ASSURE: Automatic Software Self-healing Using REscue points

Stelios Sidiroglou, Oren Laadan, Carlos R. Perez, Nicolas Viennot,
Jason Nieh, and Angelos D. Keromytis

Columbia University

{stelios, orenl, carlosrene, nv2195, nieh, angelos}@cs.columbia.edu

Abstract

Software failures in server applications are a significant problem for preserving system availability. We present ASSURE, a system that introduces *rescue points* that recover software from unknown faults while maintaining both system integrity and availability, by mimicking system behavior under known error conditions. Rescue points are locations in existing application code for handling a given set of programmer-anticipated failures, which are automatically repurposed and tested for safely enabling fault recovery from a larger class of (unanticipated) faults. When a fault occurs at an arbitrary location in the program, ASSURE restores execution to an appropriate rescue point and induces the program to recover execution by virtualizing the program's existing error-handling facilities. Rescue points are identified using fuzzing, implemented using a fast coordinated checkpoint-restart mechanism that handles multi-process and multi-threaded applications, and, after testing, are injected into production code using binary patching. We have implemented an ASSURE Linux prototype that operates without application source code and without base operating system kernel changes. Our experimental results on a set of real-world server applications and bugs show that ASSURE enabled recovery for all of the bugs tested with fast recovery times, has modest performance overhead, and provides automatic self-healing orders of magnitude faster than current human-driven patch deployment methods.

Categories and Subject Descriptors K.6.5 [*Security and Protection*]: Invasive Software

General Terms Security, Reliability, Design

Keywords Software Self-healing, Error Recovery, Reliable Software, Binary Patching, Checkpoint Restart

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

1. Introduction

Software errors and vulnerabilities in server applications are a significant problem for preserving system integrity and availability. The accepted wisdom is to use 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. However, experience has shown that it is very hard to achieve bug-free software [18]. As a result, even under the best of circumstances, buggy software is deployed and developers face a constant and time-consuming battle of creating and releasing patches fast enough to fix newly discovered bugs. Patches can take days if not weeks to create, and it is not uncommon for systems to continue running unpatched applications long after an exploit of a bug has become well-known [23].

In the absence of perfect software, many approaches have been proposed in an effort to tolerate and recover from software errors. These approaches have considered various options for recovering from a fault, including filtering malicious input [8, 16, 31], crashing to prevent system exploitation [10], rebooting or restarting the system or parts of the system [6], returning arbitrary values to mask faults [25], replaying in a changed execution environment [22], or slicing off faulty application functionality [26, 27].

However, previous approaches suffer from one or more problems that can limit their effectiveness and utility in practice. These problems include the inability to deal with polymorphic input behavior or common application scenarios involving encryption, inability to handle deterministic bugs, applicability only to memory errors and not other kinds of bugs, lack of assurances of correct program execution, inability to work with unmodified application binaries and requiring application or operating system modifications, lack of validation in working with more realistic and commonly used multi-process or multi-threaded application deployments, significant performance overhead during normal application execution or recovery, and inability to preserve system integrity and availability after a fault occurs.

To address these problems, we present ASSURE, a system that provides *A*utomatic *S*oftware *S*elf-healing *U*sing *RE*scue points. ASSURE introduces *rescue points*, loca-

tions in existing application code for handling programmer-anticipated failures which are automatically repurposed and tested for safely enabling general fault recovery. When a fault occurs at an arbitrary location in the program, ASSURE restores execution to the closest rescue point and induces the program to recover execution by virtualizing and using its existing error-handling facilities.

Rescue points virtualize error handling by creating a mapping between the (potentially infinite) set of errors that could occur during a program's execution (*e.g.,* a detected buffer-overflow attack, or an illegal memory-dereference exception) and the limited set of errors that can be handled by the program's code. Thus, a failure that would cause the program to crash is translated into a "return with an error" from an error-handling function along the execution path in which the fault occurred. By reusing existing error-handling facilities and automatically testing them before use in production code, rescue points can reduce the chance of unanticipated execution paths, thereby making recovery more robust. Rescue points do not simply mask errors. Instead, they "teleport" the faults to locations that are known or suspected, with high probability, to handle faults correctly (including correct program-state cleanup).

ASSURE first identifies candidate rescue points in an application offline using fuzzing [15], then implements, tests, and deploys rescue points online in response to the occurrence of faults using an Observe Orient Decide Act (OODA) feedback loop [2]. During the application's production use, ASSURE monitors the application for faults. If a fault is detected for the first time, ASSURE uses a replica of the application (a copy of the application and all of its state) to determine which candidate rescue point can be most effectively used. The selected candidate rescue point is then implemented using exception handling and an operating system checkpoint-restart mechanism that handles multi-process and multi-threaded applications. ASSURE confirms that it has repaired the fault by re-running the application against the event sequence that apparently caused the failure, as well as against already known good and bad input. Upon success, ASSURE uses runtime binary injection to insert the rescue point into the application running on the production server. When the fault occurs again on the production server, the application uses the rescue point to roll back state to the rescue point, where the program is forced to return an error, imitating the behavior observed during fuzzing. The system is designed to operate without human intervention to minimize reaction time.

We have implemented an ASSURE Linux prototype that operates without application source code and without base operating system kernel changes. To demonstrate its effectiveness, we have evaluated our prototype on a wide range of real-world server applications and bugs. We focus on server applications because they typically have higher availability requirements and also tend to have short error-propagation distances [25] that lend themselves to our approach. Our experimental results show that ASSURE identified and used rescue points to successfully recover from all of the bugs tested. Unlike other approaches, our evaluation validated ASSURE's ability to recover in the presence of bugs for application deployments in typical multi-process and multi-threaded configurations while running widely used workloads for measuring performance. Our performance measurements showed that ASSURE recovered from faults in just a few milliseconds for all applications and incurred less than 10% performance overhead during normal execution. Furthermore, our results show that ASSURE provides automatic and tested self-healing of legacy applications in a few seconds to minutes depending on the level of testing desired, orders of magnitude faster than current human-driven patch deployment methods.

ASSURE provides several key advantages over other approaches: (1) It operates without human intervention. (2) It does not require access to or modification of application or operating system kernel source code. (3) It does not require additional network infrastructure for deployment. (4) It handles polymorphic input behavior and encrypted traffic. (5) It goes beyond just handling memory errors and is better at dealing with deterministic bugs. (6) It works for both multi-threaded and multi-process applications. (7) It uses application error handling semantics and includes a testing phase to provide greater assurance of correct application execution in the presence of faults. (8) It incurs modest performance overhead. The unique end result is automatic self-healing of software services from what were previously unknown and unforeseen software failures, maintaining both system integrity and availability.

This paper presents the design, implementation and evaluation of ASSURE. Section 2 discusses related work. Section 3 presents the ASSURE system architecture and discusses in detail the concept of rescue points. Section 4 presents experimental results. Finally, we present some concluding remarks.

2. Related Work

Many approaches have been proposed to tolerate and recover from software errors. Schemes such as StackGuard [10] and ASLR [20] focus on protection from code injection attacks and preventing a system from being exploited due to a bug. They preserve system integrity by terminating the application when a fault occurs, but are unable to maintain system availability.

Reboot techniques, including whole program restart [29], software rejuvenation [13], and micro-rebooting [6], attempt to return a system to a clean state before or after encountering a fault. Whole program restart can take a long time, resulting in substantial application down-time. Micro-rebooting can be faster by only restarting parts of the system, but requires a complete rewrite of applications to compart-

mentalize failures. None of these techniques effectively deal with deterministic bugs, since these may recur post-restart.

Checkpoint-restart techniques [3, 11] can be used in a manner similar to whole program restart, but can provide faster restart times since restarts are done from a checkpoint. When used in this way, these techniques still do not handle deterministic bugs, since these bugs will still occur after restarting. Other uses of checkpoint-restart in conjunction with running multiple program versions have also been proposed [3] which may survive deterministic bugs if failures occur independently. However, they incur prohibitive costs for most applications in terms of developing, maintaining, and running multiple application versions at the same time.

Automatic signature generation for network intrusion detection systems [21, 17] defends against vulnerabilities by filtering inputs to weed out attacks. A key problem is that such signatures are quite susceptible to false positives, especially for polymorphic attacks. Furthermore, polymorphic behavior has been shown to be far too varied to be modeled effectively by signatures [28].

Vigilante [8] improved on network input filtering through automated creation of host-based input filters. Host-based filters offer improved accuracy and higher tolerance for detecting semantically equivalent inputs. Unfortunately, they require protocol-specific parsers and cannot handle complex rules, encryption and specific application state. Shield [32] and VSEF [4] 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.

Rx [22] uses a checkpoint-restart mechanism in conjunction with mechanisms to change the execution environment in an effort to recover from bugs. However, previous work [7] found that over 86% of application faults are independent of the operating environment and entirely deterministic and repeatable, and that recovery is likely to be successful only through application-specific or application-aware techniques. While Rx examines a broader scope for changing the environment including dropping malicious input requests, dropping requests have been shown to be ineffective in practice due to polymorphic behavior [28]. Rx attempts to mask the manifestation of faults to the client, but 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 use of a proxy complicates the use of the growing numbers of applications that employ encryption. Rx requires operating system kernel changes, which serves as another impediment to deployment. Finally, Rx does not address consistency issues in checkpointing and restarting applications involving multiple processes.

Sweeper [31] combines the Rx checkpoint-restart mechanism and proxy with VSEF. 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).

Acceptability-oriented computing [9, 24, 25] promotes the idea that current software development efforts might be misdirected, based on the observation that certain regions of a program can be neglected without adversely affecting the overall availability of the system. Failure-oblivious computing [25] is a speculative recovery technique that builds on a compiler to insert code to deal with memory-writes to unallocated memory by virtually expanding the target buffer. Such a capability aims to provide a more robust fault response than simply crashing albeit at significant performance overhead, ranging from 80% up to 500% for a variety of different applications.

Selective Transactional EMulation (STEM), as used in the Reactive Immune System [27], is a speculative recovery technique by two of the authors that identifies the function in which a fault occurs, then selectively emulates that function and potentially others within a larger scope to return error values in an attempt to recover from the fault. STEM uses the notion of error virtualization to mean the return of an heuristic-based error value from a function in which a fault occurs. This is very different from the notion of rescue point error virtualization used in ASSURE, which reuses existing error handling code in applications and returns values based on profiling those functions to mimic system behavior under controlled and anticipated error conditions. Unlike STEM, ASSURE does not require source code, works with multi-process and multi-thread applications, provides significant improvements in system performance, and demonstrates better fault recovery across a broader range of applications and vulnerabilities.

3. ASSURE Architecture

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 using rescue points 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.

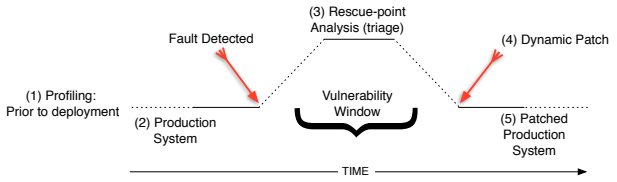

Figure 1: System overview: ASSURE lifecycle

Figure 1 illustrates the high-level operation of ASSURE. Prior to its deployment, the application is profiled 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 application state and maintains an execution log (including 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, in which the application may need to resort to full application restarts to recover service. While our system may require some downtime for the analysis phase, Section 4.3 shows that this is in the order of seconds, and the cost is amortized as it is incurred once per new fault. Combining our approach with techniques such as micro-rebooting [6] is a topic of future research.

Once a candidate rescue point is selected, ASSURE confirms that it is suitable for deployment by verifying that it satisfies three criteria: survivability, correctness and performance. 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. A rescue point is *efficient* if the performance implications of protection do not impose significant run-time overhead. Survivability is verified by replaying the sequence of events that apparently triggered the fault. Correctness is verified using extensive testing that is tailored for the specific operation of the application. Efficiency is optimized by also using performance as a metric in deciding which rescue point is more appropriate.

As soon as a suitable rescue point is verified, 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.

3.1 Rescue Point Example

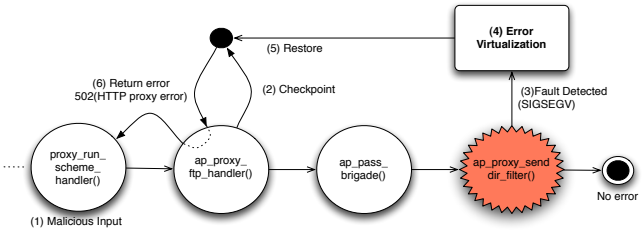

Figure 2: Self-healing using rescue points: Example with real Apache bug (ASF Bug 40733)

Figure 2 illustrates ASSURE's self healing on a real bug in the Apache web server. A description of the bug is given in Table 1 . 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_filter()`, and results in a memory fault (SIGSEGV). ASSURE intercepts the first occurrence of the fault, identifies `ap_proxy_ftp_handler()` as a suitable rescue point, and instruments the Apache server as follows. Whenever the patched server enters the function `ap_proxy_ftp_handler()`, ASSURE takes a checkpoint of the server state, and allows the server to proceed with 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 component. The error virtualization component analyzes the fault information and rolls back the server state back to the rescue point in `ap_proxy_ftp_handler()`. 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"). Using this example, we now describe in further detail how ASSURE discovers, selects, creates, tests, and deploys rescue points.

3.2 Rescue Point Discovery

To discover candidate rescue points, ASSURE profiles an application prior to deployment using dynamic analysis with fuzzing. The intuition is that there exists a set of programmer-tested application points that are routinely used to handle expected errors which can be discovered by learning how an application responds to "bad" input under controlled conditions. For example, we would like to see how a program normally handles errors when stress-tested by quality assurance tests. This knowledge is then used in the future to map previously unseen faults to a set of observed fault behaviors.

ASSURE instruments applications offline to discover candidate rescue points by inserting monitoring code at every function's entry and exit points using the run-time injection capabilities of Dyninst [5], 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*.

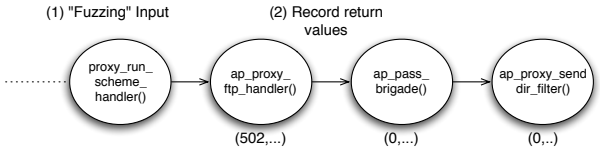

Figure 3: Rescue-trace creation

Figure 3 illustrates part of a rescue-trace for the example in Section 3.1. It shows a summarized execution trace that includes three functions: `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.

3.3 Fault Detection and Reproduction

ASSURE continuously monitors the execution of the application in the production system to detect application failures and misbehavior, and records sufficient information about a fault so that it can be reproduced to determine a suitable rescue point. To detect failures and misbehavior, ASSURE employs a variety of fault-detection mechanisms. It is not tied to any particular fault-detection mechanism, and is compatible with any such mechanism that simply notifies the system of the occurrence of a fault. In the example in Section 3.1, the segmentation violation indication (SIGSEGV) was used as a signal of improper memory handling. 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 [10], ASLR [20], and TaintCheck [16], and some that can detect violations to security policies [1, 12].

To reproduce a fault, ASSURE uses execution logging and periodic checkpointing to record and recreate the sequence of events that led to the manifestation of the fault. We only present an overview of the logging mechanism due to space constraints; a detailed description is beyond the scope of this paper. The checkpoint mechanism is the same as used for implementing rescue points as described in Section 3.6.

ASSURE records all inputs (between checkpoints) to application processes so that they can replayed for both multi-core and multi-processor environments. ASSURE accounts for nondeterministic execution by accurately recording all forms of interaction between processes and their execution environment and, in turn, is able to precisely replay them. Nearly all such interactions involve system calls, that can be divided into two broad categories: *with* and *without* side-effects. System calls without side-effects, such as `getpid` and `gettimeofday`, need not be re-executed. Their effect can be emulated by intercepting their return value. In contrast, system calls with side-effects (*e.g.,* `brk` and `fork`) must be replayed for their desired effect to take place. Among the latter are also system calls such as `pipe` and `write` whose effect may be visible after the replay completes and the system goes "live".

With multiple processes, ASSURE does not aim to repeat the exact scheduling order as in the original execution; rather, it tries to ensure that system calls and other events are ordered correctly by tracking their dependencies. ASSURE identifies related system calls (namely, the outcome of one depends on the execution of the other) and coordinates their execution using rendezvous points. The order of system calls is tracked during logging and then enforced during replay.

Periodic checkpointing at the production machine has the following benefits. First, it provides a snapshot of application state which, in conjunction with the execution log, can re-create the state of the application when the fault occurred. This is a critical requirement for the analysis step in order to reproduce the fault and subsequently provide a remedy. Second, it places a bound on the size of the execution log that needs to be maintained; the system only keeps track of the execution that happened since the last checkpoint. Third, it minimizes the time it takes to reproduce the fault. The system simply needs to replay the execution record since the last checkpoint. Fast fault reproduction is of vital importance to ASSURE as it reduces the vulnerability window.

3.4 Rescue Point Selection

The ability to identify, and more importantly, reproduce faults allows us to select the most appropriate rescue point for each detected failure. 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*. If the call-stack is corrupted, as in the case of a buffer overrun, it is reproduced while replaying the input that led to the failure.

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 . 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 occurring 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.

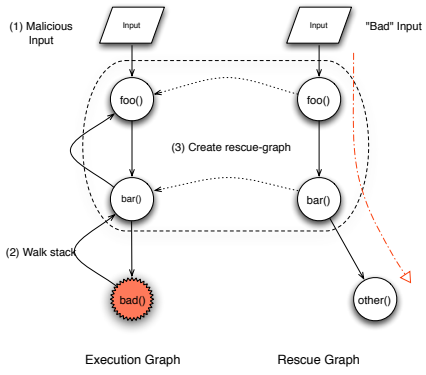

Figure 4: Rescue-graph creation

Figure 4 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. Another reason for minimizing the distance between fault and rescue point becomes apparent when dealing with multi-process/multi-threaded applications. Namely, a short distance minimizes the amount of progress non-faulty processes/threads make thus minimizing the amount of work that will need to be reverted in the case of a failure. Additionally, it reduces the chance that any externally visible communication would have occurred during this time. We measure rescue depth and rescue-to-fault distance in Section 4.2.

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 the test, ASSURE chooses the nearest active ancestor and repeats.

3.5 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 3.7. 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)` as shown in Figure 5. 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. In our example, this call is inserted in `ap_proxy_ftp_handler()`.

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

Figure 5: Rescue point capture

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.

3.6 Rescue Point Checkpoint/Rollback

To support checkpoint-rollback of cooperative processes, ASSURE places the applications inside a virtual execution environment based on Zap [14, 19]. Building on Zap, ASSURE leverages the standard interface between applications and the OS to transparently encapsulate the applications in a virtual namespace. This is essential to support the ability to continuously checkpoint, and later roll back multi-process applications, allowing them to use the same OS resource names as used before being checkpointed, even if they are mapped to different underlying OS resources upon rollback.

A rescue point must satisfy two key requirements. First, it must provide a coordinated and consistent checkpoint of multiple processes and threads and their execution environment; this is quite different from just checkpointing a single process. Second, it must have minimal impact on the application performance. To address these requirements, ASSURE takes a globally consistent checkpoint across all processes (and threads) of the application while they are stopped so that nothing can change, but then minimizes the type and cost of operations needed while everything is stopped.

The key issue with multi-process (and multi-threaded) applications is that checkpoints are always initiated by a process as they must occur at designated safe locations. Since processes share state and execution environment, they must agree on the state at any point in time. However, when a process reaches a rescue point, it will generally have to wait for a considerable amount of time for the remaining processes to also reach a suitable location. Instead, ASSURE uses a privileged process outside the execution environment to perform the consistent checkpoint of the entire application.

ASSURE stores checkpoints in main memory, eliminating the need to write the data to disk. It reduces checkpoint time due to copying memory blocks as well as the amount of memory required for the checkpoint by leveraging copy-on-write techniques. Checkpoints are associated with a context that identifies the corresponding rescue point and process. Multiple checkpoints can be can coexist for the same rescue point or for the same process. The *scope* of a checkpoint is valid until execution returns normally or following a rollback, at which point the checkpoint is discarded.

Keeping checkpoints entirely in memory allows us to not only save the state of a resource, but also keep a reference to it. ASSURE leverages this to preserve selected resources *as is* across application rollback, instead of restoring to the previous state. In particular, ASSURE uses this to eliminate the need to reset a connection between a client and the application upon a rollback, by keeping the underlying socket as is. This is particularly useful for connection-oriented services, and to processes other than the one that experienced the fault.

3.7 Rescue Point Testing

Once a candidate rescue point has been selected, 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 replays the recorded execution log that led the failure. When the fault occurs and triggers a rollback to the selected rescue point, its effects on program execution are examined. 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. Finally, the run-time performance implications of our fix are examined to ensure acceptable operation characteristics.

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.

3.8 Rescue Point Deployment

Once we have a rescue point, we want to instantiate it on the production system without delay. 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 our deployment mechanism we use Dyninst [5] for its low runtime overhead and its ability to attach and detach from already running processes. 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.

4. Experimental Evaluation

We have implemented an ASSURE prototype system for Linux. It consists of user-space utilities and loadable kernel modules for the 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 all experiments, the process was fully automated, with the exception of generating profiling information and triggering the bug. Profiling process needs to occur once per application (or be provided as part of a testing suite). 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 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. ASSURE 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.

4.1 Bug Summary

Table 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 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.

4.2 Overall Functionality Results

Table 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 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 measure 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 6 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 show the average lag time between checkpoint/rollback command issuing and completion. In detail, they 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 ex-

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 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.

amined 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 [27]. Values of 0 and -1 are often used to propagate errors but there are cases, as in Apache mod_ftp bug and openLDAP, where observed values of 502 and 80, respectively, are more appropriate values to return.

4.3 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 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, the average total patch generation time varied 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 ranged 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 [30]. We should note that ASSURE's aim is not to replace the patch-creation process but rather to add an intermediate option that administrators can use to improve system availability while waiting for manually created and thoroughly tested patches. In fact, ASSURE's testing process can be used by patch creators to augment their existing portfolio.

4.4 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 measure 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

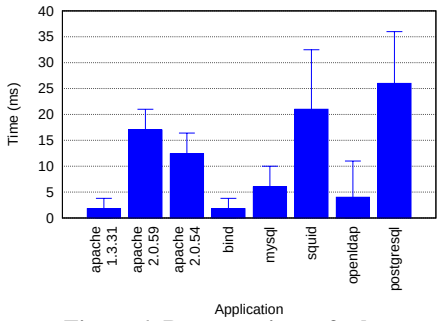

Figure 6: Rescue-point to fault

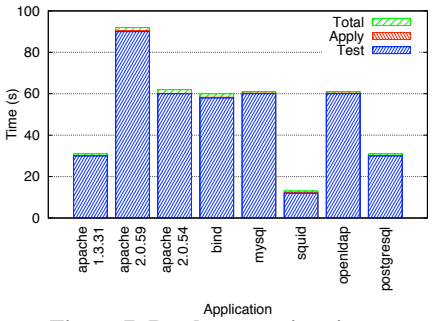

Figure 7: Patch generation time

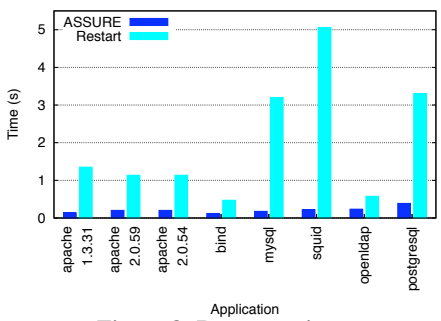

Figure 8: Recovery time

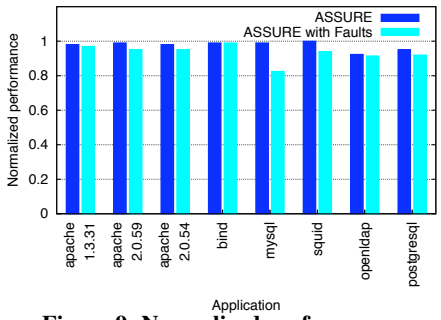

Figure 9: Normalized performance

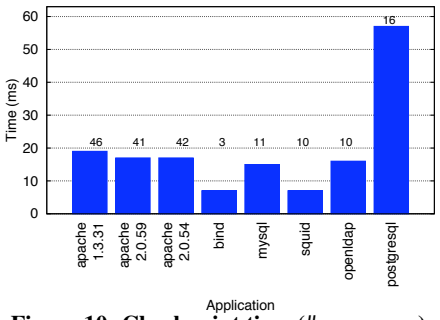

Figure 10: Checkpoint time (# processes)

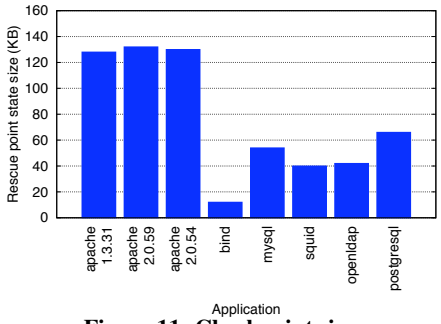

Figure 11: Checkpoint size

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 improve their performance; our measurements do not capture the negative effect of whole program restart on performance due to effectively discarding that state and caches. To measure 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 8 shows the average time required to restore execution to a rescue point, or in other words, rollback execution. As shown, restart times ranged 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 orders of magnitude 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 when the fault occurred.

To evaluate client perceived availability, we examined the number of client observed errors due to fault recovery. Specifically, we injected 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 30 second intervals. The values ranged from 1% to 10%, for a fault every 30 and 10 seconds respectively.

4.5 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 9. 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 [14]. Second, for all 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 detailed performance analysis of the cost of taking rescue points, we refer the reader to the next Section.

Also shown in Figure 9, 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.

4.6 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 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 10 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 ranged from 7 ms for Squid to roughly 50 ms, in the worst case, for `PostgreSQL`. Most values ranged between 10 ms and 20 ms which represent a modest downtime when one considers that ASSURE checkpoints multiple processes. Above each bar, we also show the average number of processes/threads checkpointed.

Figure 11 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. The checkpoint sizes ranged from 12 KB for `bind` to 130 KB for `MySQL`. These values represent the state changes between checkpoints. Our copy-on-write mechanism allows us to avoid saving full application state. The full application state sizes ranged from 20 MB for `bind` to 116 MB for `Postgres`. Since ASSURE only requires the latest checkpoint to initiate recovery, the rescue point space requirements are manageable.

5. Conclusions

ASSURE introduces *rescue points,* a new software self-healing technique for detecting, tolerating and recovering from software faults in server applications. 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. 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 ASSURE and demonstrated its effectiveness on several server applications, including web, database, domain name, and proxy servers. Our experimental results with both real-world bugs and synthetic fault injections show that our technique can be used to recover execution in most examined cases with modest operational overhead. Using an unoptimized prototype, the total automatic software healing process takes just a couple minutes, orders of magnitude faster than current human patch deployment methods. Furthermore, no application source code is required. The end result is automatic healing of software services from what were previously unknown (and unforeseen) software failures.

6. Acknowledgements

Our shepherd, Rebecca Isaacs, provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF ITR grant CNS-0426623, AFOSR MURI grant FA9550-07-1-0527 and DTO grant FA8750-06-2-0221.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS 2005)*, pages 340–353, Sept. 2005.

[2] J. Boyd. Patterns of Conflict. Unpublished Briefing, `http://www.d-n-i.net/boyd/pdf/poc.pdf`, Dec. 1986.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the* 15th *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 1–11, Dec. 1995.

[4] D. Brumley, H. Wang, S. Jha, and D. Song. Creating Vulnerability Signatures Using Weakest Preconditions. In *Proceedings of the* 20th *IEEE Computer Security Foundations Symposium*, pages 311–325, July 2007.

[5] B. Buck and J. K. Hollingsworth. An API For Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, Nov. 2000.

[6] G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the* 9th *Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 12–20, May 2003.

[7] S. Chandra. *An Evaluation of the Recovery-Related Properties of Software Faults*. PhD thesis, University of Michigan, Sept. 2000.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-To-End Containment of Internet Worms. In *Proceedings of the* 20^{th} *ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 133–147, Dec. 2005.

[9] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors In Data Structures. In *Proceedings of the* 18^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003)*, pages 78–95, Oct. 2003.

[10] J. Etoh. GCC Extension for Protecting Ppplications from Stack-smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems With Time-Traveling Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX 2005)*, pages 1–15, Apr. 2005.

[12] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, pages 191–206, Aug. 2002.

[13] N. Kolettis and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the* 25^{th} *International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 381–395, June 1995.

[14] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007)*, pages 323–336, June 2007.

[15] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[16] J. Newsome, D. Brumley, and D. Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Annual Symposium on Network and Distributed System Security (NDSS 2006)*, pages 1–15, Feb. 2006.

[17] M. Norton and D. Roelker. Snort 2.0 Protocol Flow Analyzer. Sourcefire White Paper, Apr. 2004.

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

[19] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System For Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, Dec. 2002.

[20] PaX Team. Address Space Layout Randomization, Mar. 2003. http://pax.grsecurity.net/docs/aslr.txt.

[21] V. Paxson. Bro: A System For Detecting Network Intruders In Real-Time. *Computer Networks*, 31(23-24):2435–2463, Dec. 1999.

[22] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies—A Safe Method To Survive Software Failures. In *Proceedings of the* 20^{th} *ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 235–248, Oct. 2005.

[23] E. Rescorla. Security Holes... Who Cares? In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 6–20, Aug. 2003.

[24] M. Rinard. Acceptability-Oriented Computing. In *Companion of the* 18^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, pages 221–239, Oct. 2003.

[25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 303–316, Dec. 2004.

[26] S. Sidiroglou, Y. Giovanidis, and A. Keromytis. A Dynamic Mechanism For Recovery From Buffer Overflow Attacks. In *Proceedings of the* 8^{th} *Information Security Conference (ISC 2005)*, pages 1–15, Sept. 2005.

[27] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building A Reactive Immune System For Software Services. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX 2005)*, pages 149–161, Apr. 2005.

[28] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of the* 14^{th} *ACM Conference on Computer and Communications Security (CCS 2007)*, pages 541–551, Oct. 2007.

[29] M. Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability—A Study of Field Failures In Operating Systems. In *Proceedings of the* 21^{st} *International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, June 1991.

[30] Symantec. Internet Security Threat Report. http://www.symantec.com/enterprise/threatreport/index.jsp.

[31] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-To-End System For Defending Against Fast Worms. In *Proceedings of the* 2^{nd} *European Conference on Computer Systems (EuroSys 2007)*, pages 115–128, Mar. 2007.

[32] 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 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2004)*, pages 193–204, Aug. 2004.