Hardware Support For Self-Healing Software Services

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

Abstract

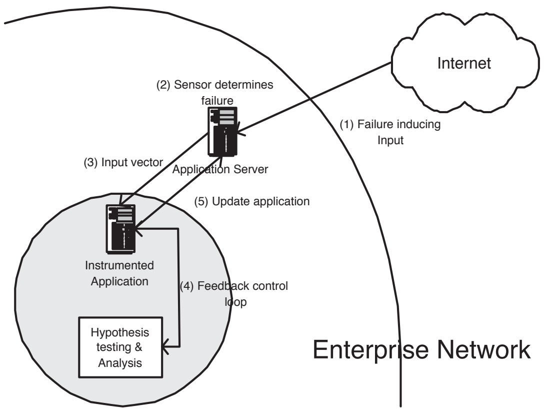
We propose a new approach for *reacting* to a wide variety of software failures, ranging from remotely exploitable vulnerabilities to more mundane bugs that cause abnormal program termination (*e.g.,* illegal memory dereference). Our emphasis is in creating "self-healing" software that can protect itself 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. This emulator allows us to predict recurrences of faults and recover 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 describe the overall system architecture, highlighting a prototype implementation for the *x86* platform. We discuss a *virtual emulator,* which uses existing and new processor features to improve performance.

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 deliberate remote attacks, high-volume events (such as fast-spreading worms that may trigger unrelated and possibly non-exploitable bugs, or simple denial of service attacks directed against the software as opposed to the underlying network). The majority of solutions to this problem are proactive and focus on making the code as dependable as possible through safe languages and compilers, code analysis, development methodologies, sandboxing, Byzantine fault-tolerance schemes, and other approaches.

We take a *reactive* approach by observing an application (or appropriately instrumented instances of it) for previously unseen failures. 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. The types of faults we examine here consist of illegal memory dereferences and division by zero exceptions, as well as 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).

To recover, 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 allows us to (a) monitor for the specific type of failure prior to executing the instruction, (b) undo any memory changes made by the code function inside which the fault occurred, by having the emulator record all memory modifications made during its execution, and (c) simulate an error-return from said function. Our intuition is that most applications can handle most errors (*i.e.,* a routine returning with an error code), even though a condition that was not predicted by the programmer allowed a fault to slip in. Our preliminary experiments with Apache, OpenSSH, and Bind support this. Although the system incurs a significant performance penalty, we believe that by using certain features at the micro-architecture level, it should be possible to create a *virtual emulator* that exhibits better performance.



Internet
(2) Sensor determines failure
(1) Failure inducing Input
(3) Input vector
Application Server
(5) Update application
Instrumented Application
(4) Feedback control loop
Hypothesis testing & Analysis
Enterprise Network

Figure 1: **Self-healing services architecture.**

2 Architecture

Our architecture, conceptually 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. The types of faults we are currently handling include those that cause abnormal program termination. Specifically, we handle illegal memory accesses, division by zero, and buffer overflow attacks (as an instance of a remotely exploitable software fault that is widely in use). We intend to enrich this set of faults in the future, specifically examining Time-Of-Check-To-Time-Of-Use (TOCTTOU) violations.

Upon detecting a fault in the monitored application, the sensor instruments the portion of the application's code that immediately surrounds the faulty instruction(s) such that the code segment is emulated. To verify the effectiveness of the fix, the application is restarted in a test environment (sandbox) with the instrumentation enabled, and is supplied with the input that caused the failure[1]. During emulation, *libtasvm* maintains a record of all memory changes (including global variables or library-internal state, *e.g., libc* standard I/O structures) 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 and slicing, as we only need to focus on the operation and side effects of individual instructions independently from each other. If the emulator determines that the fault is about to occur after (or while) the next instruction, 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 this is done later in this paper. The intuition behind our approach is that most code is designed to handle error returns from functions, despite the fact that boundary conditions that escaped the programmer's attention allowed a particular fault to manifest itself. We describe this hypothesis in more detail later in this section.

Upon forcing the function to return, emulation also terminates. If the program 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, as we shall see in Section 3, this is very 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

[1]or the N most recent inputs, if the offending one cannot be easily identified

not triggered during an emulated execution, emulation ends at the end of the vulnerable code segment making all memory changes 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, although the emulator allows us to start and stop emulation at arbitrary points.

Application Monitors The sensors we need to detect software failures depend on the nature of the flaws themselves, as well as on our tolerance of their impact to system performance. We describe two types of application monitors that we experimented with. The first depends on the behavior of the operating system when dealing with application faults such as illegal memory dereferences. The application is forced to abort, and the OS creates a core dump that includes the type of failure and the stack trace when that failure occurred. This dump provides us with sufficient information to apply selective emulation, starting with the top-most function in the stack trace. Thus, we only need a watchdog process that waits until the service terminates before it invokes our system.

A second approach is to use an instrumented version of the application on a separate server as a honeypot, as we demonstrated for the case of network worms [2]. 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 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). 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.

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. To enable selective emulation, we provide a statically-linked C library that defines special tags (combination of macros and function calls) that mark the beginning and end of selective emulation. Upon entering the vulnerable section of code, the emulator captures the program state and processes all instructions, including function calls, inside the area designated for 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 is done in [2]. 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. The use of an emulator allows us to monitor a wide array of software failures such as illegal memory dereferences, buffer overflows and underflows, and more generic faults such as division by zero. To implement this, 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 [1] operates. The emulator then simply watches for memory writes to these memory locations. Our approach allows us to stop the attack before it overwrites the stack, and to recover the execution.

Once we detect a software failure, we are able to undo any memory changes made by the code function inside which the fault occurred by having the emulator keep track of all implicit memory modifications during its execution. Currently, the emulator must be 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.

Recovery: Forcing Error Returns Upon detecting a fault, our recovery mechanism undoes all memory changes and forces an error return from the currently executing function. To determine the appropriate error value, we analyze the declared type of the function. Depending on the return type of the emulated function, an "appropriate" value is returned based on some straightforward heuristics. For example, if the return type is an *int,* a -1 is returned; if the value is *unsigned int* we return 0, *etc*. A special case is used when the function returns a pointer. Specifically, 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. These heuristics worked extremely well in our preliminmary experiments. 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.

3 Evaluation

To validate our hypothesis on control flow recovery using forced function return, we ran 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. 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 script that inserts an early return in all the leaf functions, simulating an aborted function. Specifically, we examined 154 leaf functions. For each simulated aborted transaction, we monitor the program execution of Apache by running *httperf*, a web server performance measurement tool. 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.

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.

To further validate our hypothesis against real attacks, we used set of exploits for Apache, OpenSSH, and Bind and tested them against our system. No prior knowledge was encoded in our system with respect to the vulnerabilities: for all purposes, this was a zero-day attack. For Apache, we used the apache-scalp exploit that takes advantage of the fact that Apache improperly calculates the required buffer sizes for chunked encoding requests resulting in a buffer overflow. 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 attack. 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. Similar to previous cases, we were able to safely recover program execution while maintaining service availability.

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 previously discussed. We evaluated the performance impact of *libtasvm* by instrumenting the Apache 2.0.49 web server and OpenSSH *sshd*. We first show that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the main request processing loop for Apache (contained in *ap_process_http_connection()*) and compared our results against a non-emulated Apache instance. In this experiment, the emulator executed roughly 213,000 instructions. The impact on performance is clearly seen in Figure 2 and further elucidated in Figure 3, which plots the performance of the fully emulated request-handling procedure.

To get a more complete sense of this performance impact, we timed the execution of the request handling procedure for both the non-emulated and fully-emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked. For our test machines and sample loads, Apache normally (*e.g.,* non-emulated) spent some 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Figure 4. The fully instrumented loop running in the emulator spends an average of 278 milliseconds per request in that particular code section.

Lacking any actual attacks to launch against Apache (with the exception of the apache-scalp

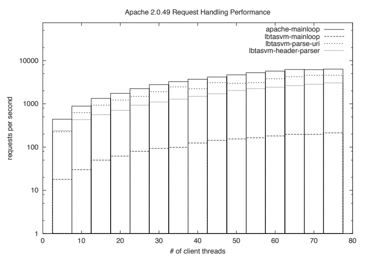


Figure 2: **Performance of various levels of emulation. While full emulation is fairly invasive in terms of performance, selective emulation of input handling routines appears quite sustainable.**

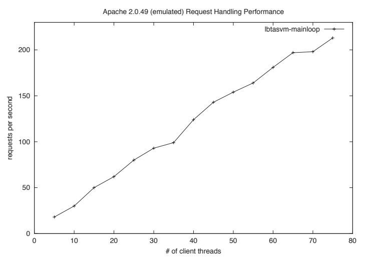


Figure 3: **A closer look at the performance for the fully emulated version of main processing loop. While there is a considerable performance impact compared to the non-emulated request handling loop, the emulator appears to scale at the characteristic linear rate, indicating that it does not create additional overhead beyond the cost of emulation.**

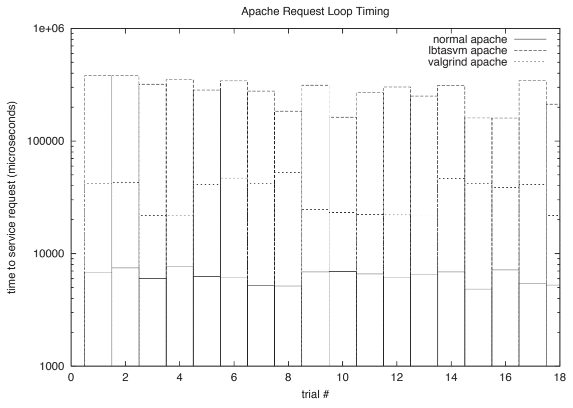


Figure 4: **Timing of main request processing loop. The y axis is on a log scale and shows the overhead of running the whole primary request handling mechanism inside the emulator. Each trial represents a user thread issuing an HTTP GET request.**

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 candidate lines of code, the majority of which contained fixed size local buffers. We then correlated the entries on the list with code that was in the primary execution path of the request processing loop. The 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.

However, it should be possible to do better than that, by using a *virtual emulator* that uses the standard memory-management subsystem of the processor and operating system, and a new proposed feature we call *instruction filtering*. Effectively, we suggest a debugging enhancement to processors that causes the CPU to throw an exception when a particular instruction (or class of instructions) is about to be executed; control is then transferred to a registered handler. This is similar to the way instruction rollback after a page fault is implemented in some architectures, and also reminiscent of the way virtual machines like VMWare handle non-virtualizable instructions. Thus, we can allow a piece of code to execute until a specific class of instructions is encountered, at which point the checking code currently used in *libtasvm* can be invoked to determine whether bad behavior is about to be exhibited. Furthermore, we can use the copy-on-write feature available in most operating systems: prior to executing the vulnerable code, mark all pages as read-only; any memory writes will cause an exception that the operating system will catch and recognize as relevant to the emulation. The OS will then create a copy of the page, and let the application resume execution. Once we leave emulation, we can decide which version of the pages to keep. Using these two features (one existing, one proposed), it should be possible to achieve better performance than pure emulation.

References

[1] C. C. et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the* 7^{th} *USENIX Security Symposium*, January 1998.

[2] 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.