstrated for the case of network worms [109]. Under this scheme, we instrument the parts of the application that may be vulnerable to a particular class of attack (in this case, remotely exploitable buffer overflows) such that an attempt to exploit a new vulnerability exposes the attack vec-

tor and all pertinent information (attacked buffer, vulnerable function, stack trace, *etc.*).

This information is then used to construct an emulator-based vaccine that effectively implements array bounds checking at the machine-instruction level. This approach has great potential in catching new vulnerabilities that are being indiscriminately attempted at high volume, as may be the case with an "auto-root" kit or a fast-spreading worm. Since the honeypot is not in the production server's critical path, its performance is not a primary concern (assuming that attacks are relatively rare phenomena). In the extreme case, we can construct a honeypot using our instruction-level emulator to execute the whole application, although we do not further explore this possibility in this paper.

5.5 Selective Transactional EMulation (STEM)

The recovery mechanism uses an instruction-level emulator, *STEM*, that can be selectively invoked for arbitrary segments of code. This tool permits the execution of emulated and non-emulated code inside the same process. The emulator is implemented as a *C* library that defines special tags (a combination of macros and function calls) that mark the beginning and the end of selective emulation. To use the emulator, we can either link it with an application in advance, or compile it in the code in response to a detected failure, as was done in [109].

Upon entering the vulnerable section of code, the emulator snapshots the program state and executes all instructions on the virtual processor. When the program counter references the first instruction outside the bounds of emulation, the virtual processor copies its internal state back to the real CPU, and lets the program continue execution natively. While registers are explicitly updated, memory updates have

implicitly been applied throughout the execution of the emulation. The program, unaware of the instructions executed by the emulator, continues executing directly on the CPU.

To implement fault catching, the emulator simply checks the operands of instructions it is emulating, taking into consideration additional information supplied by the sensor that detected the fault. For example, in the case of division by zero, the emulator need only check the value of the appropriate operand to the *div* instruction. For illegal memory dereferencing, the emulator verifies whether the source or destination addresses of any memory access (or the program counter, for instruction fetches) point to a page that is mapped to the process address space using the *mincore()* system call. Buffer overflow detection is handled by padding the memory surrounding the vulnerable buffer, as identified by the sensor, by one byte, similar to the way StackGuard [46] operates. The emulator then simply watches for memory writes to these memory locations. This approach requires source code availability, so as to insert the "canary" variables. Contrary to StackGuard, our approach allows us to stop the overflow before it overwrites the rest of the stack, and thus to recover the execution. For algorithmic-complexity denial of service attacks, such as the one described in [47], we keep track of the amount of time (in terms of number of instructions) we execute in the instrumented code; if this exceeds a pre-defined threshold, we abort the execution. This threshold may be defined manually, or can be determined by profiling the application under real (or realistic) workloads, although we have not fully explored the possibilities.

We currently assume that the emulator is pre-linked with the vulnerable application, or that the source code of that application is available. It is possible to circumvent this limitation by using the CPU's programmable breakpoint register (in much the same way that a debugger uses it to capture execution at particular points

in the program) to invoke the emulator without the running process even being able to detect that it is now running under an emulator.

5.6 Recovery: Forcing Error Returns

Upon detecting a fault, our recovery mechanism undoes all memory changes and forces an error return from the currently executing function. To determine the appropriate error return value, we analyze the declared type of the function.

Depending on the return type of the emulated function, the system returns an "appropriate" value. This value is determined based on some straightforward heuristics and is placed in the stack frame of the returning function. The emulator then transfers control back to the calling function. For example, if the return type is an *int,* a value of -1 is returned; if the value is *unsigned int* the system returns 0, *etc.* A special case is used when the function returns a pointer. Instead of blindly returning a *NULL*, we examine if the returned pointer is further dereferenced by the parent function. If so, we expand the scope of the emulation to include the parent function. We handle value-return function arguments similarly. There are some contexts where this heuristic may not work well; however, as a first approach these heuristics worked extremely well in our experiments (see Section 5.9).

In the future, we plan to use more aggressive source code analysis techniques to determine the return values that are appropriate for a function. Since in many cases a common error-code convention is used in large applications or modules, it may be possible to ask the programmer to provide a short description of this convention as input to our system either through code annotations or as separate input. A similar approach can be used to mark functions that must be fail-safe and should return a specific value when an error return is forced, *e.g.,* code that checks user permissions.

5.7 Caveats and Limitations

While promising, reactive approaches to software faults face a new set of challenges. As this is a relatively unexplored field, some problems are beyond the scope of this paper.

First, our primary goal is to evolve an application protected by STEM towards a state that is highly resistant to exploits and errors. While we expect the downtime for such a system to be reduced, we do not reasonably expect zero downtime. STEM fundamentally relies on the application monitors detecting an error or attack, stopping the application, marking the affected sections for emulated execution, and then restarting the application. This process necessarily involves downtime, but is incurred only once for each detected vulnerability. We believe that combining our approach with microrebooting techniques can streamline this process.

A reaction system must evaluate and choose a response from a wide array of choices. Currently, when encountering a fault, a system can (a) crash, (b) crash and be restarted by a monitor [36], (c) return arbitrary values [97], or (d) slice off the functionality. Most proactive systems take the first approach. We elect to take the last approach. As Section 5.6 shows, this choice seems to work extremely well. This phenomenon also appears at the machine instruction level [133].

However, there is a fundamental problem in choosing a particular response. Since the high-level behavior of any system cannot be algorithmically determined, the system must be careful to avoid cases where the response would take execution down a semantically (from the viewpoint of the programmer's intent) incorrect path. An example of this type of problem is skipping a check in *sshd* which would allow an otherwise unauthenticated user to gain access to the system. The exploration of ways to bound these types of errors is an open area of research. Our initial approach is to

rely on the programmer to provide annotations as to which parts of the code should not be circumvented.

There is a key tradeoff between code coverage (and thus confidence in the level of security the system provides) and performance (processing and memory overhead). Our emulator implementation is a proof of concept; many enhancements are possible to increase performance in a production system. Our main goal is to emphasize the service that such an emulator will provide: the ability to selectively incur the cost of emulation for vulnerable program code only. Our system is directed to these vulnerable sections by runtime sensors – the quality of the application monitors dictates the quality of the code coverage.

Since our emulator is designed to operate at the user level, it hands control to the operating system during system calls. If a fault were to occur in the operating system, our system would not be able to react to it. In a related problem, I/O beyond the machine presents a problem for a rollback strategy. This problem can partially be addressed by the approach taken in [67], by having the application monitors log outgoing data and implementing a callback mechanism for the receiving process.

Finally, in our current work, we assume that the source code of the vulnerable application is available to our system. We briefly discussed how to partially circumvent this limitation in Section 5.5. Additional work is needed to enable our system to work in a binary-only environment.

5.8 Implementation

We implemented the *STEM x86* emulator to validate the practicality of providing a supervision framework for the feedback control loop through selective emulation of code slices. Integrating *STEM* into an existing application is straightforward. As

shown in Figure 5.2, four special tags are wrapped around the segment of code that will be emulated.

```
void foo() {
    int a = 1;
    emulate_init();
    emulate_begin(stem_args);
    a++;
    emulate_end();
    emulate_term();
    printf("a = %d\n", a);
}
```

Figure 5.2: A trivial example of using *STEM*. The emulate_* calls invoke and terminate execution of *STEM*. The code inside that region is executed by the emulator. In order to illustrate the level of granularity that we can achieve, we show only the increment statement as being executed by the emulator.

The *C* macro *emulate_init()* moves the program state (general, segment, eflags, and FPU registers) into an emulator-accessible global data structure to capture state immediately before *STEM* takes control. The data structure is used to initialize the virtual registers. With the preliminary setup completed, *emulate_begin()* only needs to obtain the memory location of the first instruction following the call to itself. The instruction address is the same as the return address and can be found in the activation record of *emulate_begin()*, four bytes above its base stack pointer.

The fetch/decode/execute/retire cycle of instructions continues until either *emulate_end()* is reached, or when the emulator detects that control is returning to the parent function. If the emulator does not encounter an error during its execution, the emulator's instruction pointer references the *emulate_term()* macro at completion. To enable the program to continue execution at this address, the return address of the *emulate_begin* activation record is replaced with the current value of the instruction pointer. By executing *emulate_term()*, the emulator's environment is copied to the

program registers and execution continues under normal conditions.

If an exception occurs during emulation, *STEM* locates *emulate_end()* and terminates. Because the emulator saved the state of the program before starting, it can effectively return the program state to its original setting, thus nullifying the effect of the instructions processed through emulation. Essentially, the emulated code is sliced off. At this point, the execution of the code (and its side effects in terms of changes to memory) has been rolled back.

The emulator is designed to execute in user-mode, so system calls cannot be computed directly without kernel-level permissions. Therefore, when the emulator decodes an interruption with an immediate value of $0x80$, it must release control to the kernel. However, before the kernel can successfully execute the system call, the program state needs to reflect the virtual registers arrived at by *STEM*. Thus, the emulator backs up the real registers and replaces them with its own values. An INT $0x80$ is issued by *STEM*, and the kernel processes the system call. Once control returns to the user-level code, the emulator updates its registers and restores the original values in the program's registers.

Jump instructions that move the instruction pointer outside the block of emulated instructions introduced another interesting scenario. There is no way of knowing if control will return to the emulated section of the function or not. Therefore, simply terminating the emulator will not suffice. There are two situations when this can happen. First, a return statement in *C* actually translates to a *jmp* instruction (see Figure 5.3). The *jmp* moves the instruction pointer to the end of the function immediately preceding *leave* and *ret*. If data is returned the registers are set. Second, optimization flags in the *gcc* compiler often shuffle instructions around. In both cases, the emulator continues interpreting instructions until either control returns to the emulated block, or the end of the function (the one embedded within *STEM*) is

reached. In the latter case, emulation is terminated, program state is updated, and the return address points to the *leave* instruction rather than *emulate_term()*. The program then handles the return from the function. For this reason, the instruction address that calls *emulate_end()* must be known.

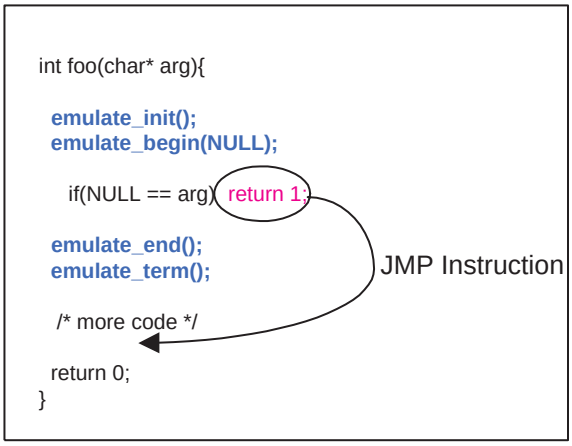


Figure 5.3: Return from within emulation.

The floating point unit (FPU) added an extra level of complexity to the emulator. We investigated the approaches of both Bochs and Valgrind. Bochs integrates a third-party *C* library that implements a virtual FPU. Functional overrides and special hooks were required to link the two modules. Interestingly enough, Valgrind simply passes control to the real FPU for processing. After careful consideration, we implemented a similar method because it provided us quicker turnaround time without impacting security. However, future versions could utilize the approach taken by Bochs.

Stack management was the most intriguing problem. If the stack was maintained in the same location as the program's %esp register, both the emulator and the emulated program would use the same memory region, resulting in a corrupted stack. Thus, we initially relocated the emulated program's stack in the heap. Keeping the virtual stack in the heap worked until we determined that some *libc* routines

branched based on the value of the virtual stack pointer. For example, threaded programs identify children by the location of the stack. However, since threading was not actually used, this caused memory access violations.

We then realized that the stack needed to remain in the same address space at initialization, but the activation records of the emulator's internal functions that are stored on the stack posed the problem. To remove these activation records, we attempted to inline all functions. This turned out to be impractical, because the following restrictions apply in order to inline a function:

1. the function cannot be referenced by its address (*e.g.,* function pointers)

2. switch statements are not permitted

The fetch/decode/execute loop depended on both function pointers to the x86 simulated instructions and switch statements. Even with these constructs replaced with alternative constructs, the *gcc* compiler requires the use of "-O3" option to inline a function, which adds an extra level of complexity during compilation. Compiling a large number of functions that inline other functions required enormous amounts of system resources. For example, after compiling the emulator library for twenty minutes, the CPU resource usage was greater than 80% and memory consumption reached 500 MB on a modern Linux PC. Inline functions, therefore, could not be justified. Another way to prevent activation records in *C* is to macro-tize the functions. However, this method could not be justified due to the complexity of maintaining the code.

5.9 Evaluation

Our description of the system raises several questions that need to be answered in order to determine the tradeoffs between effectiveness, practicality, and performance.

1. Can the system detect *real* attacks and faults and react to them (Chapter 7)?

2. How effective is our "error virtualization" hypothesis as a recovery mechanism ? Does it work for *real* software (Chapter 7)?

3. What is the performance impact of emulation, and what is the gain to be had by using selective emulation ?

In the rest of this section, we provide some preliminary experimental evidence that our system offers a reasonable and *adjustable* tradeoff between the three parameters mentioned above. Naturally, it is impossible to completely cover the space of reactive mechanisms (even within the more limited context of our specific work). Future work is needed to analyze the semantics of error virtualization and the impact that STEM has on the security properties of STEM-enabled applications. As noted below, we plan to construct a correctness testing framework. However, we believe that our results show that such an approach can work and that additional work is needed to fully explore its capabilities and limitations.

5.9.1 Performance

We next turned our attention to the performance impact of our system. In particular, we measured the overhead imposed by the emulator component. *STEM* is meant to be a lightweight mechanism for executing selected portions of an application's code. We can select these code slices according to a number of strategies, as we discussed in Section 5.4.

We evaluated the performance impact of *STEM* by instrumenting the Apache 2.0.49 web server and OpenSSH *sshd,* as well as performing micro-benchmarks on various shell utilities such as *ls*, *cat*, and *cp*.

Testing Environment

The machine we chose to host Apache was a single Pentium III at 1GHz with 512MB of memory running RedHat Linux with kernel 2.4.20. The machine was under a light load during testing (standard set of background applications and an X11 server). The client machine was a dual Pentium II at 350 MHz with 256MB of memory running RedHat Linux 8.0 with kernel 2.4.18smp. The client machine was running a light load (X11 server, *sshd,* background applications) in addition to the test tool. Both emulated and non-emulated versions of Apache were compiled with the *–enable-static-support* configuration option. Finally, the standard runtime configuration for Apache 2.0.49 was used; the only change we made was to enable the *server-status* module (which is compiled in by default but not enabled in the default configuration). *STEM* was compiled with the "*-g -static -fno-defer-pop*" flags. In order to simplify our debugging efforts, we did not include optimization.

We chose the Apache *flood httpd* testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in Figures 8.16 and 5.5. The value for total number of requests per second is extrapolated (by *flood*'s reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not be interpreted to mean that our test Apache instances and our test hardware actually served some 6000 requests per second.

Emulation of Apache Inside Valgrind

To get a sense of the performance degradation imposed by running the entire system inside an emulator other than *STEM*, we tested Apache running in Valgrind version 2.0.0 on the Linux test machine that hosted Apache for our *STEM* test trials.

Valgrind has two notable features that improve performance over our full emulation of the main request loop. First, Valgrind maintains a 14 MB cache of translated instructions which are executed natively after the first time they are emulated, while *STEM* always translates each encountered instruction. Second, Valgrind performs some internal optimizations to avoid redundant load, store, and register-to-register move operations.

We ran Apache under Valgrind with the default skin *Memcheck* and tracing all children processes. While Valgrind performed better than our emulation of the full request processing loop, it did not perform as well as our emulated slices, as shown in Figure 8.16 and the timing performance in Table 8.6.

Finally, the Valgrind–ized version of Apache is 10 times the size of the regular Apache image, while Apache with *STEM* is not noticeably larger.

Full Emulation and Baseline Performance

We demonstrate that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the main request processing loop for Apache (contained in *ap_process_http_connection()*) and compared our results against a non-emulated Apache instance. In this experiment, the emulator executed roughly 213,000 instructions. The impact on performance is clearly seen in Figure 8.16 and further elucidated in Figure 5.5, which plots the performance of the fully emulated request-handling procedure.

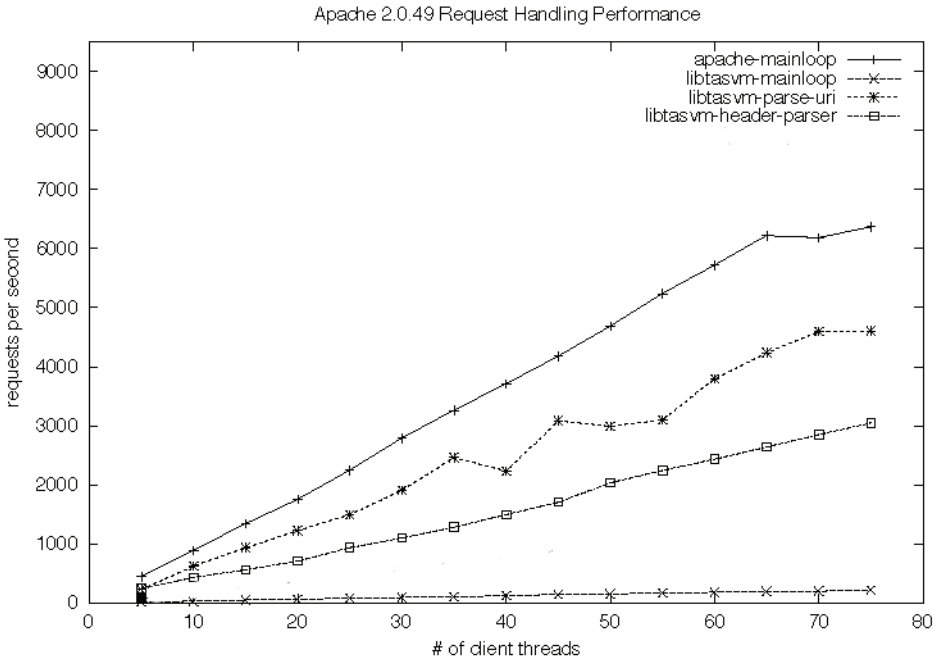


Figure 5.4: Performance of the system under various levels of emulation. This data set includes Valgrind for reference. While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable. Valgrind runs better than *STEM* when executing the entire request loop. As expected, selective emulation still performs better than Valgrind.

In order to get a more complete sense of this performance impact, we timed the execution of the request handling procedure for both the non-emulated and fully-emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked.

For our test machines and sample loads, Apache normally (*e.g.,* non-emulated) spent 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Table 8.6. The fully instrumented loop running in the emulator spends an average of 278 milliseconds per request in that particular code section. For comparison, we also timed Valgrind's execution of this section of code; after a large initial cost (to perform the initial translation and fill the internal instruction cache)

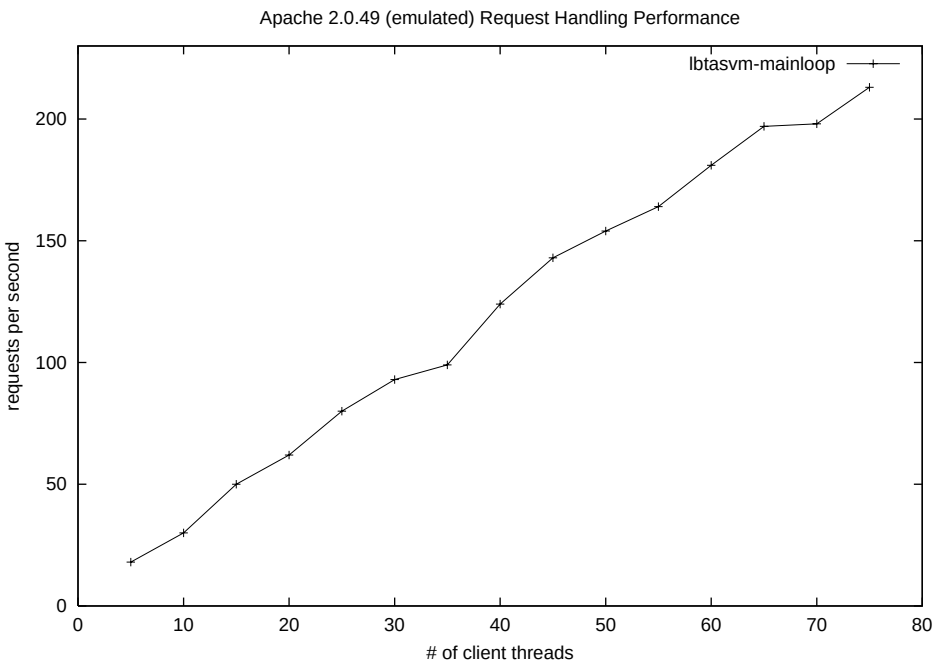


Figure 5.5: A closer look at the performance for the fully emulated version of main processing loop. While there is a considerable performance impact compared to the non-emulated request handling loop, the emulator appears to scale at the characteristic linear rate, indicating that it does not create additional overhead beyond the cost of emulation.

Valgrind executes the section with a 34 millisecond average. These initial costs sometimes exceeded one or two seconds; we ignore them in our data and measure Valgrind only after it has stabilized.

Apache	trials	Mean	Std. Dev.
Normal	18	6314	847
STEM	18	277927	74488
Valgrind	18	34192	11204

Table 5.1: Timing of main request processing loop. Times are in microseconds. This table shows the overhead of running the whole primary request handling mechanism inside the emulator. In each trial a user thread issued an HTTP GET request.

Selective Emulation

In order to identify possible vulnerable sections of code in Apache 2.0.49, we used the RATS tool. The tool identified roughly 270 candidate lines of code, the majority of which contained fixed size local buffers. We then correlated the entries on the list with code that was in the primary execution path of the request processing loop. The two functions that are measured perform work on input that is under client control, and are thus likely candidates for attack vectors.

The main request handling logic in Apache 2.0.49 begins in the *ap_process_http_connection()* function. The effective work of this function is carried out by two subroutines: *ap_read_request()* and *ap_process_request()*. The *ap_process_request()* function is where Apache spends most of its time during the handling of a particular request. In contrast, the *ap_read_request()* function accounts for a smaller fraction of the request handling work. We chose to emulate subroutines of each function in order to assess the impact of selective emulation.

We constructed a partial call tree and chose the *ap_parse_uri()* function (invoked via *read_request_line()* in *ap_read_request()*) and the *ap_run_header_parser()* function (invoked via *ap_process_request_internal()* in *ap_process_request()*). The emulator processed approximately 358 and 3229 instructions, respectively, for these two functions. In each case, the performance impact, as expected, was much less than the overhead incurred by needlessly emulating the entire work of the request processing loop.

Microbenchmarks

Using the client machine from the Apache performance tests, we ran a number of micro-benchmarks to gain a broader view of the performance impact of *STEM*. We selected some common shell utilities and measured their performance for large work-

loads running both with and without *STEM*.

For example, we issued an '*ls -R*' command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). Table 5.2 shows the result of these measurements.

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

Table 5.2: **Microbenchmark performance times for various command line utilities.**

As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.

Chapter 6

ASSURE: Autonomic Software Self-Healing Using Error Virtualization Rescue Points

In this Chapter, we introduce the logical progression to error virtualization: ASSURE or error virtualization using rescue points.

6.1 Introduction

Despite promising results from our error virtualization work, several issues remain open. First, the blind use of recovery heuristics does not inspire confidence in terms of maintaining program semantics. Although we were able to achieve program recovery in about 80% of the examined cases, our goal is to maximize chances of recovery. The logical progression from the use of heuristics seems to be the use of application-specific knowledge. Second, our previous mechanism uses a fire-and-forget approach to recovery. This technique further undermines the inherently unsafe approach we

use towards program recovery. We need some form of assurance that the program execution path initiated by our recovery mechanism does not break program semantics.

To address some of the issues with our error virtualization technique, we present a new technique for retrofitting legacy applications with exception handling techniques, which we call *Autonomic Software Self-Healing Using Error Virtualization Rescue Points* (ASSURE). ASSURE is a general software fault-recovery mechanism that uses operating system virtualization techniques to provide "rescue points" that an application can use to recover execution to, in the presence of faults. When a fault occurs at an arbitrary location in the program, we restore program execution to a "rescue point" and imitate its observed behavior to propagate errors and recover execution.

The use of rescue points reduces the chance of unanticipated execution paths, thereby making recovery more robust, by mimicking system behavior under controlled error conditions. These controlled error conditions can be thought of as a set erroneous inputs, like the ones used by most quality assurance teams during software development, designed to stress test an application. To discover "rescue points", applications are profiled and monitored during tests that bombard the program with "bad input". The intuition is that by monitoring application behavior during these runs, we gain insight into how programmer-tested program points are used to propagate faults gracefully.

The key difference between this work and previous techniques [112, 108, 97], that try to be oblivious to occurrence of faults, is the kind of guarantees one can make on program semantics. Rescue points do not try to mask errors in a demonstration of blind faith. In fact, rescue points, force the exact opposite behavior: they induce faults at locations that are *known* to propagate faults correctly.

Another distinction of this work is the decoupling of the fault detection component

and the recovery mechanism, allowing the use of a variety of fault-detection mechanisms. Decoupling fault detection and the recovery mechanism has some additional fringe benefits, namely, performance and completeness. Since the detection mechanism is not tied to the recovery mechanism, it can be applied to a smaller scope of the application reducing the performance overhead imposed by expensive protection mechanisms. For example, in [112], the function became the unit of transaction that could be aborted, which causes two problems: if the function performs a lot of work, protecting the function becomes prohibitively expensive; if the function could not be aborted without side-effects, the scope of emulation had to be expanded to include the calling function, further degrading performance.

The crux of our approach revolves around an Observe Orient Decide Act (OODA) feedback loop [26]. First, we profile programs during erroneous test runs in order to build a behavioral model for the application. Using this model, we discover candidate rescue points. We then use a set of software probes that monitor the application for specific types of faults. Upon detection of a fault, an operating system recovery mechanism is invoked that allows the application to rollback application state to a rescue point and replay execution pretending that an error has occurred. Using continuous hypothesis testing, we confirm that our "action" has repaired the fault by re-running the application against the event sequence that apparently caused the failure. Our focus is on automatic healing of services against newly detected faults (whether accidental failures or attacks). We emphasize that we seek to address a wide variety of software failures, not just attacks.

Our proposed work focuses on server-type applications for two reasons: they typically have higher availability requirements than user-oriented applications, and they tend to have short error-propagation distances [97] *i.e.,* an error that might occur during the processing of a request has little or no impact on the service of future

requests. To provide an analogy, if one considers an organization like NASA, history has shown that it would not make sense to use a mechanism like error virtualization on a software system that calculates space-travel trajectories since the correctness of the results cannot be guaranteed. However, the software on the Mars Rover would benefit greatly from a technique that allows for continued execution in the presence of faults [104].

ASSURE builds upon and improves previous techniques that attempt to tolerate faults in several ways:

- The use of rescue points reduces the chance of unanticipated execution paths by mimicking system behavior under controlled error conditions.
- Error virtualization is applicable to a wider range of faults, such as algorithmic-complexity denial-of-service (DoS) attacks [47].
- Better performance: Through the use of a low overhead copy-on-write checkpoint mechanism, fault monitoring can be localized and decoupled from recovery, allowing us to reduce the scope, and thus the cost, of monitoring vulnerable segments.
- The ability to generalize on the effects of our recovery technique on software survivability through the use of fault injection. Past techniques have only demonstrated their effectiveness against a limited set of memory errors.
- Ability to run multi-threaded applications.

To evaluate the effectiveness of our system and its performance impact on regular system operation, we conducted a series of experiments on several open-source server applications. First, we examined the effectiveness of error virtualization using rescue

points by injecting faults into several applications and monitoring system behavior and demonstrated that rescue points could be used to recover from these faults and successfully in all of the cases. Second, we explored the performance implications of our system showing that using rescue points adds as little as 1.5% overhead to normal execution and recovery from faults happens in a few milliseconds.

6.2 ASSURE Operational Overview

ASSURE provides architectural support for application self-healing in the presence of unanticipated faults in a fully automated manner. The system continuously monitors the application for failures and identifies strategies for reacting to future occurrences of the same or similar failures. Once a strategy is selected, ASSURE dynamically modifies the application, using dynamic binary injection, so that it is able to detect and recover from the same fault in the future. The objective of our system is to automatically create a temporary fix for a particular problem until a vendor's solution is made available.

ASSURE enables self-healing through a combination of operating system virtualization techniques, namely, checkpoint-rollback functionality to create rescue points, and error virtualization, a technique that maps previously unseen faults to ones that are explicitly handled by the application code.

Figure 6.1 illustrates the high-level operation of ASSURE. Prior to its deployment, the application is profiled in order to discover candidate rescue points. After the profiling completes, the application is deployed in its production environment. During normal execution, ASSURE monitors the application with a variety of light-weight instrumentation mechanisms that facilitate the detection and reporting of application and system misbehavior. In addition, the system takes periodic checkpoints of the

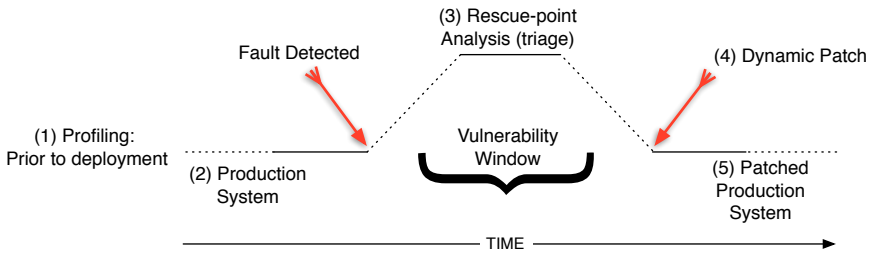


Figure 6.1: System overview: (1) Applications are profiled prior to deployment using "bad" input to create the set of potential rescue points for the applications. In the production system, (2) light-weight sensors monitor the application for faults and misbehavior. (3) Once a fault is detected, the system uses a shadow deployment of the application to find rescue points that will allow the application to recover from the fault in the future, and to test rescue points patches for correctness. Once a satisfactory rescue point is found, (4) a dynamic patch is generated to update the production system. (5) The production system is now able to detect and recover from the particular fault using error virtualization.

application state and maintains a log of incoming network traffic.

When a fault is detected during execution, the latest application checkpoint state along with the log of all the inputs since that checkpoint is transferred to a triage system, a shadow deployment of the application, where the fault is analyzed. ASSURE then carries out an automated process whose goal is to identify a suitable rescue point to which the application can recover execution should that particular fault re-manifest. During this time, the production system remains vulnerable to re-occurrences of the fault, resulting in a vulnerability window. While our system may require some downtime for the analysis phase, the cost is amortized as it is incurred once per new fault. Combining our approach with techniques such as micro-rebooting [34, 36] is a topic of future research.

Once a rescue point is selected, ASSURE confirms that it is suitable for deployment by verifying that it satisfies two criteria: survivability and correctness. A selected rescue point provides *survivability* if error virtualization at that point enables the application to survive a recurrence of the fault. A rescue point is *correct* if it

does not introduce semantic errors, and if the application can service future requests correctly. Error virtualization is verified by replaying the sequence of events that apparently triggered the fault, while semantic equivalence and correctness are verified using extensive testing that is tailored for the specific operation of the application.

Finally, once a suitable rescue point is identified, ASSURE produces a remediation patch that is dynamically applied to the software while the application is executing on the production system. The patch instantiates a rescue point inside the application to protect the application against the recurrence of the particular fault. The modified application will trigger a checkpoint whenever execution reaches the rescue point, and rolls back its state to that point should the fault recur. Once execution is rolled back to the rescue point, error virtualization is used to leverage the existing error-handling capabilities of the application to handle the fault gracefully. Instead of filtering particular inputs that can cause faults, the patch hardens the application against faults that may occur at a specific program location. The resulting recovery mechanism is input-agnostic, and thus immune to risks related to fault/input polymorphism.

6.2.1 Architectural Components

We now describe the individual components of our system: a *pre-deployment profiler*, a *monitor and fault detector*, a *rescue point analyzer* and a *remediation generator*.

Application Profiler

Prior to deployment, ASSURE profiles the application to determine the set of potential rescue points. Profiling is carried out during numerous "bad" runs to build an application behavioral model. To generate these bad runs, ASSURE uses regression tests, if available, as well as input-fuzzing techniques [80] that automate the

creation of "bad" input and thus stress the error-handling capabilities of the application. The intuition is that there exists a set of programmer-tested application points that are routinely used to handle gracefully "expected" errors. In turn, these application points can be harnessed to recover from failures that were not anticipated by the programmer, and thus maintain system availability. Using this model, ASSURE identifies program locations that can be used as candidate rescue points. Although the profiling process can be time-consuming, it need only be performed once per application. Moreover, the resulting set of candidate rescue points can be used by multiple deployments of that application.

Monitoring and Fault Detection

ASSURE continuously monitors the execution of the application in the production system. The role of the monitor is twofold. First, it needs to be able to detect application failures and misbehavior: provide fault description, indicate fault location, show stack state, *etc.* Second, it is responsible for providing sufficient information about the fault for the analysis component to be able to reproduce it. Namely, recreate the sequence of events that led to the fault's manifestation.

To detect failures and misbehavior, ASSURE employs a variety of fault-detection mechanisms; it only requires that a detection mechanism will notify the monitor of the occurrence of a fault. Besides standard operating system error handling (*e.g.,* program termination due to illegal memory dereferences), ASSURE can use additional mechanisms for detecting memory errors. There are a number of available fault detection components that can detect memory errors, for instance ProPolice [52], ASLR [89], and TaintCheck [84], and some that can detect violations to underlying security policies [12, 69, 91].

Due to performance considerations, we prefer to use lightweight fault detectors on

the actual production software [52, 89]. Alternatively the monitor can operate in a "honeypot" mode, in which a shadow instance of the application is instrumented with the fault detectors; traffic to the production system is mirrored to the shadow. While the shadow system can employ more heavy-duty fault detectors with wider coverage, we expect it to be unable to keep up with the production server, due to the additional instrumentation. Therefore, traffic to the shadow system is mirrored selectively, for example, using an anomaly detector that selects "interesting" flows/packets [17], or sampling input, or on a best-effort basis.

To provide sufficient information to reproduce a fault, ASSURE must record and identify fault-inducing input. This is the natural location in the system to keep track of causal information that is essential to link a detected fault with the input that had led to it. Consequently, the monitor maintains a log of incoming traffic and provides this log along with the fault description to the rescue point analyzer. If that input cannot be uniquely identified, a window of inputs is used instead.

Dealing with deterministic bugs, where there is a direct correspondence between inputs and failures, is straightforward using both the honeypot and lightweight fault-detector configurations. When the occurrence of a fault is related to some specific state, fault-detection components that have an unobstructed view of the application state (*i.e.,* built into application) stand a better chance of being able to re-create, and thus protect against, the fault. ASSURE combines input logs with periodic checkpointing to facilitate the reproduction of faults that manifest in the presence of particular states. Although not a strict reproduction of the sequence of events (*i.e.,* may not reproduce the order of signals or process scheduling), it can reproduce most externally influenced application state.

Periodic checkpointing at the production machine has the following benefits. First, it provides a snapshot of application state which, in conjunction with the request

logs, can help re-create the state of the application when the fault occurred. This is important for the analysis step where the system tries to reproduce the fault, and subsequently provide a remedy. Second, it places a bound on the number of requests that need to be logged; the system only keeps a copy of the requests since the last checkpoint. Third, it minimizes the time it takes to reproduce the fault. The system simply needs to replay the logs since the last checkpoint. Fast fault reproduction is of vital importance to ASSURE as it reduces the vulnerability window.

Rescue Point Analyzer

The rescue point analyzer uses the fault description provided by the monitor and the information gathered through the profiling analysis to identify likely rescue points. The analyzer then uses a shadow instance of the application on a dedicated system as a test oracle against which to evaluate candidate fault-recovery strategies; a strategy consists of a rescue point location and an error virtualization policy. Specifically, ASSURE evaluates the candidates using the following algorithm. First, it constructs a control-flow graph (CFG) of the sequence of function calls that led to the fault. From these functions, it picks and instantiates a candidate rescue point, taking a checkpoint whenever the execution reaches that point. Then, it replays the input that was responsible for the failure, while monitoring the application. When the fault occurs, ASSURE rolls back the execution state to the rescue point and initiates error virtualization. If this succeeds, ASSURE proceeds to perform consistency tests and, if satisfied with the results, creates a dynamic patch which is intended to be applied to the production system. If any of these steps fails, the algorithm repeats with a different strategy. Rescue points are discussed further in depth in Section 6.3.

Remediation Generator

The remediation generator applies the dynamic patch to the application that is executing in the production system, enabling automatic recovery from the particular fault. It consists of a checkpoint-rollback mechanism, a specialized fault detector and an error virtualization element. First, the application is instrumented to take a checkpoint whenever execution reaches the selected rescue point, ensuring that it will be possible to rollback to that point should a fault occur later on. Second, the fault detector is injected in the appropriate location at the production software to be able to detect future instances of the specific fault [112]. The fault detector is specialized for the fault in question to minimize its performance impact on the production system. Moreover, in some cases ASSURE can activate the fault detector only upon traversing the rescue point, and deactivate it once the vulnerable code has completed its execution. Upon detection of a fault on the production (patched) software, ASSURE rolls back the application state to a predetermined program location (the rescue point), where the program is forced to return a suitable error, imitating the behavior observed during the erroneous runs.

6.3 Error Virtualization Using Rescue Points

We now describe the basic concept of error virtualization using rescue points, which, combined with the checkpoint-rollback mechanism, forms the core of the remediation component. We describe the process in a top-down approach, starting with error virtualization using rescue points, followed by a detailed description of individual components.

Error virtualization using rescue points is the primary mechanism employed by ASSURE to recover from software faults. It is the end-product of ASSURE's opera-

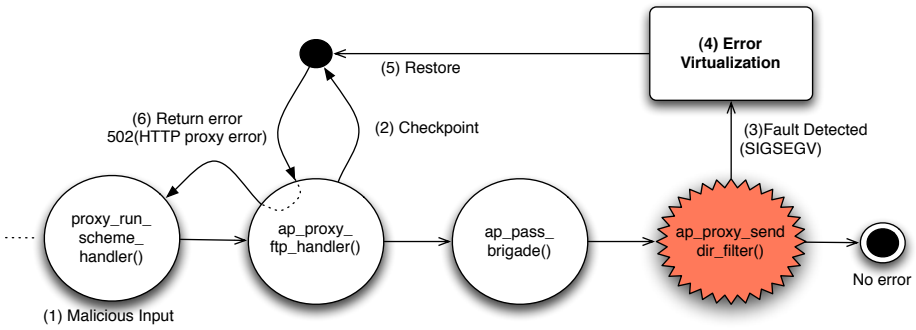


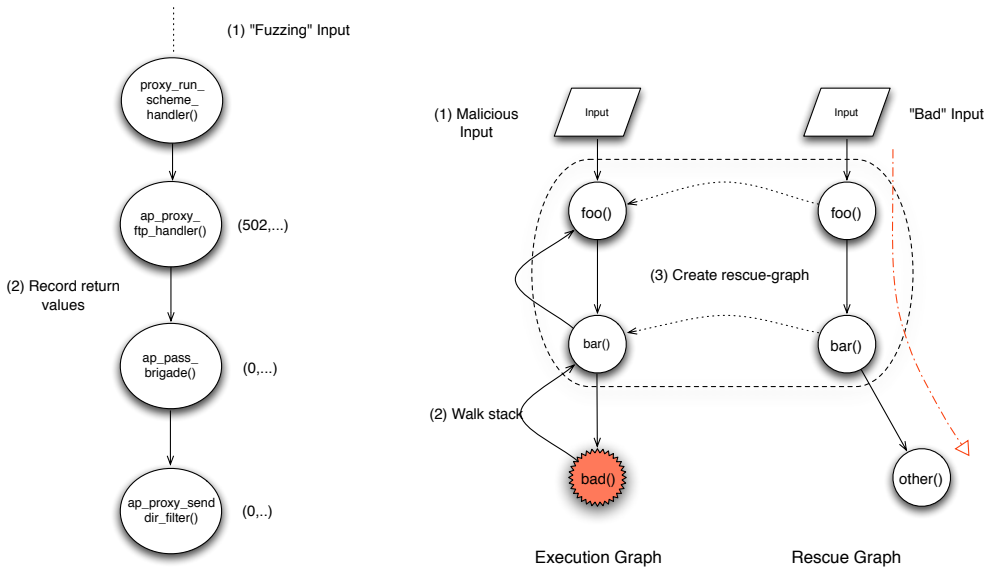
Figure 6.2: Error virtualization (EV) using rescue points: (1) Fault-triggering input is supplied to the application. (2) The application checkpoints its state at a rescue point. (3) A fault detector monitors protected functions for faults. When a fault occurs in a protected function, (4) the EV decides on a recovery strategy and (5) rolls back application state to the rescue point. (6) Once application state is restored, the EV forces an error return using the determined value.

tion, and its purpose is to protect against the manifestation of an already analyzed fault at the production system. Its operation can be summarized by the following steps: (a) checkpoint application state at the rescue point; (b) monitor application for faults in the protected code; (c) when a fault occurs, roll back application state to the rescue point; (d) after the rollback to the rescue point, force return with an error using an observed value from the application profiler.

Figure 6.2 illustrates ASSURE's self healing on a real bug in the Apache web server. A detailed description of the bug is provided in Appendix A.0.2. The scenario involved the execution of three functions: `ap_proxy_ftp_handler()`, `ap_pass_brigade()` and `ap_proxy_send_dir_filter()`. Due to bad input, the bug manifests in `ap_proxy_send_dir_filt` and results in a memory fault (SIGSEGV). ASSURE intercepted the first occurrence of the fault, identified `ap_proxy_ftp_handler()` as a suitable rescue point, and instrumented the Apache server as follows. Whenever the patched server enters `ap_proxy_ftp_handler()`, ASSURE takes a checkpoint of the server state, and allows the server to continue execution. If the same (or similar) fault occurs in

`ap_proxy_send_dir_filter()`, the fault detection component in the patched server detects the error and notifies the error virtualization (EV) component. The EV component analyzes the fault information and rolls back the server state back to the rescue point in `ap_proxy_ftp_handler()`. Furthermore, instead of allowing execution to proceed down the same path that caused the fault to manifest, ASSURE uses error virtualization to force `ap_proxy_ftp_handler()` to return with an error value identified during the application profiling stage, namely 502 (HTTP "Proxy Error").

In the remainder of this section we describe how ASSURE discovers, selects, creates, tests, and deploys rescue points.



(a) A section of the rescue-trace

(b) Generating the rescue-graph

Figure 6.3: Figure (a) shows the process of creating rescue-traces. (1) The application is bombarded with "bad" input. (2) Its behavior is monitored and error return values are recorded. Figure (b) shows how rescue-graphs are extracted to identify candidate rescue points for a particular fault: (1) When an fault-detection monitor diagnoses a fault (2) it examines that call-graph that lead to the fault. (3) The call-graph is compared with the rescue-trace to find common nodes that, in turn, form the rescue-graph: the set of candidate rescue points along with the suggested error return value.

6.3.1 Rescue Point Discovery

Determining a suitable recovery point is of pivotal importance to error virtualization. It determines, to a large extent, the likelihood that the application will survive a fault. During the application profiling stage, ASSURE uses dynamic analysis with fuzzing to discover suitable rescue points. The goal is to learn how an application responds to "bad" input, under controlled conditions, and use this knowledge in the future to map previously unseen faults to a set of observed fault behaviors. For example, we would like to see how a program normally handles errors when stress-tested by quality assurance tests. Learning from "bad" behavior has been used in machine learning and subsequently intrusion detection systems, but to the best of our knowledge has not been used to recover from software faults.

Specifically, ASSURE instruments applications by inserting monitoring code at every function's entry and exit points using the run-time injection capabilities of Dyninst [31], a runtime binary injection tool. The instrumentation records return values, function parameters, and return types (the latter two are available only when the binary is not stripped) while the application is bombarded with faults (through fault injection) and fuzzed inputs (*e.g.,* malformed protocol requests). From these traces, ASSURE extracts function call-graphs along with the history of return values used at each point in the graph. We call these graphs the *rescue-traces*.

Figure 6.3a illustrates part of the rescue-trace generated using the Apache bug example introduced earlier. The figure shows a summarized execution trace that includes `ap_proxy_ftp_handler()`, `ap_pass_brigade()` and `ap_proxy_send_dir_filter()`, as well as the observed error values that are associated with each function, which are 502, 0 and 0 respectively.

6.3.2 Rescue Point Selection

When a fault is detected in a specific code region for the first time, the call-stack is examined to derive the sequence of functions that led to the fault. At that point, ASSURE compares the call-stack with the rescue-trace from the discovery phase to derive common nodes. The common nodes form the set of candidate rescue points, or the *rescue-graph*.

Once candidate rescue points are identified, ASSURE attempts to determine their return type. If debugging information is available, function return types can be extracted directly from the binary. In the case of stripped binaries, as is the case with most commercial off-the-shelf (COTS) applications, ASSURE estimates the actual return type of the function through a set of heuristics that work on the observed return values found in profiling traces and through binary analysis [59]. Candidate rescue points are filtered according to heuristics that consider both the return type (if available) and the observed return values. Currently, candidate rescue points are functions with non-pointer return types, or functions that return pointers but the observed return value is `NULL`. Functions that return pointers require a deeper inspection of the data structures to ascertain the values of their return types beyond the simple case of returning a `NULL`. Preliminary empirical examination shows that the examined *C* programs favor the use of integer return types as failure indicators.

Next, ASSURE examines the return value distribution that was found at each candidate rescue point. The objective is to find a value that the error virtualization component can use to trigger error-handling code. The obvious strategy is to use the most frequently used return value, given that the profiling runs consist of execution traces that propagate errors. This is especially true in the absence of source code, where ASSURE cannot verify how the actual code handles errors.

Figure 6.3b illustrates how candidate rescue points are identified for a particular fault. When a fault is detected in `ap_proxy_send_dir_filter()`, the call-stack is examined to determine the execution path that had lead to the observed failure. This path is compared against the rescue-trace to determine overlapping functions that form the rescue-graph. Using the same example from the figure, functions `ap_proxy_ftp_handler()` and `ap_pass_brigade()` form the rescue graph for the particular fault instance.

The rescue points in the rescue-trace can be sorted using different selection strategies. We chose perhaps the simplest: sort the rescue points by shortest distance to the faulty code that represents an active function on the call graph. The idea is that suitable rescue points that reside closer to the fault will minimize the performance overhead that rescue points incur (due to checkpointing and monitoring for the specific fault), since they might avoid critical application paths that get invoked on each request. ASSURE follows this ordering to instantiate and test rescue points, seeking one that enables recovery from the given fault. If the closest rescue point fails during testing, ASSURE chooses the nearest active ancestor and repeats.

6.3.3 Rescue Point Creation

Having determined a set of candidate rescue points, ASSURE can now activate and test rescue points. To activate a rescue point, ASSURE needs to insert code into the application running in the testing environment. This is done using the same mechanism for deploying the rescue point on the production server as described in Section 6.3.4. Using this mechanism, ASSURE activates a rescue point by inserting at the function designated as the rescue point a call to `int rescue_capture(id,fault)`. The parameter `id` uniquely identifies a rescue point; `fault` is a structure that con-

tains all additional information pertaining to the rescue point, including the error virtualization code to be used to force an early return.

The `rescue_capture()` function is responsible for capturing the state of the application as it executes through the rescue point by performing a checkpoint. Checkpoints are kept entirely in memory using standard copy-on-write semantics and are indexed by their corresponding identifiers. `rescue_capture()` returns the rescue point identifier upon a successful checkpoint, or zero when it returns following a rollback of the application state. A typical calling sequence is given in the following code snippet. Similar to `fork()` semantics, the return value of the function `rescue_capture()` directs the execution context.

```
int rescue_point( int id, fault_t fault ) {
    int ret = rescue_capture(id, fault);
    if (ret < 0)
        handle_error(id);  /* rescue point error */
    else if (ret == 0)
        return get_rescue_return_value(fault);
    /* all ok */
    ...
}
```

The handling of rescue points becomes more complicated in the presence of multi-process or multi-threaded applications. The key issue with rescue points is that the checkpoints are always initiated by the application, since they must occur at designated safe locations. With multiple (cooperative) processes, the state must be obtained not only in a globally consistent manner, but also with all the participating processes stationed at some suitable point. By "cooperative" we refer to processes that share resources, such as shared memory, or some portion of their state, and

therefore must agree on that state at any point in time. On the other hand, if the processes are independent, then they can be handled independently one by one as in the single-process case.

ASSURE ensures global consistency of checkpoints by using coordinated checkpoints of cooperative process groups. When a process in a group reaches a rescue point, it must wait until the remaining processes are ready in order to take the checkpoint synchronously. While processes clearly become ready when they reach a rescue point, the implied synchronization can adversely impact performance. This is because there are no hard guarantees that the other processes will reach a safe point soon. To mitigate this, ASSURE can checkpoint processes when they enter or exit the kernel due to system calls or signal handling. Thus, when a process initiates a checkpoint, ASSURE marks other processes in the group processes, telling them to request a checkpoint as soon as possible. If other processes are blocked on I/O, signals are used to force them out of blocking calls.

6.3.4 Rescue Point Testing

Once a candidate rescue point has been elected, ASSURE proceeds to verify the efficacy of the proposed fix, by testing the rescue-enabled version of the application. To accomplish this, ASSURE restarts the application from the most recent checkpoint image available in a separate testing environment, and then feeds it with the specific input that caused the fault to manifest. If the offending input cannot be identified easily, ASSURE replays longer input sequences recorded since that checkpoint. Eventually the fault occurs and triggers a rollback to the selected rescue point. If the application crashes, fails to maintain service availability, or is not semantically equivalent, a new fix is created using the next available candidate rescue point and

the testing and analysis phase is repeated.

If the fix does not introduce any faults that cause the application to crash, the application is examined for semantic bugs using a set of user-supplied tests. The purpose of these tests is to increase confidence about the semantic correctness of the generated fix. For example, an online vendor could run tests that verify that client orders can be submitted and processed by the system.

For our initial approach, we are primarily concerned with failures where there is a one-to-one correspondence between inputs and failures, and not with those that are caused by a combination of inputs. Note, however, that many of the latter types of failures are in fact addressed by our system, because the last input (and the code leading to a failure) will be recognized as "problematic" and handled as we have discussed.

6.3.5 Rescue Point (Patch) Deployment

Swift patch deployment is of foremost importance in "reactive" systems. First, it reduces system downtime and subsequently improves system availability. Second, it allows for the deployment of critical fixes that could curtail the spread of large-scale epidemics such as in the case of worms. Previous work has relied on a traditional software development cycle of making changes to source code (albeit automatically through source-to-source transformations), compiling, linking, testing and then instantiating the new version of the application. For this work, we examined binary instrumentation as a mechanism for deploying patches. Specifically, we chose to use runtime injection and Dyninst [31] in particular due to its low runtime overhead and its ability to attach and detach from already running processes. The Dyninst API provides a rich interface that facilitates code steering through injection and call-site

replacement within functions. Note that in addition to being used for the final rescue point patch deployment on the production server, the same runtime injection mechanism is also used to insert rescue points into the shadow deployment of the application during rescue-point testing, and to inject the fault monitoring mechanism into the production server.

Chapter 7

Evaluation of Error Virtualization

In this Chapter, we experimentally evaluate the efficacy of error virtualization as a recovery strategy for general software failures and vulnerabilities.

We begin our evaluation of error virtualization with an examination of real-world software failures for a number of server applications. We examine 10 real-world bugs using error virtualization with rescue-points. We examine each of the bugs in detail to provide insight into the applicability of our recovery techniques.

Finally, we validate our examination of real-world bugs with comprehensive fault-injection study on six server applications using an end-to-end version our system, including comprehensive fault injection, full system deployment, operating on a stripped binary, and verification of output correctness in addition to survivability tests. For this study, we use the ASSURE system implementation to compare the original error virtualization algorithm, error virtualization using rescue points and the use of random error values.

Application	Version	Bug	Reference	Depth	Value	Benchmark
Apache	1.3.31	Buffer overflow	CVE-2004-0940	1	NULL	httperf-0.8
Apache	2.0.59	NULL dereference	ASF Bug 40733	3	502	httperf-0.8
Apache	2.0.54	Off-by-one	CVE-2006-3747	2	-1	httperf-0.8
ISC Bind	8.2.2	Input Validation	CAN-2002-1220	2	-1	dnsperf 1.0.0.1
MySQL	5.0.20	Buffer overflow	CAN-2002-1373	2	1	sql-bench 2.15
Squid	2.4	Input Validation	CVE-2005-3258	1	void	WebStone 2.5b3
OpenLDAP	2.3.39	Design Error	CVE-2008-0658	2	80	DirectoryMark 1.3
PostgreSQL	8.0	Input Validation	CVE-2005-0246	1	0	BenchmarkSQL 2.3.2

Table 7.1: **List of real vulnerabilities and bugs used in the evaluation. ASSURE recovered from all bugs; for each bug we show the rescue-distance and the virtualized error value used.**

7.1 Experimental Evaluation

We have implemented an ASSURE prototype system for Linux. It consists of user-space utilities and loadable kernel modules for off-the-shelf Linux 2.6 kernel that provide the virtual execution environment with checkpoint-restart and log-replay facilities, and Dyninst 5.2b3 for runtime code injection. Using this prototype, we evaluate the effectiveness of ASSURE on real bugs and standard workloads for a number of popular multi-process and multi-threaded server applications. For our experiments, with the exception of generating sufficient profiling information for specific application modules, a process that needs to occur one per application (or provided as part of a testing suite), and triggering the initial fault, the process for dealing with the bugs was fully automated, requiring no human intervention or application source code. All experiments were conducted on machines with dual Intel Xeon 3.06 GHz processors and 2.5 GB of RAM, connected through a 1 Gbps Ethernet connection. The servers and clients used for our experiments ran on separate machines.

We evaluate the effectiveness of ASSURE in handling bugs along three axes: *survivability*, *correctness* and *performance*. *Survivability* examines ASSURE's ability to maintain service availability in the presence of a bug-induced software failure. AS-

SURE detects failures and automatically initiates the recovery process. Post-recovery, we monitor the server for failures that might have been induced by our mechanism and verify that the server continues to service requests correctly. Since it is possible that the recovery mechanism introduced side-effects, we verify the *correctness* of server output following recovery: we not only examine the ability of the server to provide service, but also compare server output to predefined test-suites to support claims of semantic equivalence. Finally, we look at various *performance* implications of ASSURE in terms of both full system overhead at the production server and piecewise examination of system components.

7.1.1 Bug Summary

Table 7.1 lists the bugs and vulnerabilities that we used to evaluate ASSURE. We used eight bugs for six popular applications: `Apache`, `named (ISC Bind)`, `MySQL`, `Squid`, `OpenLDAP` and `PostgreSQL`. The bugs range from illegal memory dereferences to off-by-one errors and buffer overflows as indicated by column *Bug* in Table 7.1. We trigger bugs using existing or specially crafted exploit code based on information derived from online vulnerability databases. We did not have bugs available for closed-source Linux applications to evaluate since most popular Linux applications are open-source. While our examination consists of open-source applications, they were treated as commercial-off-the-shelf (COTS) software by stripping binaries of all symbols and removing access to source code.

7.1.2 Overall Functionality Results

Table 7.1 demonstrates the overall effectiveness of running ASSURE against a set of real-world bugs and vulnerabilities. For each bug, the table shows the affected

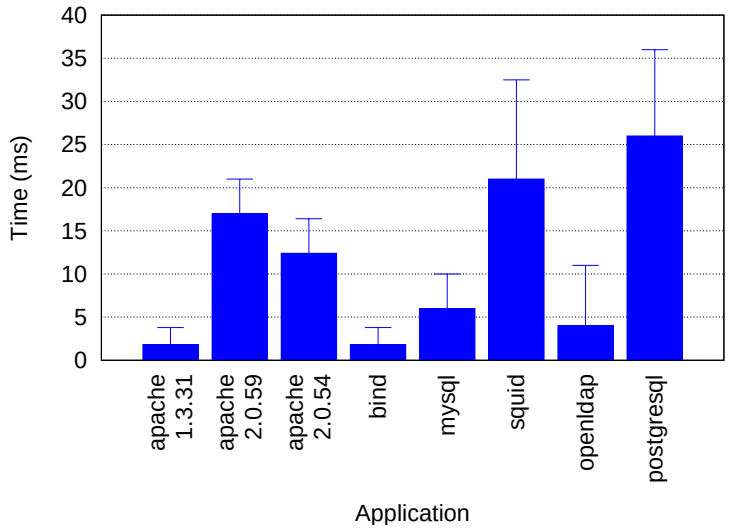


Figure 7.1: Rescue-point to fault

application, the type of bug and its reference, and a benchmark used to verify the correctness and measure the performance. For each examined bug, ASSURE was able to find a rescue point that allows the application to survive the induced failure.

In detail, bugs are triggered during the execution of a benchmark to simulate recovery when the application is under load. The application is monitored to examine its ability to successfully complete the benchmark. If the benchmark completes, we have a measure of survivability and performance. At that point, the application is tested for correctness either through an examination of the benchmark results (if they report correctness) or through additional tests that examine and compare the output to an expected set of results. For each bug, we report the rescue depth and rescue value: the distance between the fault and the rescue point and the error virtualization value used to propagate errors.

Columns *depth* and *value* indicate the rescue depth and rescue value, respectively, for each bug. The average observed rescue depth for the bugs is 2. As previously mentioned, we evaluate rescue points for survivability, correctness and performance.

In the case of `MySQL`, ASSURE found a rescue point at rescue depth 1 that allowed the application to pass the survivability and correctness tests but it was a rescue point at depth 2 that provided better performance characteristics. The reason was that the rescue point at depth 2 allowed the benchmark to complete without triggering excessive checkpoints. Our testing framework was able to determine this behavior automatically.

A short rescue depth is encouraging because it indicates that rescue points tend to cluster close to faults, minimizing the effect they might have on system performance. For multi-process (or multi-threaded) servers, this also means that the amount of progress by processes (or threads) other than the one that experienced the fault is limited, and therefore their rollback is less likely to cause collateral damage. For instance, a short rescue depth can reduce the chance that any client-visible communication will have occurred between checkpoint and rollback. Figure 7.1 presents the distance between a checkpoint and a rollback in milliseconds. Specifically, we measure the time between when a checkpoint is taken and when a subsequent rollback occurs. The error bars shown in the graph indicate the average time from checkpoint completion to execution continuation, and elapsed time from when a rollback begins and until activity of the old processes ceases. The values shown in the graph range from 1.8 ms for `Apache 1.3` to 26 ms in the case of `Postgres`. We argue that for most of the examined applications, the recorded checkpoint-to-fault times represents less than one request thus minimizing impact on progress made by other processes/threads.

The range of return values used by rescue points show a degree of correlation with previously observed results [112]. Values of 0 and -1 are often used to propagate errors but there are cases, as in Apache mod_ftp bug and openLDAP, where an observed value of 502 and 80 respectively is a more appropriate value to return.

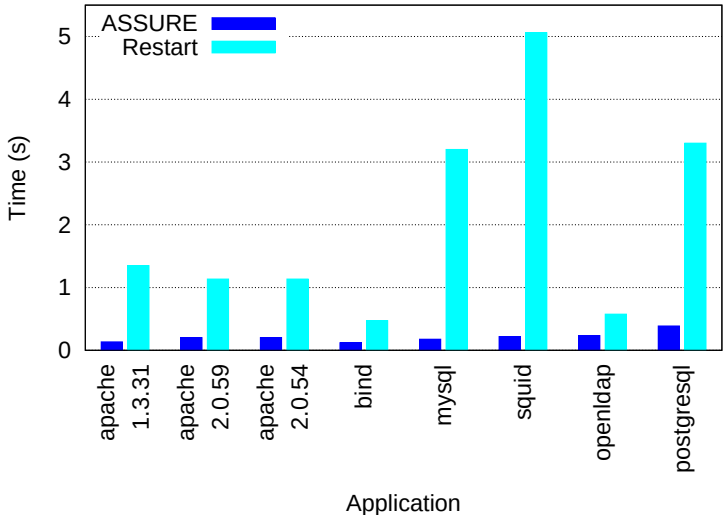


Figure 7.2: Recovery time

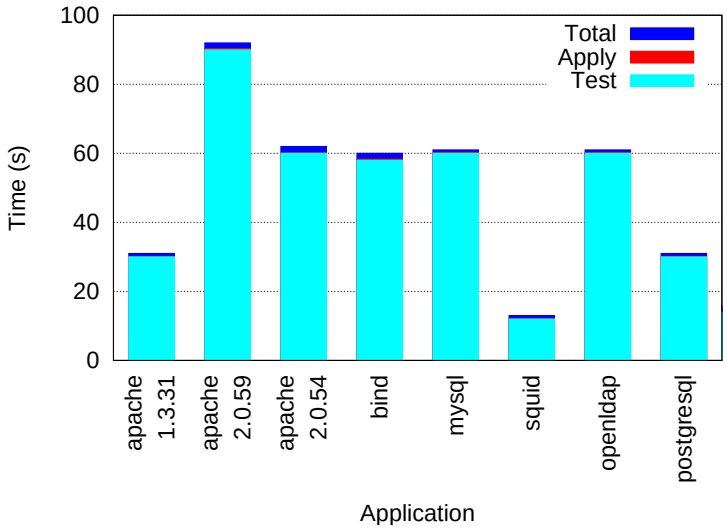


Figure 7.3: Patch generation time

7.1.3 Patch Generation Performance

To evaluate the responsiveness of ASSURE in generating a patch for a newly discovered failure, we measured the total time required to go from fault-to-patch. In other words, from when a fault is first detected on the production system to the dynamic application of the patch.

Figure 7.3 shows the average times, in seconds, to create a working, tested patch for each of the bugs. *Total* denotes the total time required to create, test and apply a patch. The total time is broken down into two parts: *apply* shows the time it takes to attach to a running process and insert the rescue-point using ASSURE's instrumentation, and *test* shows the time required to run the survivability, correctness and performance test, on average, for a successful rescue point. Note that the time required to generate the rescue graph for a particular fault is negligible, given that the stack trace was less than 15 functions deep in all cases.

As shown in Figure 7.3, the average total patch generation time varies between 15 and 92 seconds. For a given application and a correctness test-suite, the total time was roughly linear with the rescue depth as tests are repeated for each potential rescue point. These times are conservative in two ways. First, our prototype allows the correctness test-suite to run to completion before moving on to the next rescue point candidate. This can be optimized by rejecting an unsuitable rescue point at the time a test fails rather than analyzing results after completing the test. Second, our prototype serializes the process of rescue point selection and testing. With a typical rescue depth of at most 3, this can be optimized by simply testing rescue points in parallel. Other parallelization can also be done, such as parallel execution of different tests for a given rescue point.

The breakdown of the total time shows that the survivability and correctness testing is the dominant factor in the end-to-end latency of the patch generation process. For this evaluation we used test-suites that stress test the applications in order to reveal long-term side-effects such as memory leaks. In practice, organizations deploying ASSURE can choose to minimize testing so that it encompasses core functionality and thus reduce the time required to test each rescue-point. Alternatively, if they are concerned with correctness, more comprehensive tests can be used, providing a

trade-off between the time to generate a patch and testing coverage.

The time required by ASSURE to create and dynamically apply a rescue-point patch ranges from 70 ms for `Apache 1.3.1` to 120 ms for `mysql`. The brunt of that cost is loading and parsing the ASSURE instrumentation library into the runtime image of the server. The numbers represent great improvements over the traditional patch, compile, stop and restart cycle.

Although these results represent an unoptimized implementation, they show a patch turn-around-time that is orders of magnitude faster than manually created patches. According to Symantec, the average time between discovery of a critical memory error and a subsequent patch is 28 days [2].

7.1.4 Recovery Performance

For each bug, we evaluate the fault recovery performance of ASSURE. Specifically, we measure the time to recover application state to a rescue point once a fault has been detected. As in previous experiments, the fault was triggered while the application was busy completing the specified benchmarks to simulate recovery under load.

We compared ASSURE's recovery time with that of a whole application restart after a fault, in which we measured the elapsed time from launching the application until it becomes operational and ready to serve requests. Whole application restart does not necessarily allow recovery, but it can reset the server to enable it to serve future requests even if it does not allow a workload that was running at the time of the fault to complete. While it does not provide the same level of survivability as ASSURE, it provides a useful comparison of recovery time. Note that this comparison is on the conservative side, since most servers accumulate state in dedicated caches to significantly improves their performance; our measurements do not cap-

ture the negative effect of while program restart on performance due to effectively discarding that state and caches. To simulate realistic application restart times, we test application restart using real workloads. For PostgreSQL, we measure the time required to restart the application when it is pre-loaded with the Wisconsin dataset. For OpenLDAP, we use a snapshot of the directory server of Columbia University's Computer Science department.

Figure 7.2 shows the average time required to restore execution to a rescue point, or in other words, rollback execution. As shown, restart times range from 135 ms for `Apache 1.3`, to 388 ms for `Postgres`. Whole application restart times ranged from 470 ms for `bind` to 5 seconds for `Squid`. These results show that ASSURE's recovery time is order of magnitudes faster (4x-23x) than whole application restart. This holds true even for applications, such as Apache, that do not need to rebuild considerable amounts of application state before they become operational. Note that whole application restart can take longer after a fault than starting the application on a clean system due to checks the application may do as a result of a crash. In the case of PostgreSQL, a whole restart of the server was even unsuccessful in enabling the server to serve future requests after a bug. Because of the bug, the benchmark that was executing in the background failed to complete gracefully due to data corruption. The data corruption prevented the server from serving requests for corrupted portions of the database, so the benchmark could not even be re-run after restarting the server. In contrast, ASSURE allowed the application to successfully complete the benchmark that was executing when the fault occurred.

To evaluate client perceived availability, we examined the number of client observed errors due to fault recovery. Specifically, we inject bugs while executing the benchmark, and measured the number of dropped connections and unanswered requests as a portion of the total requests. We varied fault injection at 10, 20 and

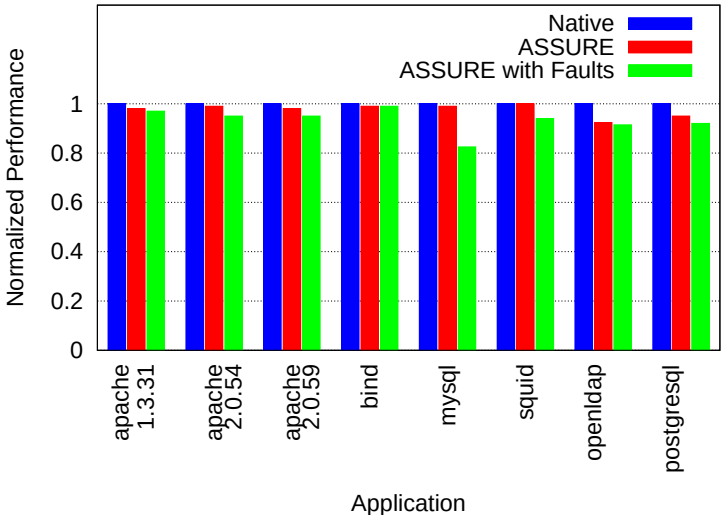


Figure 7.4: Normalized performance

30 second intervals. The values range from 1% to 10%, for a fault every 30 and 10 seconds respectively.

7.1.5 Patch Overhead

Given ASSURE's success in finding rescue points that enable the system recover execution from the injected faults, we wanted to examine the performance implications of our "fixes". Specifically, for each bug, we examined the effects of our patches on system performance. We compare the execution performance of an unmodified version of the application versus the ASSURE generated patch using the previously described benchmarks. We also measure the performance overhead of triggering a failure during the execution of the benchmark. The results, as normalized performance overhead, are shown in Figure 7.4. As shown in the figure, ASSURE has minimal impact on performance. The values range from 0% for Squid to 7.6% for OpenLDAP. These results are expected for two reasons. First, the virtualization and instrumentation overhead is small, similar to what is reported in previous work [73]. Second, for all

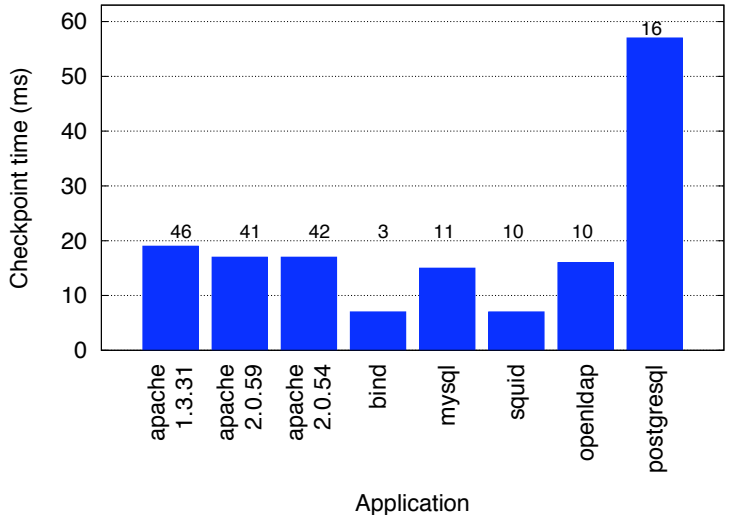


Figure 7.5: Checkpoint Times

examined bugs, the failures occur in code regions which are not in the main execution path of the application. This allows ASSURE to not burden the cost of rescue points (checkpoints) unless that vulnerable code path is taken by the application. For a more detail performance analysis of the cost of taking rescue points, we defer the reader to the next Section.

Also shown in Figure 7.4, is the performance overhead for triggering a failure during the execution of the benchmarks. Again, the overhead is low, ranging from 1% for `bind` to 8.5% for `OpenLDAP`. This result is expected given the short recovery times reported above. It translates to a few requests not being serviced during the benchmark.

7.1.6 ASSURE Component Overhead

To obtain a better understanding of the underlying costs of rescue points, we measure the cost of individual components that comprise the cost of rescue points for each of the examined bugs. Specifically, we measure the time required to take a checkpoint

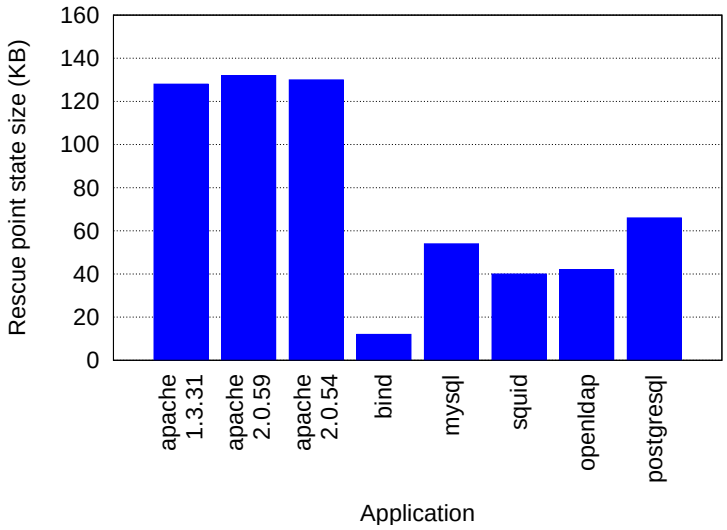


Figure 7.6: Checkpoint Size

and record its size. As the parameters depend on application size and number of processes/threads it encompasses, we report the average number of processes (including threads) for each application executing through the rescue point.

Figure 7.5 shows the average time required to complete a rescue point (checkpoint) for each of the examined bugs. Specifically, it shows the *application downtime* during which the application is unresponsive. The results show rescue point downtime ranges from 7 ms for Squid to roughly 50 ms, in the worst case, for `PostgreSQL`. Most values range between 10 ms and 20 ms which represent a modest downtime when one considers that ASSURE checkpoints multiple processes. Also shown in the graph is the average number of processese/threads.

Figure 7.6 indicates the average memory requirements per rescue point. The size of a rescue point is directly correlated with the number of processes and memory footprint of the application. As expected, the results represent a range in value that commensurate with the size of the application. In detail, the checkpoint size values range from 12KB for `bind` to 130KB for `MySQL`. These values represent the state

changes between checkpoints. Our copy-on-write mechanism allows to avoid saving full application state. The full application state size ranges from 20MB for `bind` to 116MB for `Postgres`. Since ASSURE only requires the latest checkpoint to initiate recovery, the rescue point space requirements are manageable.

7.2 Evaluation on Injected Faults

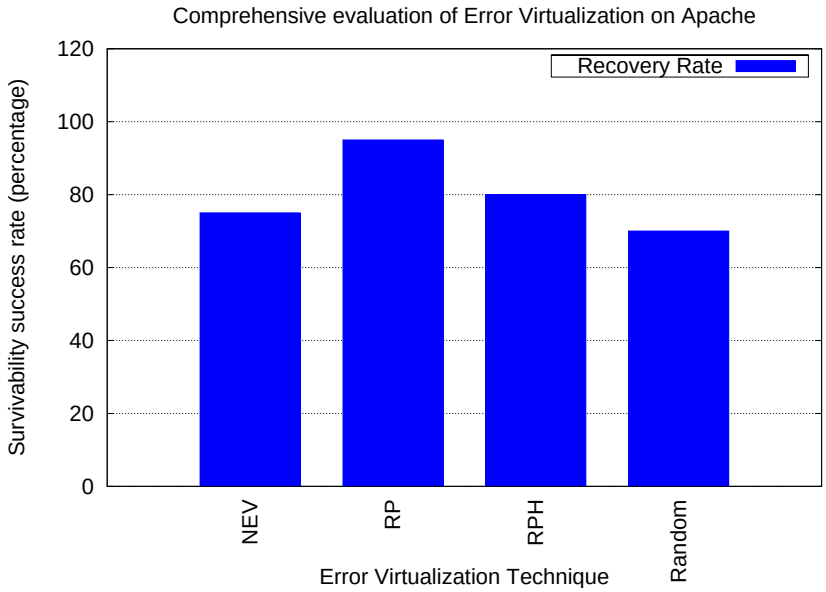


Figure 7.7: Error Virtualization Examination on Apache. The figure shows results for the different techniques examined: Naive Error Virtualization (NEV), Rescue Points (RP), Heuristic-based Rescue Poinst (RPH) and Random error values.

Following the successful evaluation of ASSURE using real bugs, we expand our testing with a comprehensive fault-injection study on the Apache web server and less exhaustive testing framework for five server applications: `thttpd, sshd, wu-ftpd, bind, mysql`.

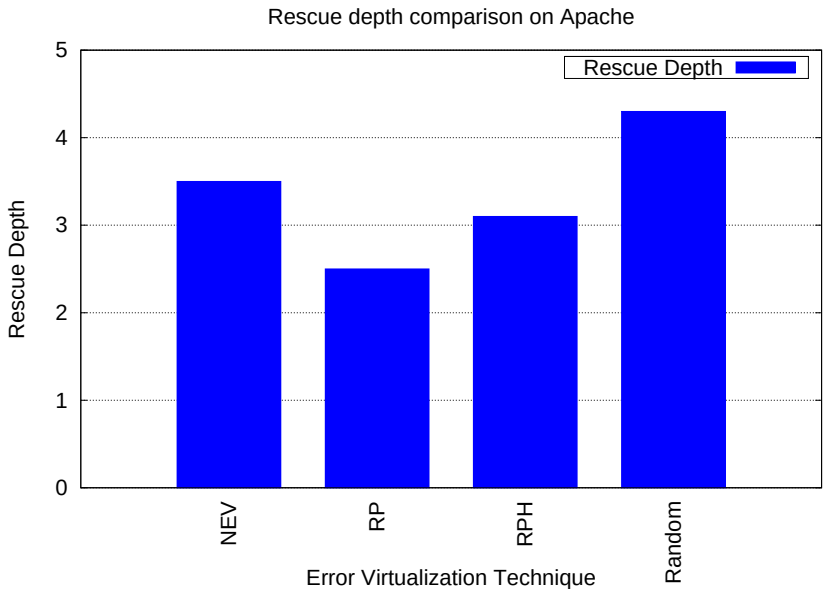


Figure 7.8: Rescue depth using different error virtualization heuristics on Apache. The figure shows results for the different techniques examined: Naive Error Virtualization (NEV), Rescue Points (RP), Heuristic-based Rescue Poinst (RPH) and Random error values.

7.2.1 Comprehensive Evaluation on Apache

We begin our evaluation with a comprehensive fault-injection study on the Apache web server. The goal of these tests is to *exhaustively* examine Error Virtualization's ability to recovery program execution. We use an end-to-end version of our system, including comprehensive fault injection, full system deployment, operating on a stripped binary, and verification of output correctness in addition to survivability tests.

For a comprehensive (and resource-consuming) end-to-end analysis of ASSURE, we examine the effects of thorough fault injection using a strict definition of application correctness.

We profile the application using our instrumentation infrastructure. We run tests that exercise standard application functionality and extract execution control-flow

graphs (CFG). For each function in the extracted CFG, we artificially inject faults using Dyninst. We insert the equivalent of `char s = *(char*) 0;` in the beginning of each basic block of the target function, causing a memory error (segmentation violation) when that code is reached.

Using the above fault-injection methodology, we examine four recovery strategies:

- **Error virtualization using rescue points (ASSURE):** For each injected fault, ASSURE attempts to find a rescue-point (from the rescue-graph of each function) using Algorithm 2 that will pass the *survivability* and *correctness* tests.

- **Heuristic-based ASSURE:** This version of ASSURE, examines how the rescue-point algorithm fares by using the heuristics described for NEV instead of discovered return values.

- **Naive Error Virtualization (NEV):** For the case of naive error virtualization, we force an early return on the observed function using Algorithm 1. Namely, we examine the function's return type and return a value based on the type. We return -1 for `int`, 0 for `unsigned` and for pointers we return `NULL`.

- **Random error values:** Finally, we evaluate the success rate of the ASSURE algorithm when using randomly selected return values.

For injected fault, we evaluate each strategy against two tests: *survivability* and *correctness*.

- **Survivability:** The first experiment evaluates the effectiveness of ASSURE in maintaining service availability in the presence of a bug-induced software failure. Specifically, we trigger a fault by sending an appropriately crafted request to the server and let ASSURE detect the failure and initiate recovery through

error virtualization. We then monitor the server for any failures that might be induced by our mechanism and verify that the server can continue handling normal service requests.

- **Correctness:** Since it is possible that our recovery mechanism could have introduced side-effects, we also measure the correctness of server output following recovery. In detail, we not only examine the ability of the server to service requests, following recovery, but also compare the output of the replies to expected values.

The purpose of these tests is to motivate the use of error return mapping for general application faults.

Apache Recovery Rate

Figure 7.7 shows the error recovery results for the different error virtualization strategies. As shown, the rescue point algorithm is able to produce a correct patch for 95% (198/207) of the functions. NEV and rescue points using heuristics perform similarly with 75% and 80% respectively. This is primarily due to the fact that the NEV algorithm does not continue to iterate until it finds a correct rescue point. The return values used by both are identical. What is of particular interest is the success rate of returning random values. Returning random values was able to produce a correct patch in about 70% of the cases. This is due to return value elasticity of a few key functions in Apache where returning any value other than `0` indicates a failure. In the next section, we show returning other values affects the rescue depth of the fix.

Apache Rescue Depth

Next, we examined how the different error virtualization strategies effect the rescue depth of a fix. The results are shown in Figure 7.8. Rescue points are able to produce patches with lower rescue depths than the other techniques for two reasons. First, using observed values has a higher chance of triggering existing error handling code. Second, rescue points are able to use functions that return pointers if the observed value is `NULL`. This translates to rescue depths of 2.5 functions for rescue points at one end to 4.2 functions for random return values at the other.

A short distance between a fault and a rescue point implies that the performance cost of checkpointing and monitoring for faults while executing at that rescue point can be compartmentalized. Thus, a vulnerable function that is not routinely used is less likely to affect performance, since rescue-enabled functions do not adversely impact performance unless they are in the execution path. For example, if a rescue point resides near an `accept()` call, it is in an advantageous position to handle faults, but will trigger a checkpoint on every request. A rescue point that is closer to the vulnerable function will affect performance only when that function is invoked.

7.2.2 Error Virtualization Survivability Results

Next, we extend our application evaluation to five additional server applications beyond Apache. Specifically, we examine `thttpd, sshd, wu-ftpd, bind, mysql`. For this set of experiments we limit our examination to *survivability* testing.

The results of these tests are shown in Figure 7.9. The figure shows a comparison between NEV and rescue points. As can been from the results, using the rescue points algorithm our system was able to produce a fix that enabled the server to survive the injected failure in every case. On the other hand, NEV was able to produce a fix for

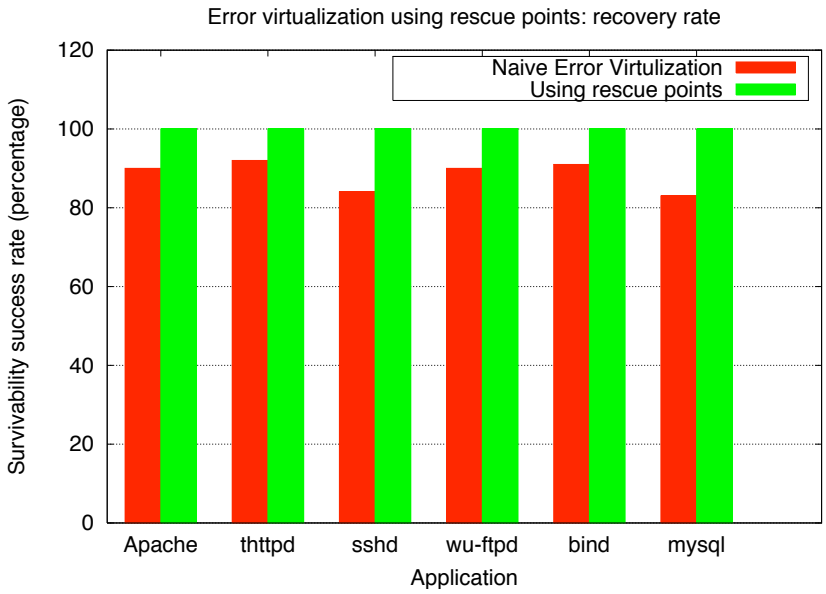


Figure 7.9: **Error Virtualization Recovery Rate**

just over 80% of the cases. This result shows that although heuristics tend to work fairly well in practice, observed values are a more appropriate mechanism. In the next section, we provide some intuition into why heuristics can only work in some of the cases.

7.2.3 Return Value Distribution

To provide insight into why our observation-based mechanism outperforms heuristics-based approaches, we examine the return value distributions of the applications used in our experiment. Figure 7.10 illustrates the observed return values along with their frequency distribution for "bad" input; pointer return values are filtered for clarity. The graphs show a concentration around low-value integers, which helps explain why heuristics tend to give relatively good results. However, the graphs also demonstrate that the examined applications have a fairly wide coverage of values. This range of return values explains why heuristics will always produce sub-par results for recovering

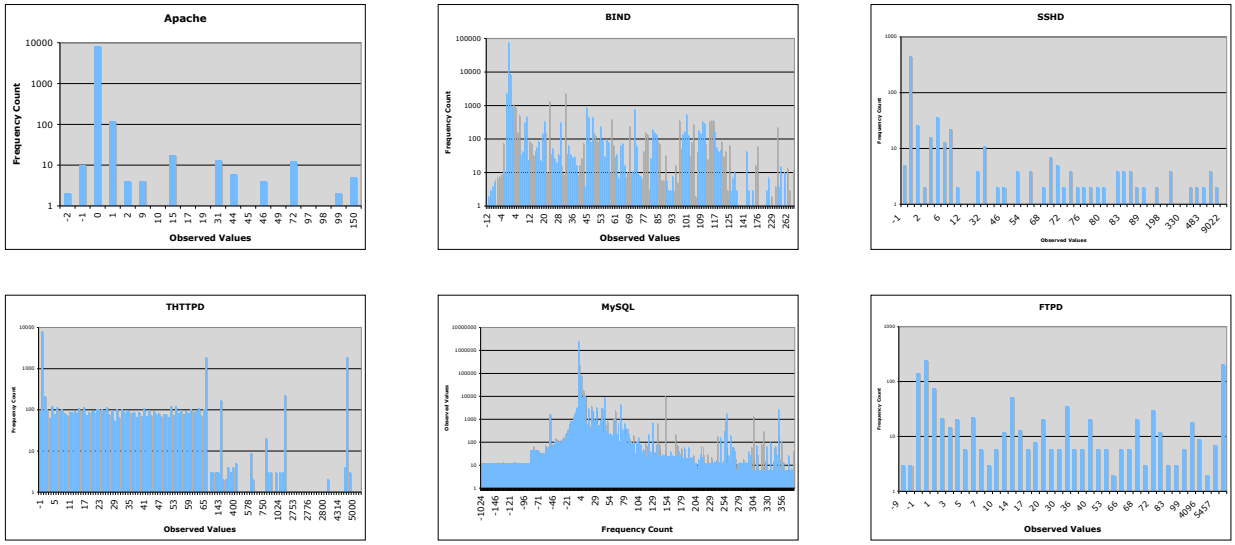


Figure 7.10: **Return value distribution: The purpose of this busy graph is to illustrate the distribution of values returned by functions during erroneous input. This range of return values explains why heuristics tend to produce sub-par results for recovering program execution.**

program execution.

These results, along with supporting evidence from [97] and [133], validate our "error virtualization" hypothesis and approach.

Naturally, it is possible that the applications we examined might exhibit long-term side effects, *e.g.,* through data structure corruption. Our experimental evaluation through benchmark suites, which issue many thousand requests to the same application, gives us some confidence that their internal state does not "decay" quickly. To address longer-term deterioration, we can use either micro-rebooting (software rejuvenation) [36] or automated data-structure repair [50]. We intend to examine the combination of our approach with either of these techniques in future work.

Chapter 8

Deployment Scenarios

In the previous Chapters, we introduced a number of self-healing mechanism. A common characteristic was the requirement that the protected application have instrumentation capable of notifying the system of any detected failures. Most instrumentation mechanisms have a non-negligible performance overhead. In the next two Chapters, we discuss deployment strategies for software self-healing systems than can ameliorate the cost of monitoring an application.

8.1 Shadow Honeypots

In this Section, we introduce the concept of a shadow honeypot. Shadow honeypots extend traditional honeypots by having the ability to share state with the application they are trying to protect and can therefore be used to detect targeted attacks (*e.g.* attacks that exploit specific application state) against the application. Furthermore, shadow honeypots can reduce the cost of monitoring an application by deciding at run-time whether to service a request using "expensive" instrumentation or to run it natively.

8.2 Introduction

Due to the increasing level of malicious activity seen on today's Internet, organizations are beginning to deploy mechanisms for detecting and responding to new attacks or suspicious activity, called Intrusion Prevention Systems (IPS). Since current IPS's use rule-based intrusion detection systems (IDS) such as Snort [99] to detect attacks, they are limited to protecting, for the most part, against already known attacks. As a result, new detection mechanisms are being developed for use in more powerful reactive-defense systems. The two primary such mechanisms are honeypots [92, 48, 139, 117, 74, 22] and anomaly detection systems (ADS) [128, 132, 127, 25, 72]. In contrast with IDS's, honeypots and ADS's offer the possibility of detecting (and thus responding to) previously unknown attacks, also referred to as *zero-day attacks*.

Honeypots and anomaly detection systems offer different tradeoffs between accuracy and scope of attacks that can be detected, as shown in Figure 8.1. Honeypots can be heavily instrumented to accurately detect attacks, but depend on an attacker attempting to exploit a vulnerability against them. This makes them good for detecting scanning worms [4, 7, 48], but ineffective against manual directed attacks or topological and hit-list worms [120, 119]. Furthermore, honeypots can typically only be used for server-type applications. Anomaly detection systems can theoretically detect both types of attacks, but are usually much less accurate. Most such systems offer a tradeoff between false positive (FP) and false negative (FN) rates. For example, it is often possible to tune the system to detect more *potential* attacks, at an increased risk of *misclassifying* legitimate traffic (low FN, high FP); alternatively, it is possible to make an anomaly detection system more insensitive to attacks, at the risk of missing some real attacks (high FN, low FP). Because an ADS-based IPS can adversely affect legitimate traffic (*e.g.,* drop a legitimate request), system designers

often tune the system for low false positive rates, potentially misclassifying attacks as legitimate traffic.

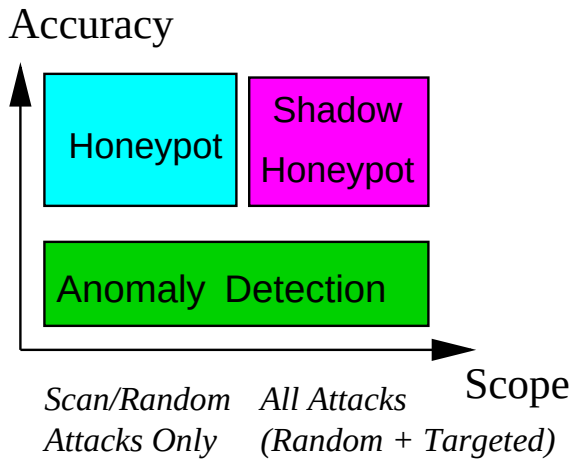


Figure 8.1: A simple classification of honeypots and anomaly detection systems, based on attack detection accuracy and scope of detected attacks. Targeted attacks may use lists of known (potentially) vulnerable servers, while scan-based attacks will target any system that is believed to run a vulnerable service. AD systems can detect both types of attacks, but with lower accuracy than a specially instrumented system (honeypot). However, honeypots are blind to targeted attacks, and may not see a scanning attack until after it has succeeded against the real server.

We propose a novel hybrid approach that combines the best features of honeypots and anomaly detection, named *Shadow Honeypots*. At a high level, we use a variety of anomaly detectors to monitor all traffic to a protected network. Traffic that is considered anomalous is processed by a shadow honeypot. The shadow version is an instance of the protected application (*e.g.,* a web server or client) that shares all internal state with a "normal" instance of the application, but is instrumented to detect potential attacks. Attacks against the shadow honeypot are caught and any incurred state changes are discarded. Legitimate traffic that was misclassified by the anomaly detector will be validated by the shadow honeypot and will be *transparently* handled correctly by the system (*i.e.,* an HTTP request that was mistakenly flagged

as suspicious will be served correctly). Our approach offers several advantages over stand-alone ADS's or honeypots:

- First, it allows system designers to tune the anomaly detection system for low false negative rates, minimizing the risk of misclassifying a real attack as legitimate traffic, since any false positives will be weeded out by the shadow honeypot.

- Second, and in contrast to typical honeypots, our approach can defend against attacks that are *tailored* against a specific site with a particular internal state. Honeypots may be blind to such attacks, since they are not typically mirror images of the protected application.

- Third, shadow honeypots can also be instantiated in a form that is particularly well-suited for protecting against *client-side* attacks, such as those directed against web browsers and P2P file sharing clients.

- Finally, our system architecture facilitates easy integration of additional detection mechanisms.

We apply the concept of shadow honeypots to a proof-of-concept prototype implementation tailored against memory-violation attacks. Specifically, we developed a tool that allows for automatic transformation of existing code into its "shadow version". The resulting code allows for traffic handling to happen through the regular or shadow version of the code, contingent on input derived from an array of anomaly detection sensors. When an attack is detected by the shadow version of the code, state changes effected by the malicious request are rolled back. Legitimate traffic handled by the shadow is processed successfully, albeit at higher latency.

In addition to the server-side scenario, we also investigate a client-targeting attack-detection scenario, unique to shadow honeypots, where we apply the detection heuristics to content retrieved by protected clients and feed any positives to shadow honeypots for further analysis. Unlike traditional honeypots, which are idle whilst waiting for active attackers to probe them, this scenario enables the detection of passive attacks, where the attacker lures a victim user to download malicious data. We use the recent `libpng` vulnerability of Mozilla [10] (which is similar to the buffer overflow vulnerability in the Internet Explorer's JPEG-handling logic) to demonstrate the ability of our system to protect client-side applications.

Our shadow honeypot prototype consists of several components. At the front-end of our system, we use a high-performance intrusion-prevention system based on the Intel IXP network processor and a set of modified *snort* sensors running on normal PCs. The network processor is used as a smart load-balancer, distributing the workload to the sensors. The sensors are responsible for testing the traffic against a variety of anomaly detection heuristics, and coordinating with the IXP to tag traffic that needs to be inspected by shadow honeypots. This design leads to the scalability needed in high-end environments such as web server farms, as only a fraction of the servers need to incur the penalty of providing shadow honeypot functionality.

In our implementation, we have used a variety of anomaly detection techniques, including Abstract Payload Execution (APE) [127], and the Earlybird algorithm [114]. The feasibility of our approach is demonstrated by examining both false-positive and true attack scenarios. We show that our system has the capacity to process all false-positives generated by APE and EarlyBird and successfully detect attacks. We also show that when the anomaly detection techniques are tuned to increase detection accuracy, the resulting additional false positives are still within the processing budget of our system. More specifically, our benchmarks show that although instrumentation

is expensive (20-50% overhead), the shadow version of the Apache Web server can process around 1300 requests per second, while the shadow version of the Mozilla Firefox client can process between 1 and 4 requests per second. At the same time, the front-end and anomaly detection algorithms can process a fully-loaded Gbit/s link, producing 0.3-0.5 false positives per minute when tuned for high sensitivity, which is well within the processing budget of our shadow honeypot implementation.

8.3 Architecture

The Shadow Honeypot architecture is a systems approach to handling network-based attacks, combining filtering, anomaly detection systems and honeypots in a way that exploits the best features of these mechanisms, while shielding their limitations. We focus on transactional applications, *i.e.,* those that handle a series of discrete requests. Our architecture is *not* limited to server applications, but can be used for client-side applications such as web browsers, P2P clients, *etc.* As illustrated in Figure 8.2, the architecture is composed of three main components: a filtering engine, an array of anomaly detection sensors and the shadow honeypot, which validates the predictions of the anomaly detectors. The processing logic of the system is shown graphically in Figure 8.3.

The filtering component blocks known attacks. Such filtering is done based either on payload content [131, 3] or on the source of the attack, if it can be identified with reasonable confidence (*e.g.,* confirmed traffic bi-directionality). Effectively, the filtering component short-circuits the detection heuristics or shadow testing results by immediately dropping specific types of requests before any further processing is done.

Traffic passing the first stage is processed by one or more anomaly detectors. There

are several types of anomaly detectors that may be used in our system, including payload analysis [132, 114, 66, 127] and network behavior [63, 138]. Although we do not impose any particular requirements on the AD component of our system, it is preferable to tune such detectors towards high sensitivity (at the cost of increased false positives). The anomaly detectors, in turn, signal to the protected application whether a request is potentially dangerous.

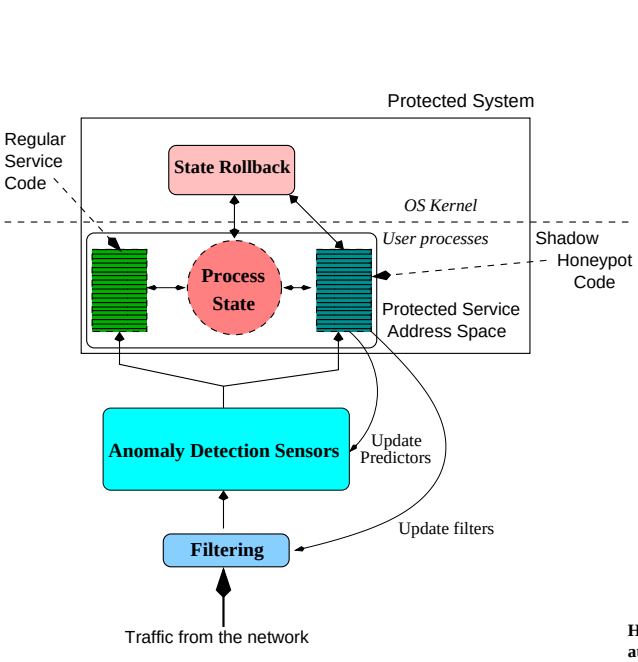


Figure 8.2: **Shadow Honeypot architecture.**

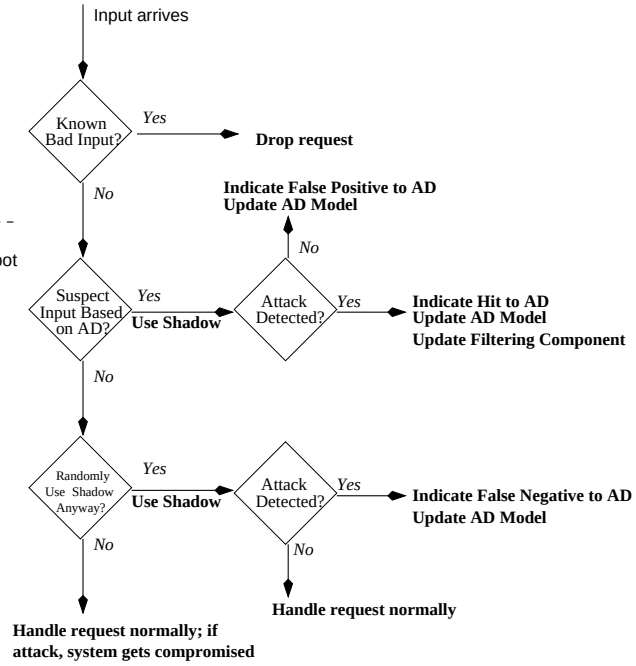


Figure 8.3: **System workflow.**

Depending on this prediction by the anomaly detectors, the system invokes either the regular instance of the application or its *shadow*. The shadow is an instrumented instance of the application that can detect specific types of failures and rollback any state changes to a known (or presumed) good state, *e.g.*, before the malicious request was processed. Because the shadow is (or should be) invoked relatively infrequently, we can employ computationally expensive instrumentation to detect attacks. The shadow and the regular application fully share state, to avoid attacks that exploit

differences between the two; we assume that an attacker can only interact with the application through the filtering and AD stages, *i.e.,* there are no side-channels. The level of instrumentation used in the shadow depends on the amount of latency we are willing to impose on suspicious traffic (whether truly malicious or misclassified legitimate traffic). In our implementation, described in Section 8.4, we focus on memory-violation attacks, but any attack that can be determined algorithmically can be detected and recovered from, at the cost of increased complexity and potentially higher latency.

If the shadow detects an actual attack, we notify the filtering component to block further attacks. If no attack is detected, we update the prediction models used by the anomaly detectors. Thus, our system could in fact self-train and fine-tune itself using verifiably bad traffic and known mis-predictions, although this aspect of the approach is outside the scope of the present paper.

As we mentioned above, shadow honeypots can be integrated with servers as well as clients. In this paper, we consider tight coupling with both server and client applications, where the shadow resides in the same address space as the protected application.

- **Tightly coupled with server** This is the most practical scenario, in which we protect a server by diverting suspicious requests to its shadow. The application and the honeypot are tightly coupled, mirroring functionality and state. We have implemented this configuration with the Apache web server, described in Section 8.4.

- **Tightly coupled with client** Unlike traditional honeypots, which remain idle while waiting for active attacks, this scenario targets passive attacks, where the attacker lures a victim user to download data containing an attack, as with the

recent buffer overflow vulnerability in Internet Explorer's JPEG handling. In this scenario, the context of an attack is an important consideration in replaying the attack in the shadow. It may range from data contained in a single packet to an entire flow, or even set of flows. Alternatively, it may be defined at the application layer. For our testing scenario, specifically on HTTP, the request/response pair is a convenient context.

Tight coupling assumes that the application can be modified. The advantage of this configuration is that attacks that exploit differences in the state of the shadow *vs.* the application itself become impossible. However, it is also possible to deploy shadow honeypots in a *loosely coupled* configuration, where the shadow resides on a different system and does not share state with the protected application. The advantage of this configuration is that management of the shadows can be "outsourced" to a third entity.

Note that the filtering and anomaly detection components can also be tightly coupled with the protected application, or may be centralized at a natural aggregation point in the network topology (*e.g.,* at the firewall).

Finally, it is worth considering how our system would behave against different types of attacks. For most attacks we have seen thus far, once the AD component has identified an anomaly and the shadow has validated it, the filtering component will block all future instances of it from getting to the application. However, we cannot depend on the filtering component to prevent polymorphic or metamorphic [125] attacks. For low-volume events, the cost of invoking the shadow for each attack may be acceptable. For high-volume events, such as a Slammer-like outbreak, the system will detect a large number of correct AD predictions (verified by the shadow) in a short period of time; should a configurable threshold be exceeded, the system can

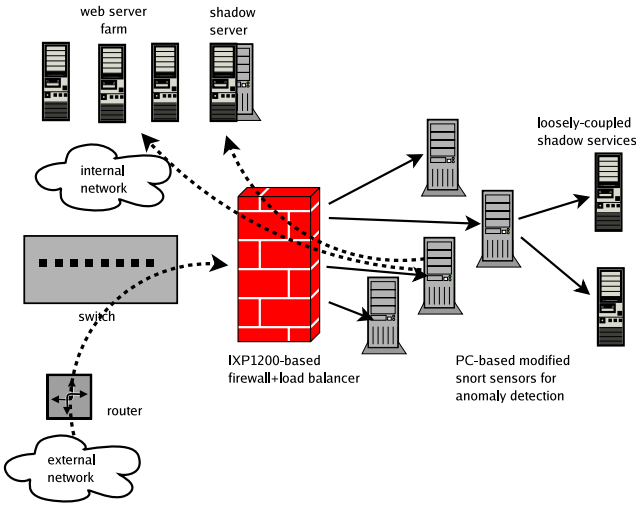


Figure 8.4: **High-level diagram of prototype shadow honeypot implementation.**

enable filtering at the second stage, based on the unverified verdict of the anomaly detectors. Although this will cause some legitimate requests to be dropped, this could be acceptable for the duration of the incident. Once the number of (perceived) attacks seen by the ADS drop beyond a threshold, the system can revert to n

8.4 Implementation

8.4.1 Filtering and anomaly detection

During the composition of our system, we were faced with numerous design issues with respect to performance and extensibility. When considering the deployment of the shadow honeypot architecture in a high-performance environment, such as a Web server farm, where speeds of at least 1 Gbit/s are common and we cannot afford to missclassify traffic, the choice for off-the-shelf components becomes very limited. To the best of our knowledge, current solutions, both standalone PCs and network-processor-based network intrusion detection systems (NIDSes), are well under the 1

Gbit/s mark [42, 102].

Faced with these limitations, we considered a distributed design, similar in principle to [126, 71]: we use a network processor (NP) as a scalable, custom load balancer, and implement all detection heuristics on an array of (modified) snort sensors running on standard PCs that are connected to the network processor board. We chose not to implement any of the detection heuristics on the NP for two reasons. First, currently available NPs are designed primarily for simple forwarding and lack the processing capacity required for speeds in excess of 1 Gbit/s. Second, they remain harder to program and debug than standard general purpose processors. For our implementation, we used the IXP1200 network processor. A high-level view of our implementation is shown in Figure 8.4.

A primary function of the anomaly detection sensor is the ability to divert potentially malicious requests to the shadow honeypot. For web servers in particular, a reasonable definition of the attack context is the HTTP request. For this purpose, the sensor must construct a request, run the detection heuristics, and forward the request depending on the outcome. This processing must be performed at the HTTP level thus an HTTP proxy-like function is needed. We implemented the anomaly detection sensors for the tightly-coupled shadow server case by augmenting an HTTP proxy with ability to apply the APE detection heuristic on incoming requests and route them according to its outcome.

For the shadow client scenario, we use an alternative solution based on passive monitoring. Employing the proxy approach in this situation would be prohibitively expensive, in terms of latency, since we only require detection capabilities. For this scenario, we reconstruct the TCP streams of HTTP connections and decode the HTTP protocol to extract suspicious objects.

As part of our proof-of-concept implementation we have used two anomaly de-

tection heuristics: payload sifting and abstract payload execution. Payload sifting as developed in [114] derives fingerprints of rapidly spreading worms by identifying popular substrings in network traffic. It is a prime example of an anomaly detection based system that is able to detect novel attacks at the expense of false positives. However, if used in isolation (*e.g.,* outside our shadow honeypot environment) by the time it has reliably detected a worm epidemic, it is very likely that many systems would have already been compromised. This may reduce its usage potential in the tightly-coupled server protection scenario without external help. Nevertheless, if fingerprints generated by a distributed payload sifting system are disseminated to interested parties that run shadow honeypots locally, matching traffic against such fingerprints can be of use as a detection heuristic in the shadow honeypot system. Of further interest is the ability to use this technique in the loosely-coupled shadow server scenario, although we do not further consider this scenario here.

The second heuristic we have implemented is buffer overflow detection via abstract payload execution (APE), as proposed in [127]. The heuristic detects buffer overflow attacks by searching for sufficiently long sequences of valid instructions in network traffic. Long sequences of valid instructions can appear in non-malicious data, and this is where the shadow honeypot fits in. Such detection mechanisms are particularly attractive because they are applied to individual attacks and will trigger detection upon encountering the first instance of an attack, unlike many anomaly detection mechanisms that must witness multiple attacks before flagging them as anomalous.

8.4.2 Shadow Honeypot Creation

To create shadow honeypots, we use a code-transformation tool that takes as input the original application source code and "weaves" into it the shadow honeypot code. In

this paper, we focus on memory-violation errors and show source-code transformations that detect buffer overflows, although other types of failures can be caught (*e.g.,* input that causes illegal memory dereferences) with the appropriate instrumentation, but at the cost of higher complexity and larger performance bottleneck. For the code transformations we use TXL [78], a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Naur (BNF). In our prototype, called DYBOC, we use TXL for *C*-to-*C* transformations with the GCC *C* front-end.

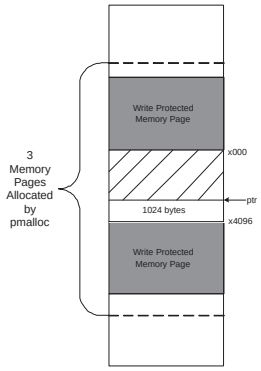


Figure 8.5: Example of *pmalloc()*-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate "waste" memory. This is needed to ensure that *mprotect()* is applied on complete memory pages.

The instrumentation itself is conceptually straightforward: we move all static buffers to the heap, by dynamically allocating the buffer upon entering the function in which it was previously declared; we de-allocate these buffers upon exiting the function, whether implicitly (by reaching the end of the function body) or explicitly (through a *return* statement). We take care to properly handle the *sizeof* construct, a fairly straightforward task with TXL. Pointer aliasing is not a problem with our

system, since we instrument the allocated memory regions; any illegal accesses to these will be caught.

Original code

```
int func()
{
    char buf[100];
...
    other_func(buf, sizeof(buf);
...
    return 0;
}
```

Modified code

```
int func()
{
    char *buf;
    char _buf[100];
    if (shadow_enable())
      buf = pmalloc(100);
    else
      buf = _buf;
...
    other_func(buf, sizeof(_buf));
...
    if (shadow_enable()) {
      pfree(buf);
    }
    return 0;
}
```

Figure 8.6: Transforming a function to its shadow-supporting version. The *shadow_enable()* macro simply checks the status of a shared-memory variable (controlled by the anomaly detection system) on whether the shadow honeypot should be executing instead of the regular code.

For memory allocation, we use our own version of *malloc()*, called *pmalloc()*, that allocates two additional zero-filled, write-protected pages that bracket the requested buffer, as shown in Figure 8.5. The guard pages are *mmap()*'ed from */dev/zero* as read-only. As *mmap()* operates at memory page granularity, every memory request is rounded up to the nearest page. The pointer that is returned by *pmalloc()* can be adjusted to immediately catch any buffer overflow or underflow depending on where attention is focused. This functionality is similar to that offered by the *ElectricFence* memory-debugging library, the difference being that *pmalloc()* catches both buffer overflow and underflow attacks. Because we *mmap()* pages from */dev/zero*, we do not waste physical memory for the guards (just page-table entries). Memory is wasted, however, for each allocated buffer, since we allocate to the next closest page. While

this can lead to considerable memory waste, we note that this is only incurred when executing in shadow mode, and in practice has proven easily manageable.

Figure 8.6 shows an example of such a translation. Buffers that are already allocated via *malloc()* are simply switched to *pmalloc()*. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the *C* grammar to free the variables before the function returns. After making changes to the standard ANSI *C* grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial. For single-threaded, non-reentrant code, it is possible to only use *pmalloc()* once for each previously-static buffer. Generally, however, this allocation needs to be done each time the function is invoked.

Any overflow (or underflow) on a buffer allocated via *pmalloc()* will cause the process to receive a Segmentation Violation (SEGV) signal, which is caught by a signal handler we have added to the source code in *main()*. The signal handler simply notifies the operating system to abort all state changes made by the process while processing this request. To do this, we added a new system call to the operating system, *transaction()*. This is conditionally (as directed by the *shadow_enable()* macro) invoked at three locations in the code:

- Inside the main processing loop, prior to the beginning of handling of a new request, to indicate to the operating system that a new transaction has begun. The operating system makes a backup of all memory page permissions, and marks all heap memory pages as read-only. As the process executes and modifies these pages, the operating system maintains a copy of the original page and allocates a new page (which is given the permissions the original page had

from the backup) for the process to use, in exactly the same way copy-on-write works in modern operating system. Both copies of the page are maintained until *transaction()* is called again, as we describe below. This call to *transaction()* must be placed manually by the programmer or system designer.

- Inside the main processing loop, immediately after the end of handling a request, to indicate to the operating system that a transaction has successfully completed. The operating system then discards all original copies of memory pages that have been modified during processing this request. This call to *transaction()* must also be placed manually.

- Inside the signal handler that is installed automatically by our tool, to indicate to the operating system that an exception (attack) has been detected. The operating system then discards all modified memory pages by restoring the original pages.

Although we have not implemented this, a similar mechanism can be built around the filesystem by using a private copy of the buffer cache for the process executing in shadow mode. The only difficulty arises when the process must itself communicate with another process while servicing a request; unless the second process is also included in the transaction definition (which may be impossible, if it is a remote process on another system), overall system state may change without the ability to roll it back. For example, this may happen when a web server communicates with a remote back-end database. Our system does not currently address this, *i.e.,* we assume that any such state changes are benign or irrelevant (*e.g.,* a DNS query). Specifically for the case of a back-end database, these inherently support the concept of a transaction rollback, so it is possible to undo any changes.

The signal handler may also notify external logic to indicate that an attack associated with a particular input from a specific source has been detected. The external logic may then instantiate a filter, either based on the network source of the request or the contents of the payload [131].

8.5 Experimental Evaluation

We have tested our shadow honeypot implementation against a number of exploits, including a recent Mozilla PNG bug and several Apache-specific exploits. In this section, we report on performance benchmarks that illustrate the efficacy of our implementation.

First, we measure the cost of instantiating and operating shadow instances of specific services using the Apache web server and the Mozilla Firefox web browser. Second, we evaluate the filtering and anomaly detection components, and determine the throughput of the IXP1200-based load balancer as well as the cost of running the detection heuristics. Third, we look at the false positive rates and the trade-offs associated with detection performance. Based on these results, we determine how to tune the anomaly detection heuristics in order to increase detection performance while not exceeding the budget alloted by the shadow services.

8.5.1 Performance of shadow services

Apache To determine the workload capacity of the shadow honeypot environment, we used DYBOC on the Apache web server, version 2.0.49. Apache was chosen due to its popularity and source-code availability. Basic Apache functionality was tested, omitting additional modules. The tests were conducted on a PC with a 2GHz Intel P4 processor and 1GB of RAM, running Debian Linux (2.6.5-1 kernel).

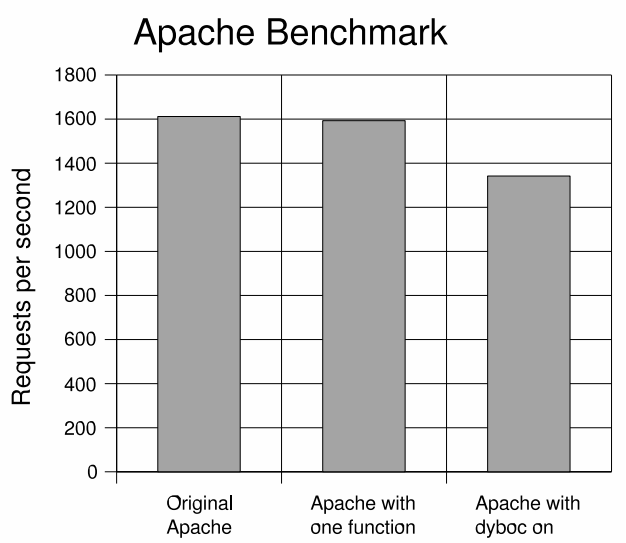


Figure 8.7: **Apache benchmark results.**

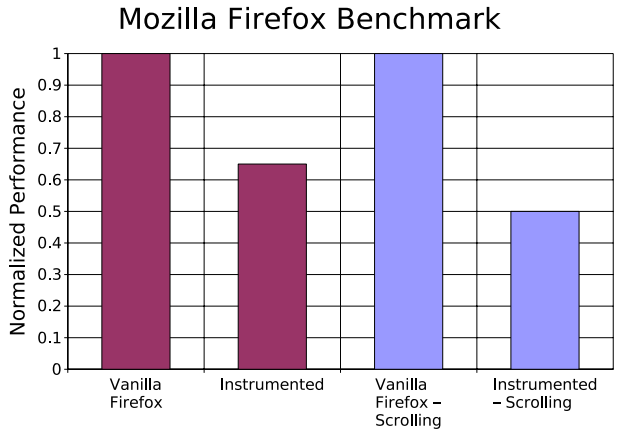


Figure 8.8: Normalized Mozilla Firefox benchmark results using modified version of i-Bench.

We used ApacheBench [6], a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance.

Figure 8.7 illustrates the requests per second that Apache can handle. There is a 20.1% overhead for the patched version of Apache over the original, which is

expected since the majority of the patched buffers belong to utility functions that are not heavily used. This result is an indication of the worst-case analysis, since all the protection flags were enabled; although the performance penalty is high, it is not outright prohibitive for some applications. For the instrumentation of a single buffer and a vulnerable function that is invoked once per HTTP transaction, the overhead is 1.18%.

Of further interest is the increase in memory requirements for the patched version. A naive implementation of *pmalloc()* would require two additional memory pages for each transformed buffer. Full transformation of Apache translates into 297 buffers that are allocated with *pmalloc()*, adding an overhead of 2.3MB if all of these buffers are invoked simultaneously during program execution. When protecting *malloc()*'ed buffers, the amount of required memory can skyrocket.

To avoid this overhead, we use an *mmap()* based allocator. The two guard pages are *mmap*'ed write-protected from */dev/zero,* without requiring additional physical memory to be allocated. Instead, the overhead of our mechanism is 2 page-table entries (PTEs) per allocated buffer, plus one file descriptor (for */dev/zero*) per program. As most modern processors use an MMU cache for frequently used PTEs, and since the guard pages are only accessed when a fault occurs, we expect their impact on performance to be small.

Mozilla Firefox For the evaluation of the client case, we used the Mozilla Firefox browser. For the initial validation tests, we back-ported the recently reported *libpng* vulnerability [10] that enables arbitrary code execution if Firefox (or any application using *libpng*) attempts to display a specially crafted PNG image. Interestingly, this example mirrors a recent vulnerability of Internet Explorer, and JPEG image handling [9], which again enabled arbitrary code execution when displaying specially crafted

images.

In the tightly-coupled scenario, the protected version of the application shares the address space with the unmodified version. This is achieved by transforming the original source code with our DYBOC tool. Suspicious requests are tagged by the ADS so that they are processed by the protected version of the code as discussed in Section 8.4.2.

For the loosely-coupled case, when the AD component marks a request for processing on the shadow honeypot, we launch the instrumented version of Firefox to replay the request. The browser is configured to use a null X server as provided by `Xvfb`. All requests are handled by a transparent proxy that redirects these requests to an internal Web server. The Web server then responds with the objects served by the original server, as captured in the original session. The workload that the shadow honeypot can process in the case of Firefox is determined by how many responses per second a browser can process and how many different browser versions can be checked.

Our measurements show that a single instance of Firefox can handle about one request per second with restarting after processing each response. Doing this only after detecting a successful attack improves the result to about four requests per second. By restarting, we avoid the accumulation of various pop-ups and other side-effects. Unlike the server scenario, instrumenting the browser does not seem to have any significant impact on performance. If that was the case, we could have used the rollback mechanism discussed previously to reduce the cost of launching new instances of the browser.

We further evaluate the performance implications of fully instrumenting a web browser. These observations apply to both loosely-coupled and tightly-coupled shadow honeypots. Web browsing performance was measured using a Mozilla Firefox 1.0

browser to run a benchmark based on the i-Bench benchmark suite [1]. i-Bench is a comprehensive, cross-platform benchmark that tests the performance and capability of Web clients. Specifically, we use a variant of the benchmark that allows for scrolling of a web page and uses cookies to store the load times for each page. Scrolling is performed in order to render the whole page, providing a pessimistic emulation of a typical attack. The benchmark consists of a sequence of 10 web pages containing a mix of text and graphics; the benchmark was ran using both the scrolling option and the standard page load mechanisms. For the standard page load configuration, the performance degradation for instrumentation was 35%. For the scrolling configuration, where in addition to the page load time, the time taken to scroll through the page is recorded, the overhead was 50%. The results follow our intuition as more calls to *malloc* are required to fully render the page. Figure 8.8 illustrates the normalized performance results. It should be noted that depending on the browser implementation (whether the entire page is rendered on page load) mechanisms such at the automatic scrolling need to be implemented in order to protected against targeted attacks. Attackers may hide malicious code in unrendered parts of a page or in javascript code activated by user-guided pointer movement.

How many different browser versions would have to be checked by the system? Figure 8.9 presents some statistics concerning different browser versions of Mozilla. The browser statistics were collected over a 5-week period from the CIS Department web server at the University of Pennsylvania. As evidenced by the figure, one can expect to check up to 6 versions of a particular client. We expect that this distribution will be more stabilized around final release versions and expect to minimize the number of different versions that need to be checked based on their popularity.

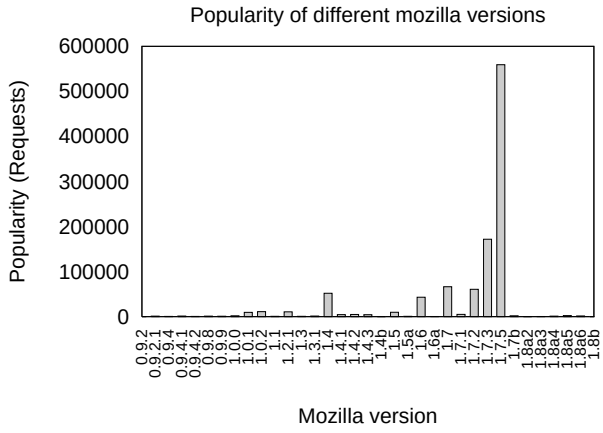


Figure 8.9: Popularity of different Mozilla versions, as measured in the logs of CIS Department Web server at the University of Pennsylvania.

8.5.2 Filtering and anomaly detection

IXP1200-based firewall/load-balancer. We first determine the performance of the IXP1200-based firewall/load-balancer. The IXP1200 evaluation board we use has two Gigabit Ethernet interfaces and eight Fast Ethernet interfaces. The Gigabit Ethernet interfaces are used to connect to the internal and external network and the Fast Ethernet interfaces to communicate with the sensors. A set of client workstations is used to generate traffic through the firewall. The firewall forwards traffic to the sensors for processing and the sensors determine if the traffic should be dropped, redirected to the shadow honeypot, or forwarded to the internal network.

Previous studies [116] have reported forwarding rates of at least 1600 Mbit/s for the IXP1200, when used as a simple forwarder/router, which is sufficient to saturate a Gigabit Ethernet interface. Our measurements show that despite the added cost of load balancing, filtering and coordinating with the sensors, the firewall can still handle the Gigabit Ethernet interface at line rate.

To gain insight into the actual overhead of our implementation we carry out a

Detection method	Throughput/sensor
Content matching	225 Mbit/s
APE	190 Mbit/s
Payload Sifting	268 Mbit/s

Table 8.1: **PC Sensor throughput for different detection mechanisms.**

second experiment, using Intel's cycle-accurate IXP1200 simulator. We assume a clock frequency of 232 MHz for the IXP1200, and an IX bus configured to be 64-bit wide with a clock frequency of 104 MHz. In the simulated environment, we obtain detailed utilization measurements for the *microengines* of the IXP1200. The results are shown in Table 8.10. The results show that even at line rate and worst-case traffic the implementation is quite efficient, as the microengines operate at 50.9%-71.5% of their processing capacity. These results provide further insight into the scalability of our design.

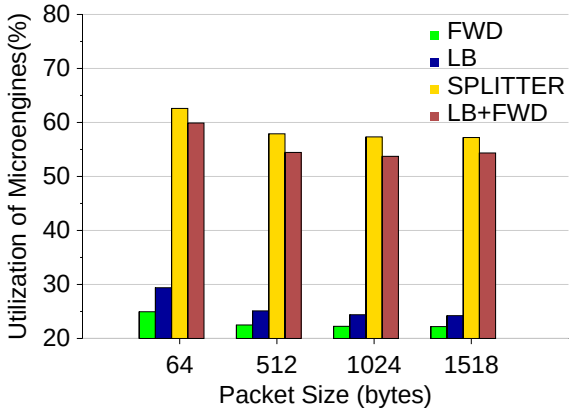


Figure 8.10: Utilization(%) of the IXP1200 Microengines, for forwarding-only (FWD), load-balancing-only (LB), both (LB+FWD), and full implementation (FULL), in stress-tests with 800 Mbit/s worst-case 64-byte-packet traffic.

PC-based sensor performance. We also measure the throughput of the PC-based sensors that cooperate with the IXP1200 for analyzing traffic and performing anomaly

detection. For this experiment, we use a 2.66 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC has 512 Mbytes of DDR DRAM at 266 MHz. The PCI bus is 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.22, Red-Hat 9.0).

We use LAN traces to stress-test a single sensor running a modified version of *snort* that, in addition to basic signature matching, provides the hooks needed to coordinate with the IXP1200 as well as the APE and payload sifting heuristics. We replay the traces from a remote system through the IXP1200 at different rates to determine the *maximum loss-free rate* (MLFR) of the sensor. For the purpose of this experiment, we connected a sensor to the second Gigabit Ethernet interface of the IXP1200 board.

The measured throughput of the sensor for signature matching using APE and Earlybird is shown in Table 8.1. The throughput per sensor ranges between 190 Mbit/s (APE) and 268 Mbit/s (payload sifting), while standard signature matching can be performed at 225 Mbit/s. This means that we need at least 4-5 sensors behind the IXP1200 for each of these mechanisms. Note, however, that these results are rather conservative and based on unoptimized code, and thus only serve the purpose of providing a ballpark figure on the cost of anomaly detection.

False positive vs. detection rate trade-offs We determine the workload that is generated by the AD heuristics, by measuring the false positive rate. We also consider the trade-off between false positives and detection rate, to demonstrate how the AD heuristics could be tuned to increase detection rate in our shadow honeypot environment. We use the payload sifting implementation from [16], and the APE algorithm from [127]. The APE experiment corresponds to a tightly-coupled shadow server scenario, while the payload sifting experiment examines a loosely-coupled shadow

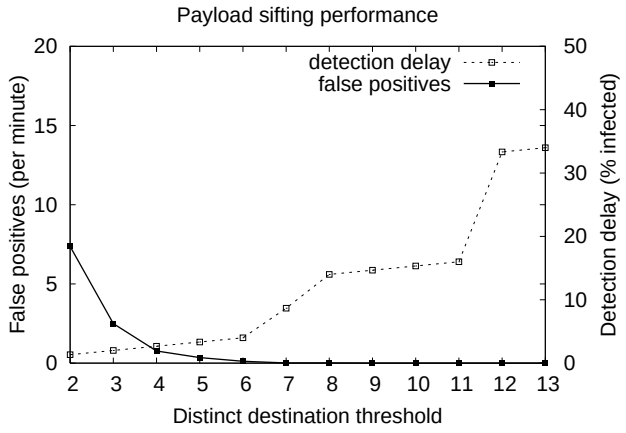


Figure 8.11: **FPs for payload sifting**

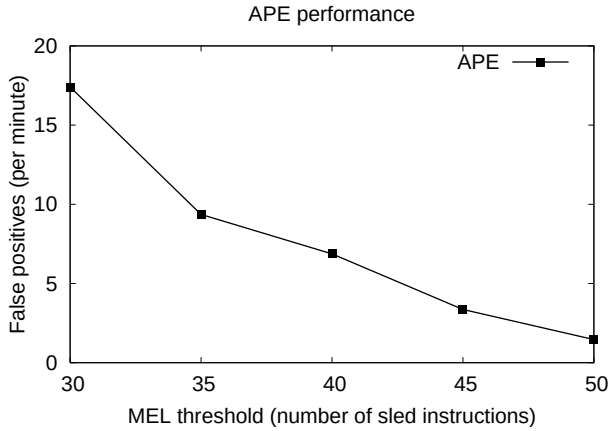


Figure 8.12: **FPs for APE**

honeypot scenario that can be used for worm detection.

We run the modified snort sensor implementing APE and payload sifting on packet-level traces captured on an enterprise LAN with roughly 150 hosts. Furthermore, the traces contain several instances of the Welchia worm. APE was applied on the URIs contained in roughly one-billion HTTP requests gathered by monitoring the same LAN.

Figure 8.11 demonstrates the effects of varying the distinct destinations threshold

of the content sifting AD on the false positives (measured in requests to the shadow services per minute) and the (Welchia worm) detection delay (measured in ratio of hosts in the monitored LAN infected by the time of the detection).

Increasing the threshold means more attack instances are required for triggering detection, and therefore increases the detection delay and reduces the false positives. It is evident that to achieve a zero false positives rate without shadow honeypots we must operate the system with parameters that yield a suboptimal detection delay.

The detection rate for APE is the minimum sled length that it can detect and depends on the sampling factor and the MEL parameter (the number of valid instructions that trigger detection). A high MEL value means less false positives due to random valid sequences but also makes the heuristic blind to sleds of smaller lengths.

Figure 8.12 shows the effects of MEL threshold on the false positives. APE can be used in a tightly coupled scenario, where the suspect requests are redirected to the instrumented server instances. The false positives (measured in requests to the shadow services per minute by each of the normal services under maximum load) can be handled easily by a shadow honeypot. APE alone has false positives for the entire range of acceptable operational parameters; it is the combination with shadow honeypots that removes the problem.

A publicly available exploit for the libpng vulnerability was used to generate a PNG image which when processed by the libpng library causes it to execute arbitrary code. The exploit's shellcode executes /bin/sh, but for our purposes we changed it to execute a simple program that records the fact that the exploit has worked. The exploit requires some tuning to work.

We wrote an HTTP protocol decoder to run over Snort that captures the objects returned by web servers to web clients. The APE detection heuristic is applied

on captured objects and, if positive, a simple server is started that will replay the response and an instrumented Mozilla Firefox instance is directed to this server.

The loosely-coupled client case is an intrusion *detection* system, as opposed to the tightly-coupled server case which prevents attacks. Therefore, we expect the experiments to demonstrate detection of the malicious content. Indeed, the buffer overflow attack contained in the image triggers the APE detection heuristic causing the system to replay the possible attack. The image causes the instrumented browser to crash and so it is identified as malicious.

The APE detection heuristic searches for sufficiently long sequences of executable instructions. Unfortunately, many innocent data sequences satisfy this criterion. For example, a sufficiently long sequence of A characters raises a false alarm. Downloading a text file including such a string does cause the system to replay the false positive using the instrumented browser. But the positive is correctly identified as a false alert.

8.6 Limitations

There are three limitations of the shadow honeypot design presented in this paper that we are aware of. First, the effectiveness of the rollback mechanism depends on the proper placement of calls to *transaction()* for committing state changes, and the latency of the detector. The detector used in this paper can instantly detect attempts to overwrite a buffer, and therefore the system cannot be corrupted. Other detectors, however, may have higher latency, and the placement of commit calls is critical to recovering from the attack. Depending on the detector latency and how it relates to the cost of implementing rollback, one may have to consider different approaches. The trade-offs involved in designing such mechanisms are thoroughly examined in the

fault-tolerance literature (c.f. [51]).

Second, the loosely coupled client shadow honeypot is limited to protecting against relatively static attacks. The honeypot cannot effectively emulate user behavior that may be involved in triggering the attack, for example, through DHTML or Javascript. The loosely coupled version is also weak against attacks that depend on local system state on the user's host that is difficult to replicate. This is not a problem with tightly coupled shadows, because we accurately mirror the state of the real system. In some cases, it may be possible to mirror state on loosely coupled shadows as well, but we have not considered this case in the experiments presented in this paper.

Finally, we have not explored in depth the use of feedback from the shadow honeypot to tune the anomaly detection components. Although this is likely to lead to substantial performance benefits, we need to be careful so that an attacker cannot launch blinding attacks, *e.g.,* "softening" the anomaly detection component through a barrage of false positives before launching a real attack.

8.7 Application Communities

8.8 Motivation

Software monocultures have been identified as a major source of problems in today's networked computing environments [57, 54, 118]. Monocultures act as force amplifiers for attackers, allowing them to exploit the same vulnerability across thousands or millions of instances of the same application across the network. Such attacks have the potential to rapidly cause widespread disruption, as evidenced by several incidents over the last few years [8, 5, 7, 4]. The severity of the problem has fueled research behind introducing diversity in software systems. However, creating a large number

of different systems manually [21] not only presents certain practical challenges [58] but can result in systems that are not diverse enough [70, 28].

As a result, recent research has focused on creating artificial diversity, by introducing "controlled uncertainty" in one of the system parameters that the attacker must know (and control) in order to carry out a successful attack. Such parameters include, but are not limited to, the instruction set [64, 23, 27], the high-level implementation [94], the memory layout [24], the operating system interface [40] and others, with varying levels of success [106, 87]. However, running different systems in a network creates its own set of problems involving configuration, management, and certification of each new platform [135, 19]. In certain cases, running such multi-platform environments can decrease the overall security of the network [90].

Given the difficulties associated with artificial diversity and the pervasive nature of homogeneous software systems, can we identify a scenario in which a homogeneous software base can be used to improve security and reliability, relative to a single instance of an application? Specifically, given a large number of almost identical copies of the same application running autonomously, is it possible to employ a collaborative distributed scheme that improves the *overall* security of the group?

To answer this question, we introduce the concept of an Application Community[1] (AC), a collection of almost-identical instances of the same application running autonomously across a wide area network. Members of an AC collaborate in identifying previously unknown (zero day) flaws/attacks and exchange information so that such failures are prevented from re-occurring. Individual members may succumb to new flaws; however, over time the AC should converge to a state of immunity against that specific fault. The system learns new faults and adapts to them, exploiting the AC

[1] To our knowledge, the term first appeared in the title of the DARPA Application Communities Workshop, in October 2004. This paper expands on our short paper that introduced some of the concepts that we explore in depth here [75].

size to achieve both coverage (in detecting faults) and fairness (in distributing the monitoring task).

This definition raises several questions. First, is the approach feasible and, if so, for what types of faults? Second, how expensive can the monitoring, coordination, and reaction mechanisms be, and is it possible to share the burden equitably across the AC members? What is the performance impact of the additional computation on individual AC members? How small can an AC be to achieve coverage *and* share fairness at the same time? Finally, how can this scheme be achieved in the presence of mutually untrusted (or possibly subverted) participants?

We do not attempt to answer all of these questions in this paper, although we outline possible directions for future research. Instead, we provide a high-level analysis of the basic parameters that govern an Application Community. We then apply this analysis in a prototype AC that is targeted against remotely exploitable software vulnerabilities and input-data-driven faults. We use the Selective Transactional EMulator (STEM) technique from [113] both for fault-monitoring and immunization. Members of the AC emulate different "slices" of the application, monitoring for low-level failures (such as buffer overflows or illegal memory accesses). When a fault is detected by an member, the relevant information is broadcast to the rest of the AC. Members may verify the fault and use STEM on the identified vulnerable code slice, possibly combining this with input filtering. Our scheme also takes into consideration input from code analysis tools that identify specific code sections as potentially more vulnerable to attacks. Because of the use of STEM, it is possible to wrap the necessary functionality "around" existing applications, without requiring source code modifications.

Our analysis indicates that AC's are an achievable goal. Specifically, we analyze the effects of risk assessment and the impact of protection mechanisms on the overall

workload for the AC. We determine that a reasonably-sized application (e.g., the Apache web server) requires an AC of about 17,000 members, assuming a normal (random) distribution of faults. Our experimental evaluation of Apache shows that an AC can be a practical method of protection; in the best case, an AC of size 15,000 can execute Apache with a performance degradation of only 6% at each member. A small AC of 15 hosts can execute Apache with a performance degradation of approximately 73%. This paper makes the following novel contributions:

- Introduce the concept of an Application Community as a way to exploit large-scale homogeneous software environments towards improving the security of the AC's members.
- Present the various parameters that define an Application Community and analytically explore the various tradeoffs among them.
- Illustrate the feasibility of the AC concept by implementing and experimenting with a prototype geared towards detecting and immunizing software against *previously unknown* general software failures and vulnerabilities.

8.9 Application Communities

An Application Community (AC) is a collection of congruent instances of the same application running autonomously across a wide area network, whose members cooperate in identifying previously unknown flaws or vulnerabilities. By exchanging information, the AC members can prevent the failure from manifesting in the future. Although individual members may be susceptible to new failures, the AC should eventually converge on a state of immunity against a particular fault, adding a dimension of learning and adaptation to the system. The size of the AC impacts both coverage

(in detecting faults) and fairness (in distributing the monitoring task). An AC is composed of three main mechanisms, for monitoring, communication, and defense, respectively.

The purpose of the monitoring mechanism is the detection of previously unknown (zero day) software failures. There exists a plethora of work in this area, namely, using the compiler to insert run-time safety checks [46], "sandboxing" [56], anomaly detection [18] and content-based filtering [3]. While shortcomings may be attributed to each of the approaches [32, 136, 121], when they are considered within the scope of an AC a different set of considerations need to be examined. Specifically, the significance of the security versus performance tradeoff is not as important as the the ability to employ the mechanism in a distributed fashion. The advantage of utilizing an AC is that the use of a fairly invasive mechanism (in terms of performance) may be acceptable, since the associated cost can be distributed to the participating members. By employing a more invasive instrumentation technique, the likelihood of detecting subversion and identifying the source of the vulnerability is increased. The monitoring mechanism in our prototype is an instruction-level emulator that can be selectively invoked for arbitrary segments of code, allowing us to mix emulated and non-emulated execution inside the same execution context [113].

Once a failure is detected by a member's monitoring component, the relevant information is distributed to the AC. Specifically, the purpose of the communication component is the dissemination of information pertaining to the discovery of new failures and the distribution of the work load within the AC. The choice of the communication model to be employed by an AC is subject to the characteristics of the collaborating community such as size and flexibility. The immediate trade-off associated with the communication model is the overhead in messages versus the latency of the information in the AC. In the simplest case, a centralized approach is arguably

the most efficient communication mechanism, however, there are a number of issues associated with this approach. If there is a fixed number of collaborating nodes, a secure structured overlay network [65, 37] can be employed with exemption from the problems associated with voluminous joins and leaves. If nodes enter and leave the AC at will, a decentralized approach may be more appropriate. Efficient dissemination of messages is outside the scope of this paper, but has been the topic of research, *e.g.,* [15].

The final component of our architecture is responsible for immunizing the AC against a specific failure. Ideally, upon receiving notification of an experienced failure, individual members independently confirm the validity of the reported weakness and create their own fix in a decentralized manner thus reducing issues regarding trust. At that point, each member in the AC decides autonomously which fix to apply in order to inoculate itself. As independent verification of an attack report may be impossible in some situations, a member's action may depend on predefined trust metrics. Depending on the level of trust among users, alternative mechanisms may be employed for the adoption of universal fixes and verification of attack reports. In the case of systems where there is minimal trust among members a voting system can be employed at the cost of an increased communication overhead. Finally, given that a fix could be universally adopted by the AC, special care must be placed in minimizing the performance implications of the immunization.

The inoculating approach that can be employed by the AC is contingent on the nature of the detection mechanism and the subsequent information provided on the specific failure. The type of protection can range from statistical blocking, behavioral or structural transformation. For example, IP address and content filtering [3], code randomization [64] and emulation [113] may be used for the protection of the AC. For the defense component in our experimental prototype, we use the STEM instruction-

level emulator as described in Chapter 5.

8.10 Analysis

Here, we present an analysis of the properties that govern the AC. Subsection 8.10.1 explains the calculations that affect the size of the AC based on the parameters we list in Table 8.2. We consider the problem of distributing work to the AC members in Subsection 8.10.2 and present some simple approaches to addressing it. Subsection 8.10.2 also defines the general form of the work distribution problem, which we term the AC-CALLGRAPH-KNAPSACK problem. In addition, we outline a strategy for solving this problem that optionally takes into consideration member-local policy. Subsection 8.10.3 briefly discusses the probability of catching new faults by duplicating monitoring responsibilities. Subsection 8.10.4 presents the results of our analysis and simulations.

To make our analysis concrete, we consider an AC aimed at low-level software attacks and faults (*e.g.,* buffer overflows, illegal memory dereferences, exceptions arising from illegal instruction operands, and other faults that cause process termination). ACs protecting against different types of failures are possible; we do not consider them further in this paper, except to the extent that our analysis apply to such systems.

Work Overview We formalize the notion of total work in the AC, W, as a function of both the cost of the monitoring mechanisms and the perceived vulnerability of each function. The *actual* work done can be calculated by two runtime metrics: (a) the number of machine instructions executed by the function during a request, and (b) the amount of real time that a function takes to execute a request. Each metric has advantages and drawbacks. For example, while instruction count is an intuitive unit and is straightforward to measure, there is a clear difference in computation

Variable	Description	Variable	Description
N	total AC members needed	F	set of application functions
n	the size of F	E	set of edges for F
G	directed call graph of (F, E)	W	the total amount of work
Z	the base unit of work	C	a cost function
M	the set of AC members	m_i	the i^{th} member of M
f_i	the i^{th} member of F	c_i	the total cost of executing f_i
x_i	the performance cost of f_i	v_i	the risk cost of executing f_i

Table 8.2: Various parameters and data sets for an Application Community. The risk score and performance score for each function combine to define the amount of work in the system. To be fair to each member, an equivalent amount of resources must be allocated to the monitoring of some subset of functions.

between 100 logical "AND" operations and 100 floating point "MUL" operations (everything else, like data dependencies and structural hazards, being equal). On the other hand, using only timing information can obscure the effects of nondeterminism or interaction with other systems even though it may provide a more realistic sense of system response or throughput.

Our main focus is on calculating the amount of work in the system and determining the level of resources needed to achieve both a *fair coverage* and a *full coverage*. That is, we wish to determine an assignment of monitoring tasks that dictates an equal amount of work for each member of the AC while simultaneously guaranteeing that all functions in an application are being monitored. If the size of the AC is already fixed, then W dictates how much work each member should do. If it is not yet fixed, then W serves as a lower bound on the size of the "optimally fair" AC. If the value of "fairness" is predetermined, falling below the minimum set of AC members means that we must either reduce coverage to maintain fairness or reduce fairness to maintain coverage. If fairness means that each node does an equal amount of work, the system can degrade gracefully.

8.10.1 Work Calculation

The cost, c_i, of executing each f_i is a function of the amount of computation present in f_i (we denote this computation as x_i) and the amount of risk present in f_i (we denote this risk as v_i). All the information (an annotated call graph of a profiling run) needed to perform the analysis is present at each member of the AC. The calculations can be kept in a form similar to Table 8.3.

The calculation of x_i can be driven by at least two different metrics: o_i, the raw number of machine instructions executed as part of f_i, or t_i, the amount of time spent executing f_i. Since the cost of certain functions (as noted above) may not be easy to extrapolate from total instructions executed, the experimental evaluation in Section 8.11 uses the running time of a function as a measure of x_i, but this analysis will assume either metric may be used. Both o_i and t_i can vary as a function of time or application workload according to the application's internal logic[2]. For example, an application may perform logging or cleanup duties after it passes a threshold number of requests. Code that normally lies dormant would then be executed. Future work will explore functions that approximate x_i's value at a given time for either metric (o_i or t_i), as either parameter may change during the lifetime of an AC (*e.g.,* due to hardware or software upgrades).

The risk factor is somewhat harder to characterize, as it is more likely to vary during runtime and it is not clear how to classify risk in terms of execution time or number of machine operations. We approximate the risk by a simple scaling factor α based on a statistical measure of vulnerability introduced by the CoSAK project[3]. Other measures (*e.g.,* static analysis tools) may be used; exploring the range of risk

[2]In order to gain confidence in the value of x_i, we determine x_i over a range of requests to see if the application somehow varies the amount of instructions it executes based on the number of requests it has handled so far.

[3]http://serg.cs.drexel.edu/projects/cosak/

metrics is interesting future work.

Let v_i represent a vulnerability (or risk) score for f_i. This v_i may be the result of a complex function that calculates risk or may be a simple scalar factor α. Its purpose is to weight a function such that more members monitor it. Let $T = \sum_{i=1}^{n} x_i$. If we express the relative cost of executing each f_i as some cost function $c_i = C(f_i, x_i, v_i)$, then the total amount of work in the system can be represented by the equation: $W = \sum_{i=1}^{n} C(f_i, x_i, v_i)$.

We provide a cost function in two phases. The first phase calculates the cost due to the amount of computation for each f_i. The second phase normalizes this cost and applies the risk factor v_i to determine the final cost of each f_i and the total amount of work in the system. If we let $C(f_i, x_i) = \frac{x_i}{T} * 100$, then we can normalize each cost by grouping a subset of F to represent one unit of work. Membership in this subset can be arbitrary, but is meant to provide a flexible means of defining what a work unit translates to in terms of computational effort. A good heuristic is to group the k lowest cost functions together and declare the sum of their work as the base work unit, Z. Every other function's cost is normalized to this work unit, and r_i represents the relative weight of each f_i with respect to Z. As a result, we know that $W = N_{base} = \sum_{i=1}^{n} r_i$ represents the total number of AC members needed to obtain full coverage of an application when we only consider performance.

However, we still have to account for the measure of a function's vulnerability (or alternatively, the risk level of executing the function). We can treat the vulnerability score of a function as a discrete variable with a value of α (where α can take on a range of values according to the amount of risk). Thus,

$$v_i = \begin{cases} \alpha & \text{if } f_i \text{ is vulnerable, } \alpha > 1; \\ 1 & \text{if } f_i \text{ is not vulnerable.} \end{cases} \tag{8.1}$$

Given the scaling factor v_i for each function, we can determine the total amount of work in the system and the total number of members needed to monitor every function is $W = N_{vuln} = \sum_{i=1}^{n} v_i * r_i$

f_i	x_i	r_i	v_i	T	$C(f_i, x_i)$	$r_i * v_i$
a()	100	1	α_1	600	16	α_1
b()	200	2	α_2	600	33	$2\alpha_2$
c()	300	3	α_3	600	50	$3\alpha_3$

Table 8.3: An example of AC work calculation. Each member of the AC can calculate this table independently. Here, the AC is executing an application with three functions. The choice of α is somewhat arbitrary and can vary based on the context of a particular function.

8.10.2 Work Distribution

After each AC member has a clear idea of the amount of work in the system, work units (slices) must be distributed to each member. In the simplest scenario, a central controller simply assigns approximately $\frac{W}{N}$ work units to each node. A more robust method of work distribution would be for each AC member to autonomously determine their work set. Each member can simply iterate through the list of work units, flipping a coin weighted with the value $v_i * r_i$. If the result of the flip is "true" then the member adds that work unit to its work set. A member stops when its total work reaches $\frac{W}{N}$. Such an approach offers statistical coverage of the application. A more elegant method of work distribution is possible; since a full treatment of it is beyond the scope of this paper, we only provide an overview of the approach.

Distributed Bidding The problem of assigning work to individual members in the AC can be seen as an instance of the general KNAPSACK problem. We call this problem the AC-CALLGRAPH-KNAPSACK problem. For the call graph G, each node has a particular weight ($v_i * r_i$ from above). The problem is then to assign some

subset of the weighted nodes in F to each member of M such that each member does no more than $\frac{W}{N}$ work. We can relax the threshold constraint to be approximately $\frac{W}{N}$ within some tunable range ϵ. Thus, ϵ is a measure of the fairness of the system. Once the globally fair amount of work $\frac{W}{N}$ is calculated, each AC member should be able to adjust their workload ϵ by bargaining with other AC members via a distributed bidding process.

Two additional considerations impact the assignment of work units to AC members. First, we would like to preferentially allocate work units with higher weights, as these work units likely have a heavier weight due to an increased vulnerability score. Even if the weight is derived solely from the measure of performance cost, assigning more members to it is beneficial because these members can round-robin the monitoring task so that any one member does not have to assume the cost alone. Second, in some situations, the value $v_i * r_i$ will be greater than the average amount of work $\frac{W}{N}$. Achieving fairness then means that the value $v_i * r_i$ defines the quantity of AC members that must be assigned to it, and the sum of all these quantities defines the minimum number of members that must participate in an AC to achieve a fair and full coverage for a particular application.

Our algorithm works in two rounds. First, each member calculates a table similar to Table 8.3. Then, AC members enter into a distributed bidding phase to adjust their individual workload. The distributed algorithm uses tokens to bid; tokens map directly to the number of time quanta that an AC member is responsible for emulating the execution of a particular code slice. A node will accumulate tokens by taking on extra computation. The distributed algorithm makes sure that each node should not accumulate more than the total number of tokens allowed by the choice of ϵ. Since we currently assume a collaborative AC, useful future work can analyze what can be done to protect the bidding process in the face of various threats (*e.g.*, insider

accumulating tokens) and constraints (*e.g.*, anonymity for AC members).

8.10.3 Overlapping Coverage

While "full coverage" means that every work unit (or slice) of an application is being monitored for the given time unit, it does not mean that every AC member's individual application is being fully monitored. Consider the following situation: member A is monitoring function Z, and member B is monitoring function Y. If a fault is present in function Z, B will miss it. Even though the *community* may catch the fault (by virtue of A's willingness to monitor Z), there may exist individual servers that have not yet detected the fault (*e.g.,* B, or even A if A is executing another part of the application and not Z). There is a tradeoff between the amount of individual coverage and how quickly the AC can identify a new fault.

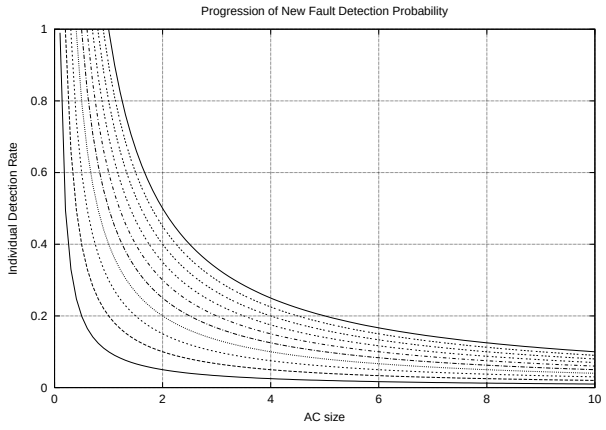


Figure 8.13: Rate of detection varies with AC size. Each line represents a member-local coverage level in 10% increments, with 10% being the bottom curve. Note how detection degrades as the AC size grows: each member is only doing a constant amount of extra coverage. However, when every member performs 100% local coverage, they regain the best chance to detect the fault, and achieve a *community probability* of 1 that the fault is detected when it first occurs; *e.g.,* an AC of size 2 with each member doing 100% coverage gives each member a probability of $\frac{1}{2}$ in detecting the fault, *i.e.,* the probability that the fault is seen by that member.

If AC members monitor more than their share (*e.g.*, A also monitors Y and B also monitors Z, then we have increased coverage to 200% and made sure that the fault, if present, is detected as quickly as possible. A similar situation is presented in Table 8.4. Assuming a uniform random distribution of new faults across AC members, the probability of a fault happening at a particular member k is: $P(\text{fault}) = \frac{1}{N}$. The probability of member k detecting the error is a function of k's individual coverage level. For Alice in Table 8.4, $P(\text{detection}) = \frac{1}{4}$. Thus, the probability of Alice detecting a new fault is the probability that the fault happens at Alice *and* that Alice detects the fault: $P(\text{fault at Alice} \wedge \text{detection}) = \frac{1}{N} * \frac{1}{4}$. Given that $N = 4$ for Alice's AC, the probability that Alice will detect a new fault is $\frac{1}{16}$. Similar calculations for each member shown in Table 8.4 show that the application has an overall new fault detection probability of $\frac{3}{8}$. If every AC member adds the missing functions to its auxiliary set, then each member has a $\frac{1}{4}$ chance of detecting the new fault: this probability is exactly $\frac{1}{N}$, their best possible chance (because the fault could happen to one of the other three). At the cost of 400% coverage, the AC has reached a probability of 1 for new fault detection. We can generalize this relationship: the probability of the AC detecting the fault is

$$P(\text{AC detect}) = \sum_{i=1}^{N} \frac{1}{N} * k_i \tag{8.2}$$

where k_i is the percentage of coverage at AC member k. Figure 8.13 shows how the AC's detection rate improves as individual member coverage tends towards 100%. As each k_i goes to 100%, Equation 8.2 becomes $\sum_{i=1}^{N} \frac{1}{N}$, or $\frac{N}{N}$, a probability of 1 that the fault is detected when it first occurs. The worst case in terms of performance is the best case in terms of rapid detection and requires $N * 100\%$ coverage.

AC Member ID	Monitored Set	Auxiliary Set
Alice	{A, F}	{∅}
Bob	{B, C}	{A, F}
Carol	{D, E}	{G, H}
David	{G, H}	{∅}

Table 8.4: A distribution of work and overlapping monitoring. Here, Alice and David choose not to do extra monitoring. However, Bob and Carol are each monitoring two more functions than strictly necessary for "fairness" and 100% *application community* coverage. Bob and Carol have increased their individual coverage from 25% to 50%, and their overall chances of detecting a new fault from $\frac{1}{16}$ to $\frac{1}{8}$.

8.10.4 Analytical Results

Our simulations explore the influence of various parameters on the amount of work in the AC: (a) the size of the application (number of functions it contains), (b) the distribution of work between functions, (c) the level of work present in each function, and (d) the policy for determining the α score (and thus v_i) for each function.

We simulate an application with a small, medium, large, and massive (20, 200, 2000, and 20000 functions, respectively) size. Similarly, the level of work for each function is small, medium, large, and massive (50, 500, 5000, 50000, respectively) normalized work units. The work level is interpreted differently for each distribution scheme. We examine three types of distributions of r_i. The *even* distribution defines an equal work level for every function. The *norm* distribution is an approximately "normal" distribution that is centered on an average value of the work level. The *skew* distribution sets the cost of most functions relatively low, but includes a few functions that account for a large part of the execution cost.

We determine α according to two policies: *exp* and *flat*. The *flat* policy applies a static factor of 10 for every function deemed vulnerable. The *exp* policy exponentially increases the value of α for "more vulnerable" functions. Every function is assigned

a default α value of 1. For both policies, we determine if a function is vulnerable or not by examining the distance of the function (in the application call graph) from a `read()` system call, using the heuristics proposed by the COSAK project. For our simulations, we assume that the path length from each function f_i to a `read()` system call is normally distributed around a mean of *log(n)*, where n is the size of the call graph, leaving exploration of different distributions as future work. Thus, our simulation assigns a normally distributed distance about this mean to each function, representing the distance from a `read()` system call. If a program is heavily saturated with `read()`'s, our simulation underestimates the weight that should be assigned to each function. However, this is not a problem, as this situation can be easily detected from the application's call graph, and every function can be scaled accordingly. The behavior of the *flat* policy is seen in Figure 8.14. Figure 8.15 shows the relationship between a program's size and the workload W of the AC. While the values for workload are quite large, they are based on a program where each function performs about 50000 work units. Our simulations for smaller workloads show the same relationship with lower total cost. We also consider a more realistic case (see Figure 8.14) for an Apache-like application: of medium size (200 functions), with a normal distribution of x_i (cost) and a *flat* policy for determining α.

8.11 Evaluation

In this section, we quantitatively measure the tradeoffs presented in Section 8.10, namely, the size of the an application community and the length of the work time quantum. Measurements are conducted using the Apache web server as the protected application and STEM as the monitoring and remediation component.

Effectiveness of STEM For our monitoring and remediation mechanism we use

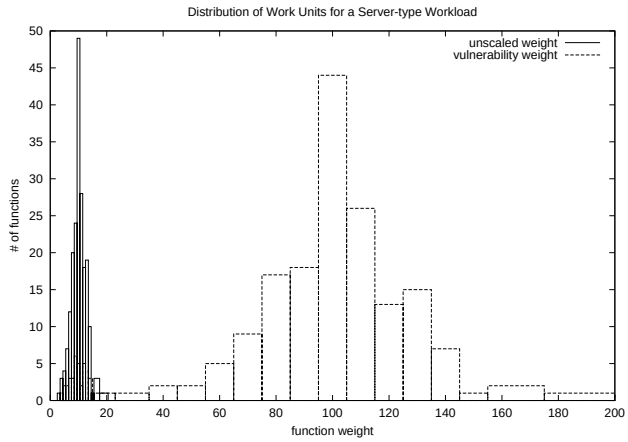


Figure 8.14: Workload scaling for a realistic parameter set. For an application of about 200 functions in size, with each function's work normally distributed around a normalized r_i of 10 and a *flat* policy for α, the workload (W) scales from 2020 to 16897.

an instruction-level emulator, *STEM*, that can be selectively invoked for arbitrary segments of code, allowing the mix of emulated and non-emulated execution inside the same execution run. The emulator allows us to (a) monitor for the specific type of failure prior to executing the instruction, (b) undo any memory changes made by the code function inside which the fault occurred, by having the emulator record all memory modifications made during its execution, and (c) simulate an error-return from said function. One of the key assumptions behind STEM is that we can create a mapping between the set of errors and exceptions that *could* occur during a program's execution and the limited set of errors that are explicitly handled by the program's code. We call this approach "error virtualization".

In a series of experiments using a number of open-source server applications including Apache, OpenSSH, and Bind, we showed that our "error virtualization" mapping assumption holds for more than 88% of the cases we examined. Testing with real attacks against Apache, OpenSSH, and Bind, we showed that this technique can be effective in quickly and automatically protecting against zero day attacks. Although

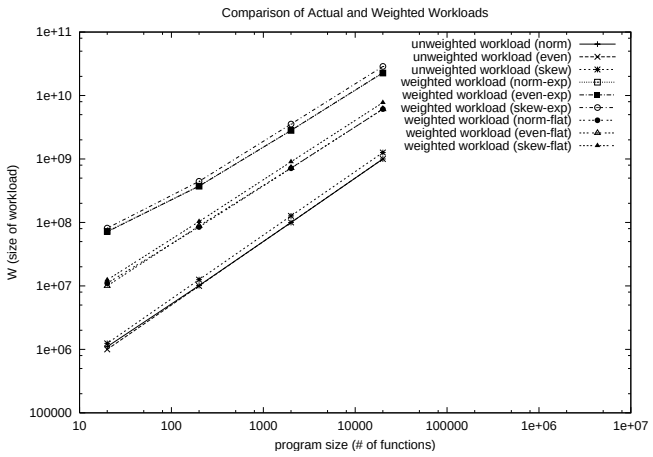


Figure 8.15: A logscale comparison of workloads given a vulnerability policy. Note that the raw values are quite high, but are drawn from data that assumes a massive value (50000) for normalized workload. More important is how the relationship between the size of the program and the total workload is affected by the choice of vulnerability policy.

full emulation is prohibitively expensive (30-fold slowdown), selective emulation imposes an overhead between 1.3 and 2, depending on the size of the emulated code segment, assuming the fault is localized within a small code region.

Slice size	Requests/sec	Number of servers
10.34	148 (27%)	15
5.24	333 (62%)	30
0.25	380 (70%)	635
0.14	497 (92%)	1135
0.04	471 (87%)	3973
0.01	506 (94%)	15893

Table 8.5: **Work-time quantums and their effects on Apache performance and AC size.**

Performance In order to understand the performance implications of an AC, we run a set of performance benchmarks which we use to explore the tradeoffs presented by our system. We employ STEM on the Apache web server and measure the overhead

Apache	trials	Mean	Std. Dev.
Normal	18	6314	847
STEM	18	277927	74488

Table 8.6: Timing of main request processing loop. Times are in microseconds. This table shows the overhead of running the whole primary request handling mechanism inside the emulator. In each trial a user thread issued an HTTP GET request.

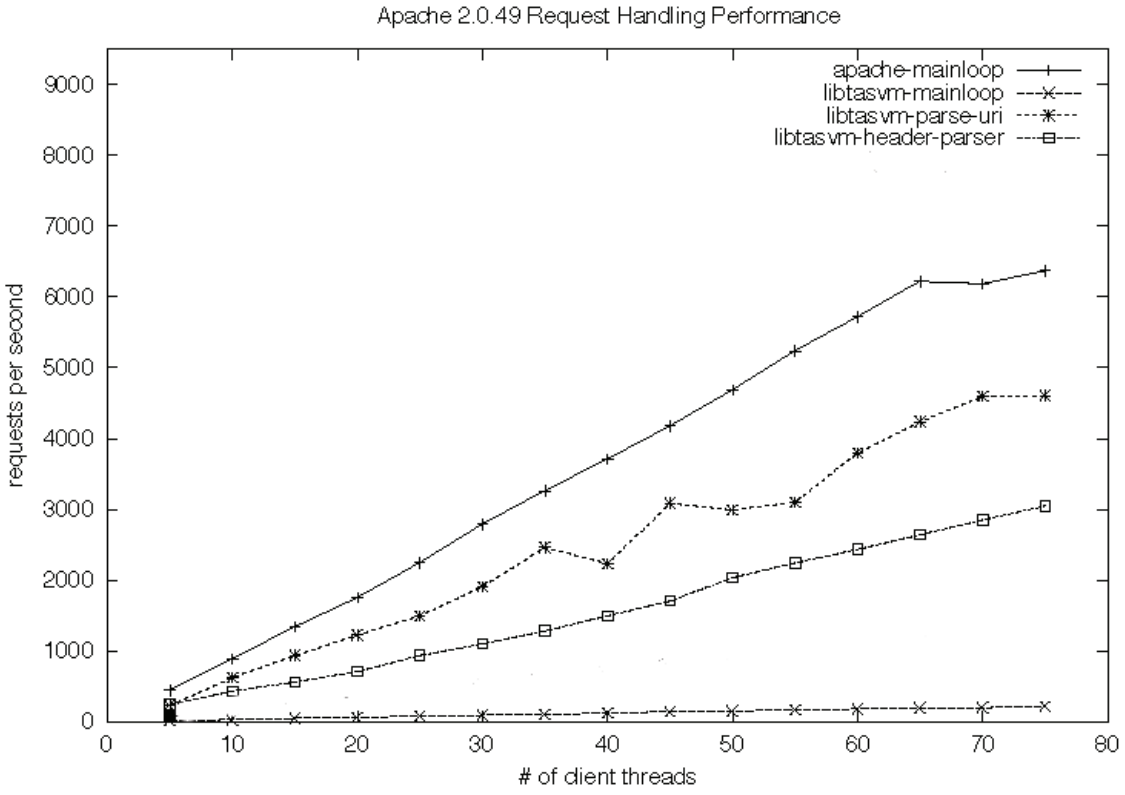


Figure 8.16: **Performance of the system under various levels of emulation. While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable.**

of our protection mechanism in terms of coverage and fairness.

Before we explore the costs associated with using an AC, we examine the cost of protecting a single instance of Apache. We demonstrate that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the

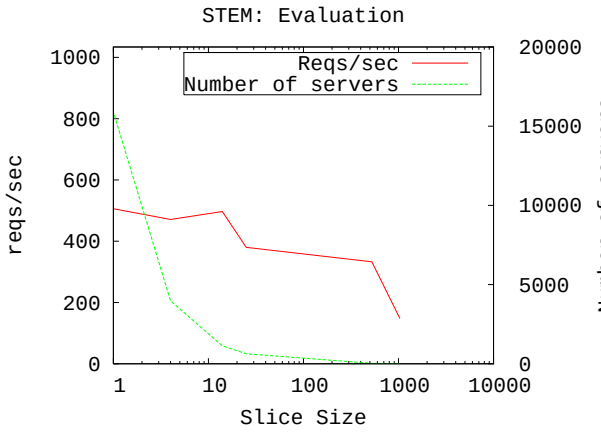


Figure 8.17: **The effect of different work-time quantums on request/sec for Apache and on the size of the AC.**

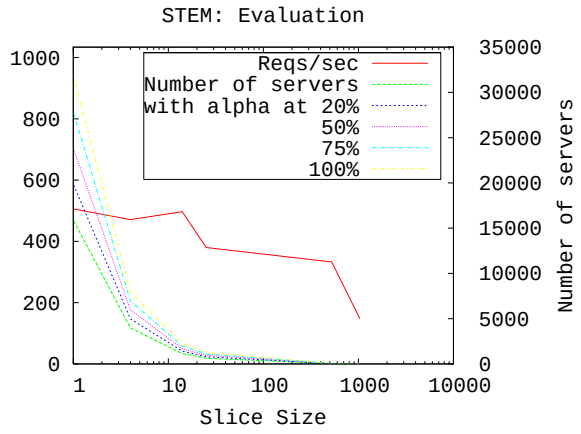


Figure 8.18: **The impact of the vulnerability index on the size of an AC.**

main request processing loop for Apache (contained in *ap_process_http_connection()*) and compared our results against a non-emulated Apache instance. In this experiment, the emulator executed roughly 213,000 instructions. The impact on performance is clearly seen in Figure 8.16, which plots the performance of the fully emulated request-handling procedure.

To get a more complete sense of this performance impact, we timed the execution of the request handling procedure for both the non-emulated and fully-emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked.

For our test machines and sample loads, Apache normally (*e.g.,* non-emulated) spent 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Table 8.6. The fully instrumented loop running in the emulator spends an average of 278 milliseconds per request in that particular code section.

To calculate the amount of work in the system and determine the level of resources needed to achieve *fair coverage* and *full coverage* as explained in Section 8.10, we first need to get a detailed analysis of the run-time characteristics of the protected

application. For this purpose, we ran a profiled version of Apache against a set of test suites and examined the subsequent call-graph generated by these tests with *gprof* and Valgrind [105]. The ensuing call trees were analyzed in order to extract the time spent doing work for each function. Using the corresponding costs, we evaluate the performance of Apache in requests per second, by employing STEM as the protection mechanism on different work time quantums to achieve full coverage.

We start with the examination of the performance of an unmodified Apache server using ApacheBench. We then proceed with the emulation of different functions representing varying work-time quantums and measure the performance overhead in terms of requests per second. Specifically, all functions invoked at least once per transaction are examined for their relative cost (time spent in function). Given the per function cost, we sample 6 functions that represent a characteristic distribution of work done per request. At that point, we wrap each function with STEM and measure the performance overhead imposed by the emulation.

The machine we chose to host Apache was a single Pentium IV at 3GHz with 1GB of memory running RedHat Linux with kernel 2.4.24. The client machine was a Pentium IV at 2 GHz with 1GB of memory running Debian Linux with kernel 2.6.8-1. For the performance evaluation of Apache, we use ApacheBench, a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance. Specifically, we look at the requests per second served by Apache for 10000 requests at a concurrency of 5. We use the average of 100 runs omitting statistical outliers.

As illustrated in Figure 8.17 and Table 8.5, we examine the use of a variety of work-time quantums on raw Apache performance and coverage. As expected, emulating large “slices” using STEM translates into lower performance for each participating

member but requires the smallest community size for 100% coverage. Concretely, using the largest work-time quantum translates into a performance degradation of 73% per member and an AC size of 15 members. As the "slice" size is reduced (using a less expensive function as the base), the performance overhead per member is decreased at the cost of a larger community. For the smallest work-time quantum, a performance overhead of 6% is experienced per member whilst the size of the AC grows to 15893. These results are very encouraging and closely follow the intuition provided in Section 8.10

Figure 8.18 illustrates the effect of varying the vulnerability index on the size of the community for 100% coverage. In this example, we double the number of servers required to cover an α region. We start with the case where 25% of the code is considered potentially vulnerable and increment the α value until the entire code base is covered. As expected, when a higher percentage of the code base is deemed vulnerable, the community needs to be larger to realize fair coverage. Note that the effect on Apache performance is linear despite an aggressive protection policy. Our experiments demonstrate that the use of an AC can alleviate the problems associated with using an invasive protection mechanism by fairly distributing work to participating members. Furthermore, we show that the flexibility of our protection mechanism can facilitate the adjustment of parameters associated with the requirements of an AC.

Chapter 9

Conclusion

9.1 Summary

In the absence of perfect software, error toleration and recovery techniques become a necessary complement to proactive approaches.

This dissertation introduced the notion of error virtualization, a software self-healing technique for detecting, tolerating and recovering from software faults in server applications. The basic premise of error virtualization is that every complex system has a well-tested core that almost always performs as expected. If the infinite set of errors that can occur in an application can be mapped into errors that are explicitly handled by the application's core then the system will be able to survive most faults.

Error virtualization operates under the assumption that there exists a mapping between the set of errors that *could* occur during a program's execution (*e.g.,* a caught buffer overflow attack, or an illegal memory reference exception) and the limited set of errors that are explicitly handled by the program's code. Thus, a failure that would cause the program to crash is translated into a return with an error code from

the function in which the fault occurred (or from one of its ancestors in the stack). Conceptually, error virtualization is a mechanism that retrofits exception-handling capabilities to legacy software. We extend the basic concept of error virtualization with introduction of error virtualization using rescue points (or ASSURE). Rescue points are locations identified in the existing application code where error handling is performed with respect to a given set of foreseen (by the programmer) failures.

To evaluate the efficacy and applicability of error virtualization as a recovery strategy for software self-healing systems we developed three prototype systems for Linux: DYBOC, STEM and ASSURE.

DYBOC is a software self-healing system that generates source-code-level patches for C/C++ server applications in response to detected attacks or program failures. DYBOC automatically instruments all statically and dynamically allocated buffers in an application so that any buffer overflow or underflow attack will cause transfer of the execution flow to a specified location in the code, from which the application can resume execution using error virtualization. DYBOC introduces the concept of *execution transactions:* the hypothesis is that function calls can be treated as transactions that can be aborted when a buffer overflow is detected, without impacting the application's ability to execute correctly. Nested function calls are treated as sub-transactions, whose failure is handled independently. DYBOC uses standard memory-protection features available in all modern operating systems and is highly portable. DYBOC is a stand-alone tool, which simply needs to be run against the source code of the target application.

STEM, extends the functionality of DYBOC through the use of binary-level emulation. STEM is a binary supervision framework that is implemented as an instruction-level emulator, that can be selectively invoked for arbitrary segments of code. This tool permits the execution of emulated and non- emulated code inside the same pro-

cess. The use of selective emulation allows STEM to handle a wide variety of software failures, ranging from remotely exploitable vulnerabilities to more mundane bugs that cause abnormal program termination. More importantly, STEM provides better support for execution transactions through its ability to completely reverse the effects of computation at instruction-level granularity.

For our evaluation of error virtualization using rescue points, we designed and implemented ASSURE. ASSURE uses rescue-points in conjunction with a process-level checkpoint/restart mechanism to implement recovery in source-code available and binary-only environments in Linux. We use existing quality assurance testing techniques to generate known bad inputs to an application, in order to identify candidate rescue points. On detecting a fault for the first time, ASSURE uses a replica (shadow) of the application to determine what rescue points can be used most effectively to recover future program execution. Once ASSURE verifies that it has produced a fix that repairs the fault, it dynamically patches the running production application to self-checkpoint at the rescue point. If the fault occurs again, the ASSURE rolls back the application to the checkpoint, and uses the application's own built-in error-handling code to recover from the fault and correctly clean up internal and external state.

We have implemented all of these systems, and extensively measured their performance and functionality. In support of the error virtualization hypothesis, we examined error virtualization on several server application using both synthetic fault injection and real bugs using DYBOC, STEM and ASSURE. Our examination of error virtualization against real bugs and vulnerabilities shows that our technique can be used to detect and recover execution for all examined cases. Using comprehensive fault injection, we show that error virtualization can be used to recover over 90% of the injected faults with an acceptable performance overhead. Our results validate our

hypothesis that existing application code can be harnessed to facilitate with recovering program execution for a variety of faults. They also show that error virtualization can be used as an effective recovery strategy for software self-healing systems.

Chapter 10

Future Work

The work presented in this dissertation, served as an initial exploration of error virtualization as a recovery strategy for self-healing systems. Several open issues and possible future directions remain.

Recovery in multi-process/multi-threaded environments One of the most pressing directions for future research is how to effectively support recovery in multi-process/multi-threaded systems. Previous work, including the work presented in this thesis, has focused on single-process systems. The advent of multi-core architectures, and subsequently the need to exploit execution parallelism dictates that self-healing strategies should be able to handle recovery in multi-threaded systems in an efficient manner.

Supporting recovery in multi-process/multi-threaded environments presents several challenges. In the case of ASSURE and multi-process environments, creating and using rescue points becomes a complicated affair. The key issue with rescue points is that the checkpoints are always initiated by the application, since they must occur at designated safe locations. With multiple (cooperative) processes, the state must be

obtained not only in a globally consistent manner, but also with all the participating processes stationed at some suitable point. By "cooperative" we refer to processes that share resources, such as shared memory, or some portion of their state, and therefore must agree on that state at any point in time. On the other hand, if the processes are independent, then they can be handled independently one by one as in the single-process case.

First, recovering program execution of a single thread/process should not come at the expense of the execution of other threads/processes. This is of particular concern if threads/processes are not idempotent, *i.e.*, if they use some form of IPC for communication.

Recovery execution shepherding A common criticism of recovery techniques that explore software elasticity is the lack of guarantees, in maintaining program semantics upon recovery. Although solving this problem is theoretically impossible (halting problem), several techniques can be used to improve semantic equivalency. One approach that I wish to explore is that of shepherding execution along a predictable (or previously seen) recovery path.

Similarly to the case where we identify candidate rescue points using predictable bad input, we feed an application with input that will invoke appropriate error handling code and monitor the data-flow recovery path. Using the path as a guide, we will monitor how the application proceeds when recovering program execution using a rescue point. Specifically, we will attempt to construct safe recovery paths by looking at information derived from control- and data-flow graphs using the construction cost and measuring the rate of false-positive and false-negative indications as guidance metrics.

10.0.1 Long-term Goals

My long-term research goal is to make software as secure and robust as possible. This problem is fundamentally challenging given the complexity of software. Unfortunately, development methodologies have not kept up with the increased level of software complexity and show no signs of doing so in the future. In future work, I wish to explore the concept of *software elasticity:* the ability of regular code to recover from certain types of failures when low-level faults are masked by appropriate instrumentation. One way to support this notion is to provide a mechanism that allows programmers to experiment with such "patches" with impunity. Towards this goal I plan to examine the general problem area: robust patch application. Knowledge of where faults might occur helps in both prevention and toleration. In support of this, I am going to explore hybrid static/dynamic techniques for bug detection.

Robust patching Given that many system bugs often lead to system vulnerabilities, system administrators need to perform the highly dextrous task of juggling between system down-time and installing updates. Unfortunately, in many cases the cost or complexity of launching another machine to test the effects of patches is non-trivial.

With these constraints in mind, I plan to explore a new approach for testing application patches prior to definitive deployment. Using binary runtime injection techniques, binaries can be modified so that when program execution reaches a program segment that has been affected (modified) by an issued patch, the program "forks" speculative execution threads that cannot interfere with each other. This approach raises numerous questions: what is required to support light-weight speculative execution; how does one evaluate misbehavior (and course of action upon detection); what information should be communicated to developers. Future research

in this area will try to answer these questions. I believe that having the ability to test patches in production environments with impunity will be an indispensable tool in the system administrators arsenal.

Bug detection The prevalence of software bugs, as a primary cause of system failures, has fuelled recent research efforts on software bug detection. In particular, a number of tools have been developed that examine applications for the existence of bugs using both static and dynamic analysis techniques. The advantage of static analysis tools lies in their ability to detect bugs prior to deployment and provide complete code coverage. Unfortunately, and despite the existence of a plethora of tools, the inaccuracy associated with static analysis has stymied adoption leaving programmers to use arcane bug finding techniques. In the OpenBSD world, when a new software bug is detected, it is common practice to create a regular expression that describes the vulnerable statement and search the source tree for similar uses.

The problem with this approach, and more powerful static analysis techniques , is that they can only be used to look for known classes or instances of problems. Furthermore, the inherent ambiguity associated with parsing source code statically, often aggravates the problem resulting in a disabling number of false positives. Dynamic bug detection techniques are used during program execution and offer a different set of trade-offs. They rely on the analysis of run-time information in order to detect bugs which, translates into higher accuracy at the cost of performance and limited program coverage.

One area I want to explore is bug detection using hybrid static and dynamic analysis techniques for source and binary environments. Specifically, I want to explore: how code can be abstracted to represent functionality rather than string matching; how dynamic detection techniques can be used to drive static source and binary analy-

sis; efficient algorithms for searching a large code base; mechanisms for automatically protecting or even fixing vulnerable code. Such techniques can help solve a considerable problem: protect against vulnerable code reuse by attackers and help developers create comprehensive patches.

Bibliography

[1] i-Bench. http://http://www.veritest.com/benchmarks/i-bench/default.asp.

[2] Symantec. Internet security threat report. http://www.symantec.com/enterprise/threatreport/index.jsp.

[3] Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Technical report, Cisco Systems, Inc.

[4] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.

[5] CERT Advisory CA-2001-26: Nimda Worm. http://www.cert.org/advisories/CA-2001-26.html, September 2001.

[6] ApacheBench: a complete benchmarking and regression testing suite. http://freshmeat.net/projects/apachebench/, July 2003.

[7] Cert Advisory CA-2003-04: MS-SQL Server Worm. http://www.cert.org/advisories/CA-2003-04.html, January 2003.

[8] CERT Advisory CA-2003-21: W32/Blaster Worm. http://www.cert.org/advisories/CA-2003-20.html, August 2003.

[9] Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing Could Allow Code Execution. http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx, September 2004.

[10] US-CERT Technical Cyber Security Alert TA04-217A: Multiple Vulnerabilities in libpng. http://www.us-cert.gov/cas/techalerts/TA04-217A.html, August 2004.

[11] National Vulnerability Database. http://nvd.nist.gov/statistics.cfm, April 2006.

[12] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[13] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[14] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Exe: A system for automatically generating inputs of death using symbolic execution. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*. ACM Press, 2006.

[15] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *Comput. J.*, 34(6):534–541, 1991.

[16] P. Akritidis, K. Anagnostakis, and E. P. Markatos. Efficient content-based fingerprinting of zero-day worms. In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2005.

[17] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targetted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 129–144, August 2005.

[18] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[19] D. Aucsmith. Monocultures Are Hard To Find In Practice. *IEEE Security & Privacy*, 1(6):15–16, November/December 2003.

[20] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 45–55, August 2004.

[21] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[22] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 167–179, February 2005.

[23] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.

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

[25] M. Bhattacharyya, M. G. Schultz, E. Eskin, S. Hershkop, and S. J. Stolfo. MET: An Experimental System for Malicious Email Tracking. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 1–12, September 2002.

[26] John Boyd. OODA Loop:Observation, orientation, decision, action. http://www.mindsim.com/MindSim/Corporate/OODA.html.

[27] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the* 2^{nd} *Applied Cryptography and Network Security Conference (ACNS)*, pages 292–302, June 2004.

[28] S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[29] Frederick P. Brooks. *Mythical Man-Month.* Addison-Wesley, first edition, 1975.

[30] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the 2007 Computer Security Foundations Symposium*, Venice, Italy, July 2007.

[31] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[32] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[33] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005.

[34] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. pages 125–132, 2001.

[35] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[36] George Candea and Armando Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[37] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, 2002.

[38] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 97–106, Washington, DC, USA, 2000. IEEE Computer Society.

[39] Subhachandra Chandra and Peter M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 91, Washington, DC, USA, 2002. IEEE Computer Society.

[40] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[41] Jorgen Christmansson* and Ram Chillarege**. Generation of an error set that emulates software faults - based on field data. Technical report, 1996.

[42] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Kone, and Ashley Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the* 3^{rd} *Workshop on Network Processors and Applications (NP3)*, February 2004.

[43] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, volume 39, pages 133–147, New York, NY, USA, December 2005. ACM Press.

[44] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–199, August 2001.

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

[46] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, January 1998.

[47] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, August 2003.

[48] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honepots. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, October 2004.

[49] 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.

[50] Brian Demsky and Martin C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, October 2003.

[51] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[52] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000.

[53] L. Garber. New Chips Stop Buffer Overflow Attacks. *IEEE Computer*, 37(10):28, October 2004.

[54] D. E. Geer. Monopoly Considered Harmful. *IEEE Security & Privacy*, 1(6):14 & 17, November/December 2003.

[55] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[56] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[57] G. Goth. Addressing the Monoculture. *IEEE Security & Privacy*, 1(6):8–10, November/December 2003.

[58] J. Gray and D. Siewiorek. High-availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[59] Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, 2005.

[60] B. Hernacki, J. Bennett, and T. Lofgren. Symantec Deception Server Experience with a Commercial Deception System. In *Proceedings of the* 7^{th} *International Symposiun on Recent Advanced in Intrusion Detection (RAID)*, pages 188–202, September 2004.

[61] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceeding of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[62] A.J. Malton J.R. Cordy, T.R. Dean and K.A. Schneider. Source Transformation in Software Engineering using the TXL Transformation System. *Journal of Information and Software Technology*, 44:827–837, 2002.

[63] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

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

[65] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, pages 61–72, August 2002.

[66] H. Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 271–286, August 2004.

[67] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In 19^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[68] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding, 2002.

[69] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[70] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[71] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.

[72] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 251–261, October 2003.

[73] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX Annual Technical Conference*, pages 323–336. USENIX, 2007.

[74] J. G. Levine, J. B. Grizzard, and H. L. Owen. Using Honeynets to Protect Large Enterprise Networks. *IEEE Security & Privacy*, 2(6):73–75, November/December 2004.

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

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

[77] M. E. Locasto, A. Stavrou, G. F. Cretu, A. D. Keromytis, and S. J. Stolfo. Quantifying Application Behavior Space for Detection and Self-Healing. Technical Report CUCS-017-06, Columbia University Computer Science Department, April 2006.

[78] Andrew J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. *Computer Languages*, 19(3):157–168, 1993.

[79] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[80] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12), December 1990.

[81] M.Vojnovic and A.Ganesh. On the effectiveness of automatic patching. In *WORM '05*, November 2005.

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

[83] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.

[84] James Newsome, David Brumley, and Dawn Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (SNDSS)*, February 2006.

[85] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13^{th} *Annual Network and Distributed Systems Security Symposium*, 2006. http://www.cs.cmu.edu/~dbrumley/.

[86] D. Nicol and M. Liljenstam. Models of active worm defenses. In *Proceedings of the Measurement, Modeling and Analysis of the Internet Workshop (IMA)*, January 2004.

[87] A. J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 121–131, October 2004.

[88] Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[89] PaX Project. Address space layout randomization, Mar 2003. http://pageexec.virtualave.net/docs/aslr.txt.

[90] Vassilis Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the* 8^{th} *USENIX Security Symposium*, August 1999.

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

[92] Niels Provos. A Virtual Honeypot Framework. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 1–14, August 2004.

[93] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In Andrew Herbert and Kenneth P. Birman, editors, *SOSP*, pages 235–248. ACM, 2005.

[94] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)*, January 2003.

[95] M. Rinard. Acceptability-oriented computing, 2003.

[96] 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)*, December 2004.

[97] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[98] Martin C. Rinard. Living in the comfort zone. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 611–622, New York, NY, USA, 2007. ACM.

[99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, November 1999. (software available from *http://www.snort.org/*).

[100] A. Rudys and D. S. Wallach. Transactional Rollback for Language-Based Systems. In *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2001.

[101] A. Rudys and D. S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[102] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, September 2003.

[103] D. Scott. Assesing the costs of application downtime, May 1998.

[104] R. Sengupta, Offenberg J. D., Fixsen D. J., Katz D. S., Springer, Stockman P. L., Nieto-Santisteban H. S., Hanisch M. A., Mather R. J., and J. C. Software Fault Tolerance for Low-to-Moderate Radiation Environments. In *ASP Conf. Ser., Vol. 238, Astronomical Data Analysis Software and Systems X*, 2001.

[105] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-linux. http://developer.kde.org/~sewardj/.

[106] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[107] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, pages 201–216, August 2001.

[108] S. Sidiroglou, Y. Giovanidis, and A.D. Keromytis. A Dynamic Mechanism for Recovery from Buffer Overflow attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005.

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

[110] S. Sidiroglou and A. D. Keromytis. Countering Network Worms Through Automatic Patch Generation. *IEEE Security & Privacy*, 2005. (to appear).

[111] Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh. Using rescue points to navigate software recovery. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 273–280, Washington, DC, USA, 2007. IEEE Computer Society.

[112] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Technical Conference*, April 2005.

[113] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building A Reactive Immune System for Software Services. In *Proceedings of the* 11^{th} *USENIX Annual Technical Conference*, pages 149–161, April 2005.

[114] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the* 6^{th} *Symposium on Operating Systems Design & Implementation (OSDI)*, December 2004.

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

[116] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the* 18^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[117] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.

[118] Mark Stamp. Risks of Monoculture. *Communications of the ACM*, 47(3):120, March 2004.

[119] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.

[120] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[121] A. Steven. Defeating compiler-level buffer overflow protection. *USENIX ;login:*, 30(3):59–71, June 2005.

[122] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.*, 38(5):85–96, 2004.

[123] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[124] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the* 12^{th} *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 265–278, February 2005.

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

[126] Top Layer Networks. http://www.toplayer.com.

[127] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[128] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2002.

[129] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 131–144, New York, NY, USA, 2007. ACM.

[130] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of 2007 EuroSys Conference*, 2007. http://www.cs.cmu.edu/~dbrumley/.

[131] 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 Conference*, August 2004.

[132] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advanced in Intrusion Detection (RAID)*, pages 201–222, September 2004.

[133] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the* 12^{th} *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[134] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 29–44, August 2004.

[135] J. A. Whittaker. No Clear Answers on Monoculture Issues. *IEEE Security & Privacy*, 1(6):18–19, November/December 2003.

[136] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 123–130, February 2003.

[137] Cynthia Wong, Chenxi Wang, Dawn Song, Stan Bielski, and Gregory R. Ganger. Dynamic quarantine of internet worms. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 73, Washington, DC, USA, 2004. IEEE Computer Society.

[138] Jian Wu, Sarma Vangala, Lixin Gao, and Kevin Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, pages 143–156, February 2004.

[139] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 146–165, October 2004.

[140] Hoijin Yoon and Byoungju Choi. Component customization testing technique using fault injection technique and mutation test criteria. pages 71–78, 2001.

[141] Andreas Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

[142] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, October 2003.

Appendix A

Analysis on Real Bugs

A.0.2 Apache (mod_ftp_proxy)

The first bug we examine is an illegal memory dereference (NULL dereference) on the `mod_ftp_proxy` Apache module. If there is a malicious remote FTP server and someone uses Apache and this proxy to connect to that FTP server, the server can reply to "LIST" with a directory listing showing directories or ordinary files with no spaces whatsoever in the line. There is a `strrchr()` call that does not check for a NULL return, leading to an illegal memory dereference. This effectively causes a denial-of-service attack on the web server.

We first initiate rescue point discovery by profiling Apache under bad input. For this case, in addition to standard web server fuzzing, we enable profiling of the ftp proxy module by crafting illegal FTP command requests. We then trigger the vulnerability by sending a request that would proxy the FTP request to our malicious FTP server. We emulate the FTP server by writing an *inetd*-driven Perl script. The malicious request induces a segmentation violation (SIGSEGV) in `ap_proxy_send_dir_filter()`. From this point onwards, ASSURE proceeds in an

automated fashion.

The malicious input is re-run against the set of possible rescue points until Apache is able to recover from the fault and continue servicing requests correctly. The first rescue point ASSURE examines is `ap_pass_brigade()`, with a rescue return value of -1. Although that rescue point allows the application to recover from the SIGSEGV, it leaves the server in a state where it is unable to service further requests properly. The first rescue point to pass evaluation is `ap_proxy_ftp_handler()` with a 502 (HTTP "Proxy Error") at a rescue depth of 2.

A.0.3 Apache (mod_rewrite)

The second Apache bug examined is caused by an off-by-one error in the `mod_rewrite` module. Specifically, the vulnerability resides in the mod_rewrite module's LDAP handling that allows for potential memory corruption when an attacker exploits certain rewrite rules. An attacker may exploit this issue to trigger a denial-of-service condition through a SIGSEGV. Similar to the mod_ftp_proxy bug, we profile the server using bad input for the mod_rewrite module. In this case, the failure manifests in `escape_absolute_uri()` and the rescue point is at `hook_uri2file()` (rescue depth of 1) with a rescue return value of -1.

A.0.4 Apache (mod_include)

The third Apache bug that we examined resides in the mod_include module. The `get_tag()` function contains a buffer overflow that can be exploited by a local attacker, by creating a specially crafted html page, to allow for the execution of arbitrary code with the privileges of the affected Apache server.

The problem presents itself when the affected module attempts to parse mod_include-

specific tag values. A failure to properly validate the lengths of user-supplied tag strings before copying them into finite buffers facilitates the overflow.

Again, we profile Apache using input that exercises the mod_include module. The fault manifests in the `get_tag` function which returns a `char *`. As mentioned previously, the rescue-point discovery algorithm only considers functions that return pointers as potential rescue points if the observed value during the profiling indicates that a NULL value was used to propagate failures. The profiling information indicated that `get_tag` used NULL as a return value and was subsequently successfully tested as the first rescue point.

A.0.5 openLDAP modrdn

OpenLDAP is prone to a remote denial-of-service (DoS) vulnerability that can by triggered by calls to the modify operation that include NOOP control directives connected to a BDB back-end. The vulnerability only occurs on operations that should succeed (i.e. should return the NOOP error code) thus is does not hinder application profiling using bad input.

A successful rescue point for this bug was found in the first function in the rescue-graph, `bdb_modrdn()`, which is used to interface to the BDB back-end database. The rescue point return a integer value to indicate success/failure codes. The rescue value that was used in this case was 80, which represent the LDAP_OTHER error condition.

A.0.6 postgreSQL

buffer overflow vulnerability has been reported for PostgreSQL. Reportedly, PostgreSQL doesn't properly handle overly large integer arguments given to the lpad() and rpad() funtions. The functions are lpad() and rpad() found in the file, src/backend/utils/adt/oracle_c

and serve to pad an existing text string with another up to a given length. This vulnerability only affects data bases that were created using special international encodings. For example, databases that were created using a 'UNICODE' encoding are vulnerable to this issue.

A.0.7 MySQL

MySQL contains a buffer overflow that may allow a remote, authenticated attacker to execute arbitrary code on a vulnerable server. Command packets are sent to the MySQL server to issue instructions to that server. One such command packet type is COM_TABLE_DUMP, which according to the MySQL Internals Manual is used by a slave server to get the master table. MySQL fails to properly validate user-controlled parameters within COM_TABLE_DUMP packets. If an attacker sends a series of specially crafted COM_TABLE_DUMP packets to a vulnerable MySQL server, that attacker may be able to take over the system by exploiting the buffer overflow vulnerability.

For this bug, we profile MySQL by sending a set of malformed commands and requests for non-existent tables. The vulnerability is triggered by using the proof-of-concept exploit published by Stefano Di Paola. The exploit sends a set of commands to the server that causes the statically allocated buffer `key` in function `open_table()` to be overflown. The rescue point for this vulnerability is found to reside in the same function as the bug (rescue depth of 0), with a rescue return value of 0. ASSURE's ability to replay a sequence of inputs was instrumental in dealing with this bug and illustrates a case where ASSURE is able to deal with faults that rely on a sequence of inputs.

A.0.8 sshd

Next, we examine a vulnerability in OpenSSH sshd. The vulnerability is remotely exploitable and may allow for unauthenticated attackers to obtain root privileges. The conditions are related to the OpenSSH SSH2 challenge-response mechanism. They are present when the OpenSSH server is configured at compile-time to support BSDAUTH or SKEY authentication. It is possible for attackers to exploit the vulnerability by constructing a malicious response. As this occurs before the authentication process completes, it may be exploited by remote attackers without valid credentials. Successful exploitation may result in the execution of shellcode or a denial of service.

We examine application behavior by creating a set of bad authentication requests to the server. The exploit code we use takes advantage of a heap overflow vulnerability in buffer `response` located in function `input_userauth_info_response()`. The rescue point that allows the application to recover execution and correctly continue to serve subsequent requests is in `auth2_challenge_start()`, with a rescue return value of 0.