Building a Reactive Immune System for Software Services

Stelios Sidiroglou Michael E. Locasto Stephen W. Boyd Angelos D. Keromytis
Department of Computer Science
Columbia University in the City of New York
{*stelios,locasto,swb48,angelos*}@*cs.columbia.edu*

Abstract

We propose a new approach for reacting to 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). Our emphasis is in creating "self-healing" software that can protect itself against a recurring fault until a more comprehensive fix is applied.

Our system consists of a set of sensors that monitor applications for various types of failure and an instruction-level emulator that is invoked for selected parts of a program's code. Use of such an emulator allows us to predict recurrences of faults, and recover program execution to a safe control flow. Using the emulator for small pieces of code, as directed by the sensors, allows us to minimize the performance impact on the immunized application.

We discuss the overall system architecture and a prototype implementation for the *x86* platform. We evaluate the efficacy of our approach against a range of attacks and other software failures and investigate its performance impact on several server-type applications. We conclude that our system is effective in preventing the recurrence of a wide variety of software failures at a small performance cost.

1 Introduction

Despite considerable work in fault tolerance and reliability, software remains notoriously buggy and crash-prone. The situation is particularly troublesome with respect to services that must maintain high availability in the face of remote attacks, high-volume events (such as fast-spreading worms, *e.g.,* Slammer[1]) that may trigger unrelated and possibly non-exploitable bugs, or simple denial of service attacks. The majority of solutions to this problem fall into four categories:

- **Proactive approaches** that seek to make the code as dependable as possible, through a combination of safe languages (*e.g.,* Java), libraries [1] and compilers [13], code analysis tools [6], and development methodologies.
- **Debugging aids** whose aim is to make post-fault analysis and recovery as easy as possible for the programmer.
- **Runtime solutions** that seek to contain the fault using some type of sandboxing, ranging from full-scale emulators such as VMWare, to system call sandboxes [21], to narrowly applicable schemes such as StackGuard [7].
- **Byzantine fault-tolerance schemes** (*e.g.,* [29]) which use voting among a number of service instances to select the correct answer, under the assumption that only a minority of the replicas will exhibit faulty behavior.

The contribution of this paper is a *reactive* approach, accomplished by observing an application (or appropriately instrumented instances of it) for previously unseen failures. The types of faults we

[1] `http://www.silicondefense.com/research/worms/slammer.php`

examine consist of illegal memory dereferences, division by zero exceptions, and buffer overflow attacks. Other types of failures can be easily added to our system as long as their cause can be algorithmically determined (*i.e.,* another piece of code can tell us what the fault is and where it occurred). We intend to enrich this set of faults in the future; specifically, we plan to examine Time-Of-Check-To-Time-Of-Use (TOCTTOU) violations.

Upon detection of a fault, we invoke a localized recovery mechanism that seeks to recognize and prevent the specific failure in future executions of the program. Using continuous hypothesis testing, we verify whether the specific fault has been repaired by re-running the application against the event sequence that apparently caused the failure. Our initial focus is on automatic healing of services against newly detected faults (whether accidental or maliciously induced). We emphasize that we seek to address a wide variety of software failures, not just attacks.

For our recovery mechanism we use an instruction-level emulator, *libtasvm*, 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.

One of our key assumptions is that we can create a mapping between the set of errors that *could* occur during a program's execution and the limited set of errors that are explicitly handled by the program's code. This "error virtualization" technique is based on heuristics that we present in Section 2.4. We believe that a majority of server applications are written to have robust error handling; by virtualizing the errors, an application can continue execution even though a boundary condition that was not predicted by the programmer allowed a fault to "slip in." Our experiments with Apache, OpenSSH, and Bind validate this intuition.

Our current work focuses on server-type applications, since they typically have higher availability requirements than user-oriented applications, which can often simply be restarted upon failure. Such an approach [5] has been advocated as "micro-rebooting." Server applications often cannot be simply restarted because they are typically long running (and thus accumulate a fair amount of state) and usually contain a number of threads that service many remote users. Restarting the whole server because of one failed thread unfairly denies service to the users of unaffected threads.

Also, unlike user-oriented applications, servers operate without direct human supervision and thus have a higher need for an automated reactive system. Furthermore, it is relatively easy to replay the offending sequence of events in such applications, as these are typically limited to input received over the network (as opposed to a user's interaction with a graphical interface). We intend to investigate other classes of applications in the future.

To evaluate the effectiveness of our system and its impact to performance, we conduct a series of experiments using a number of open-source server applications including Apache, OpenSSH, and Bind. The results show that our "virtualized error" mapping assumption holds for more than 88% of the cases we examined. Testing with real attacks against Apache, OpenSSH, and Bind, we show that our technique can be effective in quickly and automatically protecting against zero-day attacks. Furthermore, although full emulation of these is prohibitively expensive, our selective emulation imposes between a 1.3 and 30 times performance overhead, depending on the size of the emulated code segment.

The remainder of this paper is organized as follows. In Section 2, we discuss our approach, including the limitations of our system and the basic system architecture. In Section 3 we briefly discuss the implementation of *libtasvm*, and Section 4 presents some preliminary performance measurements of the system. We give an overview of the related work in Section 5 and Section 6 summarizes our contributions.

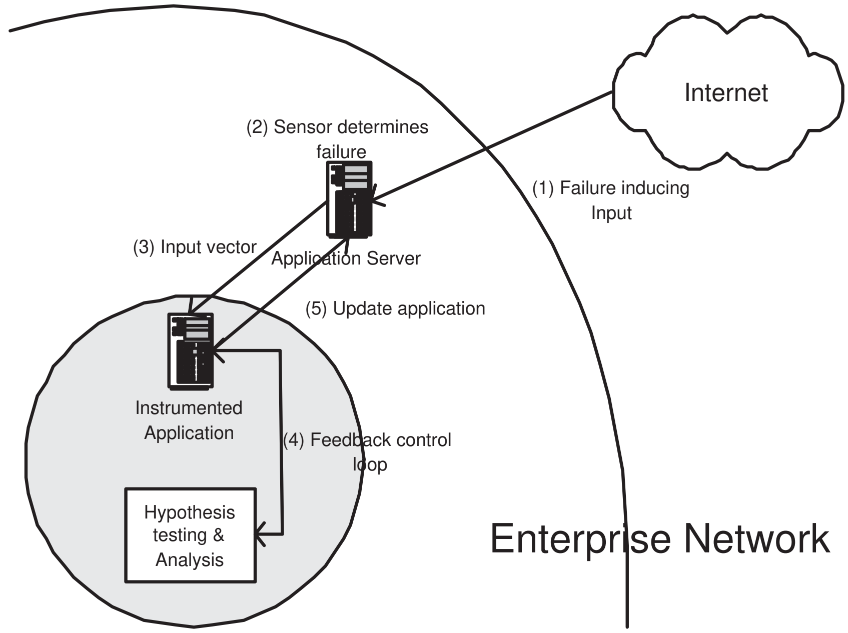

Figure 1: **Feedback control loop.**

2 Approach

Our architecture, depicted in Figure 1, uses three types of components: a sensor that monitors an application such as a web server for faults, an instruction-level emulator (*libtasvm*) that can selectively emulate "slices" (arbitrary segments) of code, and a testing environment where hypotheses about the effect of various fixes are evaluated. Note that these components can operate without human supervision to minimize reaction time.

2.1 System Overview

When the sensor detects an error in the application's execution (such as a segmentation fault), the sensor 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 3). To verify the effectiveness of the fix, the application is restarted in a test environment with the instrumentation 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) [2].

During emulation, *libtasvm* 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, *libtasvm* examines the operands and pre-determines the side effects of the instructions it emulates. Using an emulator allows us to avoid 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 made to return an error. We describe how both emulation and error virtualization are accomplished in Sections 2.3 and 2.4, respectively, and we

[2] A current limitation of our system is its inability to handle failures that occur due to extremely long inputs; this is a practical limitation, as we wish to limit the input size we need to "remember." If necessary, we can record longer input sessions for replay purposes, at the expense of storage size and testing time.

experimentally validate the error virtualization hypothesis in Section 4.

Execution now enters the testing phase. After forcing the function to return, emulation terminates. If the program then crashes, the scope of the emulation is expanded to include the parent routine, repeating as necessary. In the extreme case, the whole application could end up being emulated at a significant performance cost. However, Section 4, 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.

Note that the cost 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 3. 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 3) 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.

2.2 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 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 *libtasvm* to apply selective emulation, starting with the topmost 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 [26]. 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 vector 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, 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.

2.3 Instruction-level Emulation

For our recovery mechanism we use an instruction-level emulator, *libtasvm*, that can be selectively invoked for arbitrary segments of code, allowing us to mix emulated and non-emulated execution inside the same code execution. 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 in response to a detected failure, as was done in [26].

Upon entering the vulnerable section of code, the emulator snapshots the program state and executes all instructions on a 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 program. 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 about to emulate, also using additional information that is supplied by the sensor that detected the fault. In the case of division by zero, the emulator need only check the value of the operand to the *div* instruction. For illegal memory dereferencing, the emulator verifies whether the source and destination address of any memory access (or the program counter, for instruction fetches) points 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 [7] 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 to recover the execution.

We currently assume that the emulator is pre-linked with the vulnerable application, or that the source code of that application is available. However, it is possible to circumvent this limitation by using the processor's programmable breakpoint register (in much the same way as 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.

2.4 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. We analyze the declared type of the function using a TXL [18] script to determine the appropriate error return value.

TXL is a hybrid function 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 that drives parsing of the source input is specified in a notation similar to Extended Backus-Naur (BNF). In our system, we use TXL for C-to-C transformations using the GCC C front-end.

Depending on the return type of the emulated function, the system returns an "appropriate" value. This value is determined based on some straightforward heuristics. For example, if the return type is an *int,* a -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 later dereferenced further 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; the error return semantics of the Linux kernel are an example of such a system.

As a first approach, these heuristics worked extremely well in our experiments (see Section 4). 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 for a large application, it is possible to ask the programmer to provide a short description of this convention as input to our system.

2.5 Caveats and Limitations

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.

A reaction system must evaluate and choose a response from a wide array of choices. Furthermore, much research remains to be done in discovering the actual limits and boundaries of this broad spectrum. Currently, when encountering a fault, a system can (a) crash, (b) crash and be restarted by a monitor [5], (c) return bogus values [23], or (d) slice off the functionality. Most proactive systems take the first approach. We elect to take the last approach. As noted in Section 2.4, this choice appears to work extremely well. Such an approach also seems to work at the machine instruction level, as noted in [28].

However, there is a fundamental problem in choosing a particular response. Since the high-level behavior of any system cannot be algorithmically determined, a response 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 the *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.

There is a key tradeoff between code coverage (and thus confidence in the level of security the system provides) and performance (the amount of memory and time that the emulator adds). 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 this fault. In a related problem, I/O beyond the machine presents a problem for a roll back strategy. This problem can partially be addressed by the approach taken in [15] 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 circumvent this limitation in Section 2.3.

3 Implementation

We implemented the *libtasvm x86* emulator to validate the practicality of providing a supervision framework for the feedback control loop through selective emulation of code slices. Integrating *libtasvm* into an existing application is straightforward. As shown in Figure 2, four special tags are wrapped around the segment of code that will be emulated.

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

Figure 2: **A trivial example of using *libtasvm*. The emulate_* calls invoke and terminate execution of *libtasvm*. The code inside that region is executed by the emulator. For simplicity, we show only the increment statement as being executed by the emulator, but there is no reason why the entire function body couldn't be emulated (including calls to other functions).**

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 *libtasvm* 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 arises during emulation, *libtasvm* 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; even memory updates are backed out. 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 *libtasvm*. Thus, the emulator backs up the real registers and replaces them with its own values. An INT $0x80$ is issued by *libtasvm*, 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.

The next step was to confirm the effectiveness and performance impact of the emulator.

4 Evaluation

In this section, we qualitatively validate our approach using a set of exploits against popular server applications, and quantitatively measure the performance impact of selective emulation.

4.1 Effectiveness of Forced Return Recovery

In order to validate our hypothesis on control flow recovery using forced function return, introduced in Section 2.4, we experimentally evaluate its effects on program execution on the Apache httpd, OpenSSH sshd, and Bind. We run profiled versions of the selected applications against a set of test suites and examine the subsequent call-graphs generated by these tests with *gprof* and Valgrind [20].

The ensuing call trees are analyzed in order to extract leaf functions. The leaf functions are, in turn, employed as potentially vulnerable functions. Armed with the information provided by the call-graphs, we run a TXL script that inserts an early return in all the leaf functions (as described in Section 2.4), simulating an aborted function. Specifically, we examined 154 leaf functions. For each aborted function, we monitor the program execution of Apache by running httperf [19], a web server performance measurement tool. Success for each test was defined as the application not crashing.

The results from these tests were very encouraging; 139 of the 154 functions completed the httperf tests successfully: program execution was not interrupted. What we found to be surprising was that not only did the program not crash, but in some cases *all* the pages were served correctly. This is probably due to the fact a large number of the functions are used for statistical and logging purposes. Furthermore, out of the 15 functions that produced segmentation faults, 4 did so at start up (and would thus not be relevant in the case of a long-running process).

Similarly for sshd, we iterate through each aborted function while examining program execution during an scp transfer. In the case of sshd, we examined 81 leaf functions. Again, the results were auspicious: 72 of the 81 functions maintained program execution. Furthermore, only 4 functions caused segmentation faults; the rest simply did not allow the program to start.

For Bind, we examined the program execution of *named* during the execution of a set of queries; 67 leaf functions were tested. In this case, 59 of the 67 functions maintained the proper execution state. Similar to sshd, only 4 functions caused segmentation faults.

4.2 Exploits

Given the success of our experimental evaluation on program execution, we wanted to further validate our hypothesis against a set of real exploits for Apache, OpenSSH and Bind. No prior knowledge was encoded in our system with respect to the vulnerabilities: for all purposes, this experiment was a zero-day attack.

For Apache, we used the apache-scalp exploit that takes advantage of a buffer overflow vulnerability based on the incorrect calculation of the required buffer sizes for chunked encoding requests. We applied selective emulation on the offending function and successfully recovered from the attack; the server successfully served subsequent requests.

The attack used for OpenSSH was the RSAREF2 exploit for SSH-1.2.27. This exploit relies on unchecked offsets that result in a buffer overflow vulnerability. Again, we were able to gracefully recover from the attack and the sshd server continued normal operation.

Bind is susceptible to a number of known exploits; for the purposes of this experiment, we tested our approach against the TSIG bug on ISC Bind 8.2.2-x. In the same motif as the previous attacks, this exploit takes advantage of a buffer overflow vulnerability. As before, we were able to safely recover program execution while maintaining service availability.

4.3 Performance

We next turned our attention to the performance impact of our system. In particular, we measured the overhead imposed by the emulator component. The *libtasvm* emulator 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 2.2.

We evaluated the performance impact of *libtasvm* 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*.

4.3.1 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). *libtasvm* was compiled with the "*-g -static -fno-defer-pop*" flags.

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 **??** and 4. 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.

4.3.2 Emulation of Apache Inside Valgrind

To get a sense of the performance degradation imposed by running the entire system inside an emulator other than *libtasvm*, we tested Apache running in Valgrind version 2.0.0 on the Linux test machine that hosted Apache for our *libtasvm* 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 *libtasvm* 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 3 and the timing performance in Table 1.

Note that the Valgrind–ized version of Apache is 10 times the size of the regular Apache image, while Apache with libtasvm is not noticably larger.

4.3.3 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 213000 instructions. The impact on performance is clearly seen in Figure 3 and further elucidated in Figure 4, which plots the performance of the fully emulated request-handling procedure.

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.

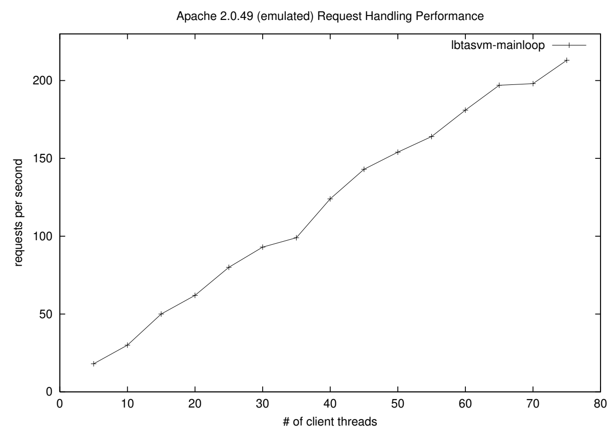

Figure 4: **A closer look at the performance for the fully emulated version of main processing loop. While there is a considerable performance impact compared to the non-emulated request handling loop, the emulator appears to scale at the characteristic linear rate, indicating that it does not create additional overhead beyond the cost of emulation.**

For our test machines and sample loads, Apache normally (*e.g.,* non-emulated) spent some 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Table 1. 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) 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 settled down.

4.3.4 Selective Emulation

Lacking any actual attacks to launch against Apache (with the exception of the apache-scalp exploit, as we previously discussed), we used the RATS tool to identify possible vulnerable sections of code in Apache 2.0.49. The tool identified roughly 270 can-

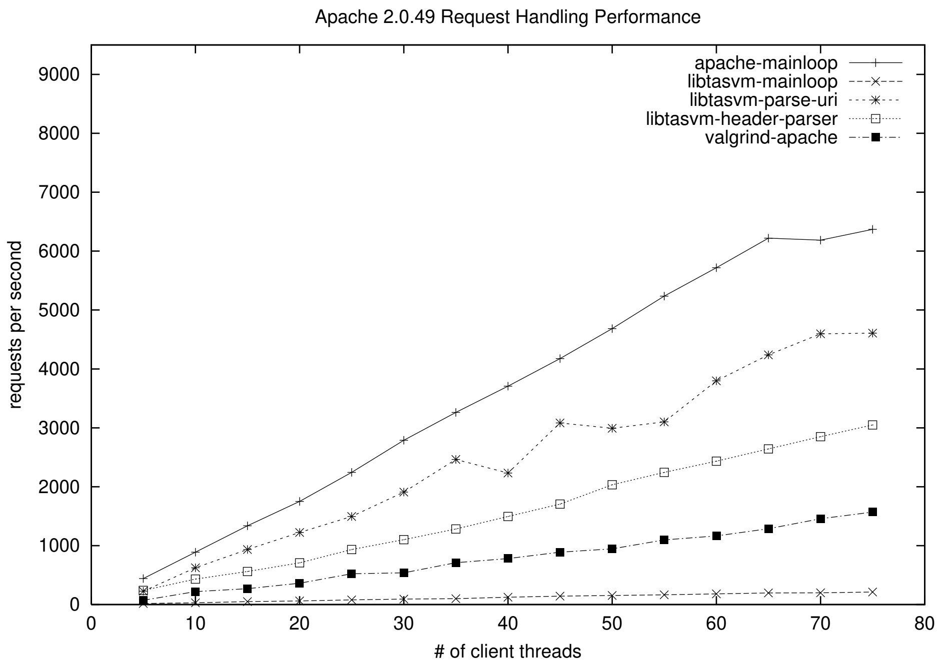

Figure 3: **Performance of the system under various levels of emulation. This data set includes Valgrind for reference. While full emulation is fairly invasive, selective emulation of input handling routines appears quite sustainable. Valgrind runs better than *libtasvm* when executing the entire request loop. As expected, selective emulation still performs better than Valgrind.**

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

Table 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.**

didate 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.

4.3.5 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 libtasvm. We selected some common shell utilities and measured their performance for large workloads running both with and without libtasvm.

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 2 shows the result of these measurements.

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.

5 Related Work

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (specifically, Java) in [25, 24]. 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 that 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 emulating, 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.,* once an attack has been observed, possibly more than once) does not succumb to attacks.

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research. Virtual machine monitors (VMM) are employed in a number of security–related contexts, from autonomic patching of vulnerabilities [26] to intrusion detection [12].

Virtual machine monitors (VMMs) are one type of protection mechanism; other approaches include compiler techniques like StackGuard [8] and safer libraries, such as *libsafe* and *libverify* [2]. Other tools exist to verify and supervise code during development or debugging. Of these tools, Purify[3] and Valgrind [20] are popular choices.

Valgrind is a program supervision framework that enables in–depth instrumentation and analysis of IA-32 binaries without recompilation. Valgrind has been used by Barrantes et al. [3] to implement instruction set randomization techniques to protect programs against code insertion attacks. Other work on instruction–set randomization includes [14], which employs the i386 emulator Bochs[4].

Program shepherding [17] is a technique developed by Kiriansky, Bruening, and Amarasinghe. The authors describe a system based on the RIO [10] architecture for protecting and validating control flows according to some security policy without modification of IA-32 binaries for Linux and Windows. The system works by validating branch instructions and storing the decision in a cache, thus incurring little overhead.

The work by Dunlap, King, Cinar, Basrai, and Chen [11] is closely related to the work presented in this paper. ReVirt is a system implemented in a VMM that logs detailed execution information. This detailed execution trace includes non–deterministic events such as timer interrupt information and user input. Because ReVirt is implemented in a VMM,

[3] `http://www.rational.com`

[4] `http://bochs.sourceforge.net/`

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 2: **Microbenchmark performance times for various command line utilities.**

it is more resistant to attack or subversion. However, ReVirt's primary use is as a forensic tool to replay the events of an attack, while the goal of *libtasvm* is to provide a lightweight and minimally intrusive mechanism for protecting code against malicious input *at runtime*.

King, Dunlap, and Chen [16] discuss optimizations that reduce the performance penalties involved in using VMMs. There are three basic optimizations: reduce the number of context switches by moving the VMM into the kernel, reduce the number of page faults by allowing each VMM process greater freedom in allocating and maintaining address space, and ameliorate the penalty for switching between guest kernel mode and guest user mode by simply changing the bounds on the guest memory area rather than re–mapping.

An interesting application of ReVirt [11] is BackTracker, [15], a tool that can the automatically identify the steps involved in an intrusion. Because detailed execution information is logged, a dependency graph can be constructed backward from the detection point to provide forensic information about an attack.

Toth and Kruegel [27] propose to detect buffer overflow payloads (including previously unseen ones) by treating inputs received over the network as code fragments; they show that legitimate requests will appear to contain relatively short sequences of valid *x86* instruction opcodes, compared to attacks that will contain long sequences. They integrate this mechanism into the Apache web server, resulting in a small performance degradation.

Some interesting work has been done to deal with memory errors at runtime. For example, Rinard et al. [22] have developed a compiler that inserts code to deal with writes to unallocated memory by automatically 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 [22] is modified in [23] and introduced as *failure-oblivious computing*. This behavior of this technique is close to the behavior of our system.

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 presented by Dempsky [9], 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.

While our prototype *x86* emulator is a fairly straightforward implementation, it can gain further performance benefits by using Valgrind's technique of caching already translated instructions. With some further optimizations, libtasvm is a viable and practical approach to protecting code. In fact, [4] outlines several ways to optimize emulators; their approaches reduce the performance overhead (as measured by two SPEC2000 benchmarks, *crafty* and *vpr*) from a factor of 300 to about 1.7. Their optimizations include caching basic blocks (essentially what VG is doing), linking direct and indirect branches, and building traces.

6 Conclusions

Software errors and the concomitant potential for exploitable vulnerabilities remain a pervasive problem. Accepted approaches to this problem are almost always proactive, but it seems unlikely that such strategies will result in error-free code. In the absence of such guarentees, reactive techniques for error toleration and recovery are powerful tools.

We have described a lightweight mechanism for supervising the execution of an application that has already exhibited a fault and preventing its recurrence. We use selective emulation of the code immediately surrounding the fault to validate the operands to machine instructions, as appropriate for the type of fault; we currently handle buffer overflows, illegal memory dereferences, and division by zero exceptions. Once a fault has been detected, we restore control to a safe flow by forcing the function containing the fault to return an error value, also rolling back any memory modifications the emulated code has made during its execution.

Our intuition is that most applications are written well enough to catch the majority of errors, but fail to consider some boundary conditions that allow the fault to manifest itself. By catching these extreme cases and returning an error, we make use of the already existing error-handling code. We validate this hypothesis using a set of real attacks, as well as randomly induced faults in a number of open-source servers (Apache, sshd, Bind). Our results show that our system works in over 88% of all cases, allowing the application to continue execution and behave correctly. Furthermore, by using selective emulation of small code segments, we minimize the performance impact on production servers.

Our approach allows quick, *automated* reaction to software failures, thereby increasing service availability in the presence of general software bugs. We re-emphasize that our approach can be used to catch a variety of software failures, not just malicious attacks. Our plans for future work include enhancing the performance of our prototype emulator and further validating our hypothesis by extending the number of applications and attacks examined.

References

[1] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

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

[6] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[7] 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 7th USENIX Security Symposium*, January 1998.

[8] 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. 1998.

[9] B. Demsky and M. 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 Applications*, October 2003.

[10] E. Duesterwald and S. P. Amarsinghe. On the Run – Building Dynamic Program Modifiers for Optimization, Introspection, and Security. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

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

[12] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *10th ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[13] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[15] S. T. King and P. M. Chen. Backtracking Intrusions. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[16] S. T. King, G. Dunlap, and P. Chen. Operating System Support for Virtual Machines. In *Proceedings of the General Track: USENIX Annual Technical Conference*, June 2003.

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

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

[19] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[20] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

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

[22] 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 20th Annual Computer Security Applications Conference (ACSAC) 2004*, December 2004.

[23] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

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

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

[26] 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.

[27] 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.

[28] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[29] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of ACM SOSP*, October 2003.