Enhancing Server Availability and Security Through Failure-Oblivious Computing

Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy,
Tudor Leu, and William S. Beebee, Jr.
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We present a new technique, *failure-oblivious computing*, that enables servers to execute through memory errors without memory corruption. Our safe compiler for C inserts checks that dynamically detect invalid memory accesses. Instead of terminating or throwing an exception, the generated code simply discards invalid writes and manufactures values to return for invalid reads, enabling the server to continue its normal execution path.

We have applied failure-oblivious computing to a set of widely-used servers from the Linux-based open-source computing environment. Our results show that our techniques 1) make these servers invulnerable to known security attacks that exploit memory errors, and 2) enable the servers to continue to operate successfully to service legitimate requests and satisfy the needs of their users even after attacks trigger their memory errors.

We observed several reasons for this successful continued execution. When the memory errors occur in irrelevant computations, failure-oblivious computing enables the server to execute through the memory errors to continue on to execute the relevant computation. Even when the memory errors occur in relevant computations, failure-oblivious computing converts requests that trigger unanticipated and dangerous execution paths into anticipated invalid inputs, which the error-handling logic in the server rejects. Because servers tend to have small error propagation distances (localized errors in the computation for one request tend to have little or no effect on the computations for subsequent requests), redirecting reads that would otherwise cause addressing errors and discarding writes that would otherwise corrupt critical data structures (such as the call stack) localizes the effect of the memory errors, prevents addressing exceptions from terminating the computation, and enables the server to continue on to successfully process subsequent requests. The overall result is a substantial extension of the range of requests that the server can successfully process.

1 Introduction

Memory errors such as out of bounds array accesses and invalid pointer accesses are a common source of program failures. Safe languages such as ML and Java use dynamic checks to eliminate such errors — if, for example, the program attempts to access an out of bounds array element, the implementation intercepts the attempt and throws an exception. The rationale is that an invalid memory access indicates an unanticipated programming error and it is unsafe to continue the execution without first taking some action to recover from the error.

Recently, several research groups have developed compilers that augment programs written in unsafe languages such as C with dynamic checks that intercept out of bounds array accesses and accesses via invalid pointers (we call such a compiler a *safe-C* compiler) [17, 58, 45, 36, 50, 37]. These checks use additional information about the layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a check, it terminates after printing an error message.

1.1 Failure-Oblivious Computing

Note that it is possible for the compiler to automatically transform the program so that, instead of throwing an exception or terminating, it simply ignores any memory errors and continues to execute normally. Specifically, if the program attempts to read an out of bounds array element or use an invalid pointer to read a memory location, the implementation can simply (via any number of mechanisms) manufacture a value to supply to the program as the result of the read, and the program can continue to execute with that value. Similarly, if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, the implementation can simply discard the value and continue. We call a computation that uses this strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

It is not immediately clear what will happen when a program uses this strategy to execute through a memory error. When we started this project, our hypothesis was that, for at least some programs, this continued execution would produce acceptable results. To test this hypothesis, we implemented a C compiler that generates failure-oblivious code, obtained some C programs with known memory errors, and observed the execution of failure-oblivious versions of these programs. Here is a summary of our observations:

- **Acceptable Continued Execution:** We targeted memory errors in servers that correspond to security vulnerabilities as documented at vulnerability tracking web sites [13, 12]. For all of our tested servers, failure-oblivious computing 1) eliminates the security vulnerability and 2) enables the server to successfully execute through the error to continue to serve the needs of its users.

- **Acceptable Performance:** Failure-oblivious computing entails the insertion of dynamic bounds checks into the compiled program. Previous experiments with safe-C compilers have indicated that these checks usually cause the program to run less than a factor of two slower than the version without checks, but that in some cases the program may run as much as eight to twelve times slower [58, 50]. Our results are consistent with these previous results. Note that many of our servers implement interactive computations for which the appropriate performance measure is the observed pause times for processing interactive requests. For all of our interactive servers, the application of failure-oblivious computing does not perceptibly increase the pause times.

Our conclusion is that continued execution through memory errors produces completely acceptable results for all of our servers *as long as failure-oblivious computing prevents these errors from corrupting the server's address space or data structures*.

1.2 Reason for Successful Execution

Memory errors can damage a computation in several ways: 1) they can cause the computation to terminate with an addressing exception, 2) they can cause the computation to become stuck in an infinite loop, 3) they can change the flow of control to cause the computation to generate a new and unacceptable interaction sequence (either with the user or with I/O devices), 4) they can corrupt data structures that must be consistent for the remainder of the computation to execute acceptably, or 5) they can cause the computation to produce unacceptable results.

Because failure-oblivious computing intercepts all invalid memory accesses, it eliminates the possibility that the computation may terminate with an addressing exception. It is still possible for the computation to infinite loop, but we have found a sequence of return values for invalid reads that, in practice, appears to eliminate this problem for our server programs. Our servers have simple interaction sequences — read a request, process the request without further interaction, then return the response. As long as the computation that processes the request terminates, control will appropriately flow back to the code that reads the next request and there will be no unacceptable interaction sequences. Discarding invalid writes tends to localize any memory corruption effects. In particular, it prevents an access to one data unit (such as a buffer, array, or allocated memory block) from corrupting another data unit. In practice, this localization protects many critical data structures (such as widely used application data structures or the call stack) that must remain consistent for the program to execute acceptably.

The remaining issue is the potential production of unacceptable results. Manufacturing values for reads clearly has the potential to cause a subcomputation to produce an incorrect or unexpected result. The key question is how (or even if) the incorrect or unexpected result may propagate through the remaining computation to affect the overall results of the program.

All of our initially targeted memory errors eventually boil down to buffer-overrun problems: as it processes a request, the server allocates a fixed-size buffer, then (under certain circumstances) fails to check that the data actually fits into this buffer. An attacker can exploit this error by submitting a request that causes the server to write beyond the bounds of the buffer to overwrite the contents of the stack or heap, typically with injected code that the server then executes. Such attacks are currently the most common source of exploited security vulnerabilities in modern networked computer systems [2]. Estimates place the total cost of such attacks in the billions of dollars annually [3].

Failure-oblivious computing makes a server invulnerable to this kind of attack — the server simply discards the out of bounds writes, preserving the consistency of the call stack and other critical data structures. For two of our servers the memory errors occur in computations and buffers that are irrelevant to the overall results that the server produces for that request. Because failure oblivious computing eliminates any addressing exceptions that would otherwise terminate the computation, the server executes through the irrelevant computation and proceeds on to process the request (and subsequent requests) successfully. For the other servers (in these servers the memory errors occur in relevant computations and buffers) , failure-oblivious computing converts the attack request (which would otherwise trigger a dangerous, unanticipated execution path) into an anticipated invalid input which the server's standard error-handling logic rejects. The server then proceeds on to read and process subsequent requests acceptably.

One of the reasons that failure-oblivious computing works well for our servers is that they have short error propagation distances — an error in the computation for one request tends to have little or no effect on the computation for subsequent requests. By discarding invalid writes, failure-oblivious computing isolates the effect of any memory errors to data local to the computation for the request that triggered the errors. The result is that the server has short data error propagation distances — the errors do not propagate to data structures required to process subsequent requests. The servers also have short control flow error propagation distances: by preventing addressing exceptions from terminating the computation, failure-oblivious computing enables the server to return to a control flow path that leads it back to read and process the next request. Together, these short data and control flow propagation distances ensure that any effects of the memory error quickly work their way out of the computation, leaving the server ready to successfully process subsequent requests.

1.3 Scope

Our expectation is that failure-oblivious computing will work best with computations, such as servers, that have short error propagation distances. Failure-oblivious computing enables these programs to survive otherwise fatal errors or attacks and to continue on to execute and interact acceptably. Failure-oblivious computing should also be appropriate for multipurpose systems with many components — it can prevent an error in one component from corrupting data in other components and keep the system as a whole operating so that other components can continue to successfully fulfill their purpose in the computation.

Until we develop technology that allows us to track results derived from computations with memory errors, we anticipate that failure-oblivious computing will be less appropriate for programs (such as many numerical computing programs) in which a single error can propagate through to affect much of the computation. We also anticipate that it will be less appropriate for programs in which it is acceptable and convenient to terminate the computation and await external intervention. This situation occurs, for example, during development — the program is typically not producing any useful results and developers with the ability and motivation to find and eliminate any errors are readily available. We therefore see failure-oblivious computing as useful primarily for deployed programs whose users 1) need the results that the program produces and 2) are unable or unwilling to tolerate failures or to find and fix errors in the program.

1.4 Advantages and Drawbacks

The primary characteristic of failure-oblivious computing as compared with previous approaches is continued execution combined with the elimination of data structure corruption caused by memory errors. The potential benefits include:

- **Availability:** The combination of protection against data structure corruption and continued execution in the face of memory errors can significantly increase the availability of the server. This combination enables the server to continue to provide service to legitimate users even in the face of repeated attacks (or, for that matter, other infrequently-triggered fatal memory errors).
- **Security:** Failure-oblivious computing eliminates the possibility that an attacker can exploit memory errors to corrupt the address space of the server. The result is a more secure system that is immune to buffer-overrun attacks.
- **Minimal Adoption Cost:** The net adoption cost to the developer is to recompile the server using a compiler that generates failure-oblivious code. There is no need to change programming languages, write exception handling code, or modify the software in any way. Failure-oblivious computing can therefore be applied immediately to today's software infrastructure.
- **Reduced Administration Overhead:** One of the most challenging system administration tasks is ensuring that servers are kept up to date with a constant stream of (potentially disruptive) patches and upgrades; this stream is driven, in large part, by the need to eliminate memory-error based security vulnerabilities in otherwise perfectly acceptable servers. Because failure-oblivious computing eliminates this class of errors, it may enable system administrators to safely ignore patches whose purpose is to eliminate security vulnerabilities caused by memory errors. Ideally, administrators would become able to patch their systems primarily to obtain new functionality, not because they need to close security vulnerabilities in programs that are otherwise fully serving the needs of their users.

There are also several potential drawbacks:

- **Unanticipated Execution Paths:** Failure-oblivious computing has the potential to take the program down an execution path that was unanticipated by the programmer, with the prospect of this path producing unacceptable results.[1] This possibility can be especially problematic if errors in the unanticipated path have long propagation distances through the relevant data or when control fails to flow back to an appropriate point in the program. This drawback is, in our view, an unavoidable consequence of *any* mechanism that is intended to increase the resilience of programs in the face of errors — errors occur precisely because the program encountered a situation that the programmer either did not anticipate or did not deem worth handling correctly.

- **The Bystander Effect:** A more abstract issue is the potential for failure-oblivious computing to trigger the *bystander effect* in developers. In a variety of settings that range from manufacturing [25] to personal relationships [40, 24], the mere presence of mechanisms that may detect and compensate for errors has the effect of reducing the effectiveness of the participants in the setting and, in the end, the overall quality of the system as a whole. A potential explanation is that the participants start to rely psychologically on the error recovery mechanisms, which reduces their motivation to eliminate errors in their own work. Deploying failure-oblivious computing into a software development setting may therefore reduce the quality of the software that the developers are able to deliver. One obvious way to combat the bystander effect in this setting is to ban the use of failure-oblivious computing during development. Once again, note that the possibility of triggering the bystander effect is not restricted to failure-oblivious computing — *any* error recovery mechanism has the potential to trigger this effect.

1.5 Contributions

This paper makes the following contributions:

- **Failure-Oblivious Computing:** It introduces the concept of failure-oblivious computing, in which the program discards illegal writes, manufactures values for illegal reads, and continues to execute through memory errors without address space or data structure corruption.

- **Experience:** It presents our experience using failure-oblivious computing to enhance the security and availability of a range of widely used open-source servers. Our results show that:

  – **Standard Compilation:** With the standard unsafe C compiler, the servers are vulnerable to memory errors and attacks that exploit these memory errors.

  – **Safe Compilation:** With a C compiler that generates code that exits with an error message when it detects a memory error, the servers exit when presented with an input that triggers a memory error (denying the user access to the services that the server is intended to provide).

  – **Failure-Oblivious Compilation:** With our C compiler that generates failure-oblivious code, all of our servers execute successfully through memory errors and attacks to continue to satisfy the needs of their users. Failure-oblivious computing improves both the availability and the security of the servers in our test suite.

- **Explanation:** By relating the properties of servers to the properties of failure-oblivious computing, we explain why failure-oblivious computing may work well for this general class of programs.

2 Example

We next present a simple example that illustrates how failure-oblivious computing operates. Figure 1 presents a (somewhat simplified) version of a procedure from the Mutt mail client discussed in Section 4.6. This procedure takes as input a string encoded in the UTF-8 format and returns as output the same string encoded in modified UTF-7 format. This conversion may increase the size of the string; the problem is that the procedure fails to allocate sufficient space in the return string for the worst-case size increase. Specifically, the procedure assumes a worst-case increase ratio of 2; the actual worst-case ratio is 7/3. When passed (the very rare) inputs with large increase ratios, the procedure attempts to write beyond the end of its output array.

With standard compilers, these writes succeed, corrupt the address space, and the program terminates with a segmentation violation. With safe-C compilers, Mutt exits with a memory error and does not even start the user interface. With our compiler, which generates failure-oblivious code, the program discards all writes beyond the end of the array and the procedure returns with an incompletely translated (truncated) version of the string. Mutt then uses the return value to tell the mail server

[1] We note in passing that this potential is already present in every program — the mere absence of memory errors provides no guarantee that the program is, in fact, operating acceptably.

```
static char *
utf8_to_utf7 (const char *u8, size_t u8len) {
    char *buf, *p;
    int ch, int n, i, b = 0, k = 0, base64 = 0;

    /* The following line allocates the return
       string.  The allocated string is too small;
       instead of u8len*2+1, a safe length would
       be u8len*4+1.
    */
    p = buf = safe_malloc (u8len * 2 + 1);

    while (u8len) {
        unsigned char c = *u8;
        if (c < 0x80) ch = c, n = 0;
        else if (c < 0xc2) goto bail;
        else if (c < 0xe0) ch = c & 0x1f, n = 1;
        else if (c < 0xf0) ch = c & 0x0f, n = 2;
        else if (c < 0xf8) ch = c & 0x07, n = 3;
        else if (c < 0xfc) ch = c & 0x03, n = 4;
        else if (c < 0xfe) ch = c & 0x01, n = 5;
        else goto bail;

        u8++, u8len--;
        if (n > u8len) goto bail;
        for (i = 0; i < n; i++) {
            if ((u8[i] & 0xc0) != 0x80) goto bail;
            ch = (ch << 6) | (u8[i] & 0x3f);
        }
        if (n>1 && !(ch >> (n*5+1))) goto bail;
        u8 += n, u8len -= n;

        if (ch < 0x20 || ch >= 0x7f) {
            if (!base64) {
                *p++ = '&';
                base64 = 1;
                b = 0;
                k = 10;
            }
            if (ch & ~0xffff) ch = 0xfffe;
            *p++ = B64Chars[b | ch >> k];
            k -= 6;
            for (; k >= 0; k -= 6)
                *p++ = B64Chars[(ch >> k) & 0x3f];
            b = (ch << (-k)) & 0x3f;
            k += 16;
        } else {
            if (base64) {
                if (k > 10) *p++ = B64Chars[b];
                *p++ = '-';
                base64 = 0;
            }
            *p++ = ch;
            if (ch == '&') *p++ = '-';
        }
    }

    if (base64) {
        if (k > 10) *p++ = B64Chars[b];
        *p++ = '-';
    }

    *p++ = '\0';
    safe_realloc ((void **) &buf, p - buf);
    return buf;

bail:
    safe_free ((void **) &buf);
    return 0;
}
```

Figure 1: String Encoding Conversion Procedure

which mail folder it wants to open. The mail server responds with an error code indicating that the folder does not exist. Mutt correctly handles this error and continues to execute, enabling the user to process email from other, legitimate, folders.

This example illustrates two key aspects of applying failure-oblivious computing:

- **Subtle Errors:** Real-world programs can contain subtle memory errors that can be very difficult to detect by either testing or code inspection, and these errors can have significant negative consequences for the program and its users.

- **Mostly Correct Programs:** Testing usually ensures that the program is mostly correct and works well except for exceptional operating conditions or inputs. Failure-oblivious computing can therefore be seen as a way to enable the program to proceed past such exceptional situations to return back within its normal operating envelope. And as this example illustrates, failure-oblivious computing can actually facilitate this return by converting unanticipated memory corruption errors into anticipated error cases that the program handles correctly.

3 Implementation

A failure-oblivious compiler generates two kinds of additional code: checking code and continuation code. The checking code detects memory errors and can be the same as in any memory-safe implementation. The continuation code executes when the checking code detects an attempt to perform an illegal access. This code is relatively simple: it discards erroneous writes and manufactures a sequence of values for erroneous reads.

Our implementation uses a checking scheme originally developed by Jones and Kelly [37] and then significantly enhanced by Ruwase and Lam [50]. This checking scheme maintains a table that maps locations to data units (each struct, array, and variable is a data unit) and uses this table to distinguish in bounds and out of bounds pointers.

Our implementation of the write continuation code simply discards the value. Our implementation of the read continuation code redirects the read to a preallocated buffer of values. In principle, any sequence of manufactured values should work. In practice, these values are sometimes used to determine loop conditions. Midnight Commander (see Section 4.5), for example, contains a loop that, for some inputs, searches past the end of a buffer looking for the "/" character. If the sequence of generated values does not include this character, the loop never terminates and Midnight Commander hangs. We therefore generate a sequence that iterates through all small integers, increasing the chance that, if the values are used to determine loop conditions, the computation will hit upon a value that will exit the loop (and avoid nontermination). Because zero and one are usually the most commonly loaded values in computer programs [59], the sequence is designed to return these values more frequently than other, less common, values.

One potential concern is that failure-oblivious computing may hide errors that would otherwise be detected and eliminated. To help make the errors more apparent, our compiler can optionally augment the generated code to produce a log containing information about the program's attempts to commit memory errors. This log may help administrators to detect and respond appropriately to the presence such errors. Note, however, that hiding errors is one of the primary goals of this research, and that any technique that makes programs more resilient in the face of errors will reduce the negative impact of the errors and therefore the incentive to find and eliminate them.

4 Experience

We implemented a compiler that generates failure-oblivious code, obtained several widely-used open-source servers with known memory errors, and evaluated the impact of failure-oblivious computing on their behavior. Many of these servers are key components of the Linux-based open-source interactive computing environment.

4.1 Methodology

We evaluate the behavior of three different versions of each server: the *Standard* version compiled with a standard C compiler (this version is vulnerable to any memory errors that the server may contain), the *Bounds Check* version compiled with the CRED safe-C compiler [50] (this version terminates the server with an error message at the first memory error), and the *Failure Oblivious* version compiled with our compiler. We evaluate three aspects of each server's behavior:

- **Security and Resilience:** We chose a workload that contains an input that triggers a known memory error in the server; this input typically exploits a security vulnerability as documented by vulnerability-tracking organizations such as Security Focus [13] and SecuriTeam [12]. We observe the behavior of the different versions on this workload; for the Failure Oblivious version we focus on the acceptability of the continued execution after the error.

- **Performance:** We chose a workload that both the Standard and Failure Oblivious versions can execute successfully. We use this workload to measure the *request processing time*, or the time required for each version to process representative requests. We obtain this time by instrumenting the server to record the time when it starts processing the request and the time when it stops processing the request, then subtracting the start time from the stop time.

- **Stability:** When possible, we deploy the Failure Oblivious version of each server into daily use as part of our normal computational environment. During this deployment we ensure that the workload contains attacks that trigger memory errors in each server. We focus on the long-term acceptability of the continued execution of the Failure Oblivious version of the deployed server.

We note that two of our servers (Pine and Midnight Commander) use out of bounds pointers in pointer inequality comparisons. While this is, strictly speaking, an error, the intention of the programmer is clear. To avoid having these errors cripple the Bounds Check versions of these servers, we (manually) rewrote the code containing the inequality comparisons to eliminate pointer comparisons involving out of bounds pointers.

We ran all the servers on a Dell workstation with two 2.8 GHz Pentium 4 processors, 2 GBytes of RAM, and running Red Hat 8.0 Linux.

4.2 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [11]. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks. We use Pine 4.44, which is distributed with Red Hat Linux version 8.0. This version of Pine has a memory error associated with a failure to correctly parse certain From fields [10].

4.2.1 The Memory Error

When Pine displays a list of messages, it processes the From field of each message to quote certain characters. This quoting is implemented by transferring the From field into a heap-allocated character buffer for display, inserting a \ character into the buffer before any quoted character. As part of the transfer, the length of the string can increase because of the additional \ characters. The procedure that calculates the maximum possible length of the character buffer fails to correctly account for the potential increase and produces a length that is too short for messages whose From fields contain many quoted characters.

4.2.2 Security and Resilience

The Standard version of Pine writes beyond the end of the buffer, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects the memory error and terminates the computation. With both of these versions, the user is unable to use Pine to read mail because Pine aborts or terminates during initialization as the mail file is loaded and before the user has a chance to interact with the server. The user must manually eliminate the From field from the mail file (using some other mail reader or file editor) before he or she can use Pine to read mail at all.

The Failure Oblivious version discards the out of bounds writes (in effect, truncating the translated From field) and continues to execute through the memory error, enabling the user to process their mail. Because the mail list user interface displays only an initial segment of long From fields, the truncation is not visible to the user. If the user selects the message, a different execution path correctly translates the From field. The displayed message contains the complete From field and the user can read, forward, and otherwise process the message.

4.2.3 Performance

Figure 2 presents the request processing times for the Standard and Failure Oblivious versions of Pine. All times are given in milliseconds. The Read request displays a selected empty message, the Compose request brings up the user interface to compose a message, and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	$0.287 \pm 7.1\%$	$1.98 \pm 1.5\%$	6.9
Compose	$0.385 \pm 4.3\%$	$3.11 \pm 2.6\%$	8.1
Move	$1.34 \pm 10.4\%$	$1.80 \pm 11.2\%$	1.34

Figure 2: Request Processing Times for Pine (milliseconds)

As these numbers indicate, the Failure Oblivious version is not substantially slower than the Standard version. Because Pine is an interactive program, its performance is acceptable as long as it feels responsive to its users. Assuming a pause perceptibility threshold of 100 milliseconds for this kind of interactive program [21], it is clear that failure-oblivious computing should not degrade the program's interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for all versions.

4.2.4 Stability

During our stability testing period, we used Pine as a default mail reader. Our activities included reading mail, replying to mails, forwarding mails, and managing mail folders. During this time we used Pine to process roughly 25 new mail messages a day (after spam filtering). To test Pine's ability to successfully execute through errors, we periodically sent an email that triggered the memory error discussed above in Section 4.2.1. We also used the failure-oblivious version of Pine to successfully process a large mail folder containing over 100,000 messages. During this usage period, the Failure Oblivious version executed successfully through all errors to perform all requests flawlessly.

4.3 Apache

The Apache HTTP server is the most widely used web server in the world: a recent survey found that 64% of the web sites on the Internet use Apache [9]. Apache version 2.0.47 contains a (under certain circumstances) remotely exploitable memory error [1].

4.3.1 The Memory Error

Apache can be configured to automatically redirect incoming URLs via a set of URL rewrite rules. Each rewrite rule contains a *match pattern* (a regular expression that may match an incoming URL) and a *replacement pattern*. The match pattern may contain parenthesized *captures*, each of which may match a substring from the incoming URL. The replacement pattern may reference these captures. When an incoming URL matches the match pattern, Apache replaces the URL with the replacement pattern after substituting out any referenced captures with the corresponding captured substrings from the incoming URL. As Apache processes the incoming URL, it uses a (stack-allocated) buffer to hold pairs of offsets that identify the captured substrings in the incoming URL. The buffer contains enough room for ten captures. If there are more, Apache writes the corresponding pairs of offsets beyond the end of the buffer.

4.3.2 Security and Resilience

The Standard version performs the out of bounds writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version correctly processes legitimate requests without memory errors until it is presented with a URL that triggers the memory error. At this point the child process serving the connection detects the error and terminates. Apache uses a pool of child processes to serve incoming requests. When one of the child processes terminates, the main Apache process creates a new child process to take its place. This mechanism allows both the Standard and Bounds Check versions of Apache to continue to service requests even when repeatedly presented with inputs that cause the child processes to terminate because of memory errors.

The Failure Oblivious version discards the out of bounds writes and continues to execute. It proceeds on to copy the first ten pairs of offsets into another data structure. Apache uses this data structure to apply the rewrite rule and generate the new URL. Because the rewrite rule uses a single digit to reference each captured substring (these substrings have names $0 through $9), it will never attempt to access any discarded substring offset data. The Failure-Oblivious version of Apache therefore processes each input correctly and continues on to successfully process any subsequent requests. Because the memory errors occur in irrelevant data structures and computations, Failure Oblivious computing eliminates the memory error without affecting the results of the computation at all.

Because Apache isolates request processing inside a pool of regenerating processes, the Bounds Check version successfully processes subsequent requests. The overhead of killing and restarting child processes, however, makes this version vulnerable to an attack that ties up the server by repeatedly presenting it with requests that trigger the error. To investigate this effect, we used several (local) machines to load the server with requests that trigger the error. We then used another client machine to repeatedly fetch the home page of our research project and measured the request throughput at the client. For this workload, the Failure Oblivious version provides a throughput roughly 5.7 times more than the Bounds Check version provides (the insecure Standard version provides a throughput roughly 4.8 times less than the Failure Oblivious version). We attribute the slowdown for the Bounds Check and Standard versions to process management overhead.

4.3.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Apache. The Small request serves an 5KByte page (this is the home page for our research project); the large request serves an 830KByte file used only for this experiment. Both requests were local — they came from the same machine on which Apache was running. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Small	44.4 ± 1.3%	47.1 ± 2.5%	1.06
Large	48.7 ± 1.8%	50.0 ± 1.3%	1.03

Figure 3: Request Processing Times for Apache (milliseconds)

4.3.4 Stability

For the last nine months we have been using the Failure Oblivious version of Apache to serve our research project's web site at www.flexc.csail.mit.edu; during this time period we measured approximately 400 requests a day from outside our institution. We also generated tens of thousands of requests from another machine, all of which were served correctly. We anticipate that we will continue to use the Failure Oblivious version to serve this web site for the foreseeable future.

During this time period we periodically presented the web server with requests that triggered the vulnerability discussed above. The Failure Oblivious version executed successfully through all of these attacks to continue to successfully service legitimate requests. We observed no anomalous behavior and received no complaints from the users of the web site.

4.4 Sendmail

Sendmail is the standard mail transfer agent for Linux and other Unix systems [15]. It is typically configured to run as a daemon which creates a new process to service each new mail transfer connection. This process executes a simple command language that allows the remote agent to transfer email messages to the Sendmail server, which may deliver the messages to local users or (if necessary) forward some or all of the messages on to other Sendmail servers. Versions of Sendmail earlier than 8.11.7 and 8.12.9 (8.11 and 8.12 are separate development threads) have a memory error vulnerability which is triggered when a remote attacker sends a carefully crafted email message through the Sendmail daemon [14]. We worked with Sendmail version 8.11.6.

4.4.1 The Memory Error

The memory error occurs when Sendmail parses a mail address. A prescan procedure processes the address one character at a time to transfer characters from the address into a fixed-size stack-allocated buffer. This transfer is coded to use a lookahead character and to treat the \ character specially. It is possible for there to be no lookahead character, in which case the integer variable that holds the lookahead character is set to -1. If this variable is set to -1 or contains a \ character that appears in an odd position (first, third, fifth, ...) in a sequence of contiguous \ characters in the address, the prescan skips the block of code that writes the lookahead character into the buffer (also skipping a check to see if the buffer has enough space to hold the lookahead character). It later writes a \ character into the buffer without a check if the lookahead character was \ and not -1. If the execution platform performs sign extension on character to integer assignments, an attack message containing an appropriately placed alternating sequence of -1 and \ characters in the address can therefore cause the prescan to write arbitrarily many \ characters beyond the end of the buffer.

4.4.2 Security and Resilience

The Standard version of Sendmail performs the out of bounds writes and corrupts its call stack. It is apparently possible for an attacker to exploit the memory error to cause the Sendmail server to execute arbitrary injected code [14]. The Bounds Check version exits with a memory error during initialization and fails to operate at all. The Failure Oblivious version is not vulnerable to the attack — when sent the attack message, it discards the out of bounds writes (preserving the integrity of the stack) and returns back out of the prescan to continue to parse the email address. The next step is to check if the input mail address is too long. This check fails, throwing Sendmail into an anticipated error case. The standard error processing logic in Sendmail then rejects the address, enabling Sendmail to continue on to successfully process subsequent commands.

4.4.3 Performance

Figure 4 presents the means and standard deviations of the request processing times for the Standard and Failure Oblivious versions of Sendmail. All times are given in milliseconds. The Receive Small request receives a message whose body is 4 bytes long; the Send Small request sends the same message. The Receive Large request receives a message whose body is 4Kbytes long; the Send Large request sends the same message. We performed each test at least twenty times to obtain the numbers in Figure 4.

Request	Standard	Failure Oblivious	Slowdown
Recv Small	$15.6 \pm 2.9\%$	$60.4 \pm 1.5\%$	3.9
Recv Large	$16.8 \pm 4.3\%$	$65.1 \pm 2.3\%$	3.9
Send Small	$20.3 \pm 3.2\%$	$75.0 \pm 3.4\%$	3.7
Send Large	$21.5 \pm 5.7\%$	$76.9 \pm 3.8\%$	3.6

Figure 4: Request Processing Times for Sendmail (milliseconds)

4.4.4 Stability

We installed the Failure Oblivious version of Sendmail on one of our machines and, over the course of several days, used it to send and receive hundreds of thousands of email messages. During this time we repeatedly sent the attack message through the Sendmail daemon, which continued through the attack to correctly process all subsequent Sendmail commands. All of the messages were correctly delivered with no problems. Our memory error logs indicate that Sendmail generates a steady stream of memory errors during its normal execution. In particular, every time the Sendmail daemon wakes up to check for incoming messages, it generates a memory error. This memory error apparently completely disables the Bounds Check version.

4.5 Midnight Commander

Midnight Commander is an open source file management tool that allows users to browse files and archives, copy files from one folder to another, and delete files [6]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [5]. We used Midnight Commander version 4.5.55 for our experiments.

4.5.1 The Memory Error

Midnight Commander converts absolute symbolic links in `tgz` files into links relative to the start of the `tgz` file. It uses the `strcat` procedure to build up the name of the relative link in a stack-allocated buffer. Unfortunately, the buffer is never initialized. If there are multiple symbolic links in the directory, the component names from all of the links simply accumulate sequentially in the buffer as Midnight Commander processes the set of links. If the combined length of all of the component names exceeds the length of the buffer, `strcat` writes the component names beyond the end of the buffer.

4.5.2 Security and Resilience

The Standard version performs the writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version detects the out of bounds access and terminates. The Failure Oblivious version discards the out of bounds writes, enabling Midnight Commander to continue and attempt to look up the data for the referenced file. This lookup always fails (apparently even for the first symbolic link, when the name in the buffer is correct). This is an anticipated case in the Midnight Commander code, which treats the symbolic link as a dangling link and displays it as such to the user. Midnight Commander then continues on to successfully process any subsequent user commands.

4.5.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Midnight Commander. The Copy request copies a 31Mbyte directory structure, the Move request moves a directory of the same size, the MkDir request makes a new directory, and the Delete request deletes a 3.2 Mbyte file. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

As these numbers indicate, the Failure Oblivious version is not dramatically slower than the Standard version. Moreover, because Midnight Commander is an interactive program, its performance is acceptable as long as it feels responsive to its users, and these performance results make it clear that the application of failure-

Request	Standard	Failure Oblivious	Slowdown
Copy	377 $\pm$ 0.7%	535 $\pm$ 2.0%	1.4
Move	0.30 $\pm$ 2.4%	0.406 $\pm$ 1.8%	1.4
MkDir	0.69 $\pm$ 7.0%	1.27 $\pm$ 6.6%	1.8
Delete	2.54 $\pm$ 11.3%	2.72 $\pm$ 11.1%	1.1

Figure 5: Request Processing Times for Midnight Commander (milliseconds)

oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.5.4 Stability

One of the authors uses Midnight Commander on a daily basis as his standard file manipulation tool. During the stability testing period, he used the Failure Oblivious version of Midnight Commander to manage his files. Periodically during the sessions he attempted to open the problematic archive (causing the program to execute through the resulting memory error), then went back to using the Midnight Commander to accomplish his work. Midnight Commander performed without a problem during this time.

The error log shows that Midnight Commander has a memory error that is triggered whenever a blank line occurs in its configuration file. We verified that this error completely disabled the Bounds Check version until we removed the blank lines. The Failure Oblivious version, on the other hand, executed successfully through all memory errors to perform flawlessly for all requests.

4.6 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [8]. It is descended from ELM [4] and supports a variety of features including email threading and correct NFS mail spool locking. We used Mutt version 1.4. As described at [7] and discussed in Section 2, this version is vulnerable to an attack that exploits a memory error in the conversion from UTF-8 to UTF-7 string formats.

4.6.1 The Memory Error

When Mutt opens a mailbox with an IMAP address, it converts the mail folder name from UTF-8 to UTF-7 character encoding. Mutt allocates (in the heap) a temporary character buffer to hold the UTF-7 encoded name. Because UTF-8 to UTF-7 conversion can increase the length of the name, Mutt allocates a buffer twice as long as the UTF-8 name to hold the converted UTF-7 name. However, this buffer is not, in general, long enough — the conversion can increase the length of the UTF-8 name by as much as a factor of 7/3 and not just a factor of 2. When presented with an appropriately constructed UTF-8 folder name, Mutt writes the converted name beyond the end of the UTF-7 buffer.

4.6.2 Security and Resilience

The Standard version performs the writes, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects the memory error and terminates before the user interface comes up. The Failure Oblivious version discards the out of bounds writes, effectively truncating the converted name. Note that even though the UTF-7 buffer may contain no null characters, the folder name is effectively null-terminated: reads beyond the end of the buffer will eventually return null. Once Mutt has obtained the converted folder name, the next step is to place a quoted and escaped version of the name into yet another buffer, then pass this name on as part of a command to the IMAP server. The IMAP server returns an error code indicating that the folder does not exist, Mutt's standard error-handling logic handles the returned error code, and Mutt continues on to successfully process any subsequent user commands.

4.6.3 Performance

Figure 6 presents the request processing times for the Standard and Failure Oblivious versions of Mutt. The Read request reads a selected empty message and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	.655 $\pm$ 4.3%	2.31 $\pm$ 4.8%	3.6
Move	6.94 $\pm$ 6.2%	9.78 $\pm$ 6.2%	1.4

Figure 6: Request Processing Times for Mutt (milliseconds)

Because Mutt is an interactive program, its performance is acceptable as long as it feels responsive to its users. These performance results make it clear that the application of failure-oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.6.4 Stability

During the stability testing period we used the Failure Oblivious version of Mutt to process email messages. We configured Mutt to trigger the security vulnerability described above when it loaded. Mutt successfully executed through the resulting memory errors to correctly execute all of his requests. We were able to read, forward, and compose mail with no problems even after executing through the memory error. We also used Mutt to process (with no problems) a large mail folder containing over 100,000 messages.

4.7 Discussion

Despite the fact that the dynamic bounds checks have, in theory, the potential to substantially degrade the performance, for several of our servers the overhead is relatively small — the execution times of many of the tasks we measured are apparently dominated by activities (such I/O or operating system functionality) outside the program. Because failure-oblivious computing does not affect the efficiency of these activities, the amortized overhead is relatively small. Moreover, several of our servers are interactive, and interactive tasks can tolerate substantial execution time increases as long as the system maintains its interactive feel. Our results show that failure-oblivious computing maintained acceptable interactive response times for all of our interactive tasks, even for tasks with substantial execution time increases.

For servers, a monitor that detects memory errors and reboots the server when it commits such an error might seem to provide an obvious potential alternative to failure-oblivious computing. Apache, for example, implements a regenerating pool of child processes. The net effect is that the Bounds Check version of Apache can terminate child processes at the first memory error without impairing its ability to continue to service new requests. In comparison with the Failure Oblivious version, the only downside is the performance degradation associated with the resulting increase in process management overhead.

The situation is somewhat different for Pine, Mutt, and Midnight Commander. All of these programs initialize with no memory errors on standard workloads. But once the mailbox contains a message that elicits a memory error (Pine), the system is configured to use a mail folder whose name elicits a memory error (Mutt), or the configuration file contains a blank line (Midnight Commander), the Bounds Check versions exit during initialization. In this situation, restarting is of no use because the restarted computations would, once again, simply exit during initialization. Because these errors are triggered only by carefully crafted or unusual inputs, they could easily make it through a fairly rigorous testing process without being detected. These servers illustrate how aggressively terminating computations at the first memory error can leave deployed systems vulnerable to unanticipated inputs that trigger memory errors and persist or recurr in the environment.

Because Sendmail has a memory error whenever it wakes up to check for work, the Bounds Check version is simply unusable with or without restarting. But note that because the memory errors occur on every execution, it should be possible to use the Bounds Check version to find and eliminate them (as well as any other reproducible memory errors that occur during testing). Even with this change, however, terminating and restarting Sendmail might prove to be problematic — the Sendmail monitor would somehow have to avoid repeatedly presenting Sendmail with messages that triggered a memory error. In contrast, the Failure Oblivious version of Sendmail correctly executed through memory errors to correctly process subsequent messages and the Failure Oblivious version of Pine correctly processed mail messages with headers that elicited memory errors.

5 Related Work

We first note that failure-oblivious computing is an instance of acceptability-oriented computing [47]. Acceptability-oriented computing replaces the concept of program correctness with a set of *acceptability properties* that must hold for the execution of the program to remain acceptable. The programmer then builds and deploys *acceptability enforcement mechanisms* whose actions ensure that these acceptability properties do, in fact, hold. In the case of failure-oblivious computing, the acceptability properties are the absence of memory errors and continued execution; the acceptability enforcement mechanism discards invalid writes and returns manufactured values for invalid reads.

Memory errors, failures, and failure recovery have been core concerns in the field of computer systems since its inception. We discuss related work in these areas.

5.1 Variants and Extensions

We have implemented with several variants and extensions of our basic failure-oblivious compiler. These include a compiler that implements *boundless memory blocks* — instead of discarding invalid writes, the generated code stores the values in a hash table indexed under the data unit identifier and offset [48]. Corresponding invalid reads return the appropriate stored values. This variant eliminates size calculation errors — if the program logic is otherwise acceptable, the program will execute acceptably. Another variant redirects out of bounds accesses back into the accessed data unit at an appropriate offset. This strategy may help related sets of out of bounds reads return consistent values from properly initialized data units. Our experience indicates that our set of servers works acceptably with both of these variants.

5.2 Transactional Function Termination

Researchers have also developed a technique to protect servers against buffer-overflow attacks by dynamically detecting buffer overflows, then immediately terminating the enclosing function and continuing on to execute the code immediately following the corresponding function call [52]. The results indicate that, in many cases, the program can continue on to execute acceptably after the premature function termination. This experience is consistent with our experience that servers can continue to execute successfully through memory errors if they simply discard out of bounds writes and manufacture values for out of bounds reads.

5.3 Safe-C Compilers

Our work builds directly on previous research into memory-safe C implementations [17, 58, 45, 36, 50, 37]. Building on Ruwase and Lam's implementation enabled us to apply failure-oblivious computing directly to legacy programs without modification (some implementations also have this property [58]); some other implementations may require source code changes [22, 38].

It is also feasible to apply failure-oblivious computing to safe languages such as Java or ML by simply replacing the generated code that throws an exception in response to a memory error. As for safe-C implementations, the new code would simply discard illegal writes and return manufactured values for illegal reads.

5.4 Static Analysis

It is also possible to attack the memory error problem directly at its source: a combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of memory errors [28, 27, 57, 49]. All of these techniques share the same advantage (a static guarantee that the program will not exhibit a specific kind of memory error) and drawbacks (the need for programmer annotations or the possibility of conservatively rejecting safe programs). Even if the analysis is not able to verify that the entire program is free of memory errors, it may be able to statically recognize some accesses that will never cause a memory error, remove the dynamic checks for those accesses, and thereby reduce any dynamic checking overhead [32, 18, 49].

Researchers have also developed unsound, incomplete analyses that heuristically identify potential errors [54, 19]. The advantage is that such approaches typically require no annotations and scale better to larger programs; the disadvantage is that (because they are unsound) they may miss some genuine memory errors.

5.5 Buffer-Overrun Detection Tools

Researchers have developed techniques that are designed to detect buffer-overrun attacks after they have occurred, then halt the execution of the program before the attack can take effect. StackGuard [23] and StackShield [16] modify the compiler to generate code to detect attacks that overwrite the return address on the stack; StackShield also performs range checks to detect overwritten function pointers. It is also possible to apply buffer-overrun detection directly to binaries. Purify instruments the binary to detect a range of memory errors, including buffer overruns [34]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [39]. A key difference is that failure-oblivious computing prevents the attack from performing the writes that corrupt the address space, which enables the program to continue to execute successfully.

5.6 Rebooting

A traditional and widely used error recovery mechanism is to reboot the system, with repair applied during the reboot if necessary to bring the system back up successfully [30]. Mechanisms such as fast reboots [51] and checkpointing [41, 42] can improve the performance of the basic reboot process.

It is also possible to subdivide (potentially recursively) a system into isolated components, then apply a partial reboot strategy at the granularity of the components. By promoting the construction of the operating system as a collection of small components, microkernel architectures [46, 33, 29] support the application of this approach to operating systems. It is also possible to use mechanisms such as software-based fault isolation [55] or fine-grained hardware memory protection [56] to apply this strategy to selected parts of monolithic operating systems such as kernel extensions. The experimental results show that this approach can eliminate the vast majority of system crashes caused by such extensions [53]. Helper agents are often useful to facilitate the clean termination and reintegration of the restarted component back into the running system (this approach generalizes to support arbitrary recovery actions) [53]. It may also be worthwhile to recursively restart larger and larger subsystems until the system successfully recovers [20].

Failure-oblivious computing differs in that it is designed to keep the system operating through errors instead of restarting. The potential advantages include better availability because of the elimination of down time and the elimination of vulnerabilities to persistent errors. Rebooting, on the other hand, may help ensure that the system stays more closely within the anticipated operating envelope.

5.7 Manual Error Detection and Recovery

Motivated in part by the need to avoid rebooting, researchers have developed more fine-grain error recovery mechanisms. The Lucent 5ESS switch and the IBM MVS operating system, for example, both contain software components that detect and attempt to repair inconsistent data structures [35, 44, 31]. Other techniques include failure recovery blocks and exception handlers, both of which may contain hand-coded recovery algorithms [43].

To apply these techniques, the programmer must anticipate some aspects of the error and, based on this understanding, develop an appropriate recovery strategy. Failure-oblivious computing, on the other hand, can be applied without programmer intervention to any system and may therefore make the system oblivious to even completely unanticipated errors. Of course, this generality cuts both ways — in particular, failure-oblivious computing may produce less appropriate responses to anticipated errors. We therefore view failure-oblivious computing as largely orthogonal to more application-tailored recovery mechanisms (although failure-oblivious computing may eliminate some of the errors that these mechanisms would otherwise have handled).

Data structure repair [26] occupies a middle ground. Like more traditional error detection and recovery techniques, it requires the programmer to provide some application-specific information (in the case of data structure repair, a data structure consistency specification). But because there is no explicit recovery procedure and because the consistency specification is not tied to specific blocks of code, data structure repair may enable systems to more effectively recover from unanticipated data structure corruption errors.

6 Conclusion

The seemingly inherent brittleness, complexity, and vulnerability (to both errors and attacks) of computer programs can make them frustrating or even dangerous to use. While existing memory-safe languages and memory-safe implementations of unsafe languages may eliminate memory-error vulnerabilities, they can also decrease availability by aggressively throwing exceptions or even terminating the program at the first sign of an error.

Our results show that failure-oblivious computation enhances availability, resilience, and security by continuing to execute through memory errors while ensuring that such errors do not corrupt the address space or data structures of the computation. In many cases failure-oblivious computing can automatically convert unanticipated and dangerous inputs or data into anticipated error cases that the program is designed to handle correctly. The result is that the program survives the unanticipated situation, returns back into its normal operating envelope, and continues to satisfy the needs of its users.

One of the major long-term goals of computer science has been understanding how to build more robust, resilient programs that can flexibly and successfully cope with unanticipated situations. Our research suggests that, remarkably, current systems may already have a substantial capacity for exhibiting this kind of desirable behavior if we only provide a way for them to ignore their errors, protect their data structures from damage, and continue to execute.

Acknowledgements

The authors would like to thank our shepherd David Wagner and the anonymous reviewers for their thoughtful and helpful comments. This research was supported in part by the Singapore-MIT Alliance and NSF grants CCR00-86154, CCR00-63513, CCR00-73513, CCR-0209075, CCR-0341620, and CCR-0325283.

References

[1] Apache HTTP Server exploit. www.securityfocus.com/bid/8911/discussion/.

[2] CERT/CC. Advisories 2002. www.cert.org/advisories.

[3] CNN Report on Code Red. www.cnn.com/2001/TECH/internet/08/08/code.red.II/.

[4] ELM. www.instinct.org/elm/.

[5] Midnight Commander exploit. www.securityfocus.com/bid/8658/discussion/.

[6] Midnight Commander website. www.ibiblio.org/mc/.

[7] Mutt exploit. www.securiteam.com/unixfocus/5FP0T0U9FU.html.

[8] Mutt website. www.mutt.org.

[9] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.

[10] Pine exploit. www.securityfocus.com/bid/6120/discussion.

[11] Pine website. www.washington.edu/pine/.

[12] SecuriTeam website. www.securiteam.com.

[13] Security Focus website. www.securityfocus.com.

[14] Sendmail exploit. www.securityfocus.com/bid/7230/discussion/.

[15] Sendmail website. www.sendmail.org.

[16] Stackshield. www.angelfire.com/sk/stackshield.

[17] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 2004.

[18] R. Bodik, R. Gupta, and V. Sarkar. Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[19] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.

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

[21] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[22] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

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

[24] J. Darley and B. Latane. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology*, pages 377–383, Aug. 1968.

[25] W. E. Deming. *Out of the Crisis*. MIT Press, 2000.

[26] B. Demsky and M. 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.

[27] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.

[28] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[29] D. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* , Dec. 1995.

[30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[31] N. Gupta, L. Jagadeesan, E. Koutsofios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.

[32] R. Gupta. Optimizing array bounds checks using flow analysis. In *ACM Letters on Programming Languages and Systems, 2(1-4):135-150*, March 1993.

[33] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer Usenix Conference* , June 1993.

[34] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[35] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

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

[37] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of Third International Workshop On Automatic Debugging*, May 1997.

[38] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference Proceedings*, 1983.

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

[40] B. Latane and J. Darley. Group inhibition of bystander intervention in emergencies. *Journal of Personality and Social Psychology*, pages 215–221, Oct. 1968.

[41] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[42] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.

[43] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.

[44] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.

[45] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.

[46] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference* , July 1986.

[47] M. Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session*, Oct. 2003.

[48] M. Rinard, C. Cadar, D. Roy, D. Dumitran, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference*, Dec. 2004.

[49] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation* , June 2000.

[50] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[51] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.

[52] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks. Technical Report CUCS-031-04, Columbia University Computer Science Department, September 2004.

[53] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles* , Dec. 2003.

[54] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium*, 2000.

[55] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* , Dec. 1994.

[56] E. Witchel, J. Cates, and K. Asanovic. Mondriaan memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[57] H. Xi and F. Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[58] S. H. Yong and S. Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.

[59] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.