Enhancing Availability and Security Through Failure-Oblivious Computing

Martin Rinard, Cristian Cadar, Daniel Dumitran,
Daniel M. Roy, and William S. Beebee, Jr.
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

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

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

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) [25, 53, 47, 39, 49, 40]. These checks use additional information about the (dynamic) layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a dynamic 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 access 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 (in effect, turning the write into a nop) and continue to execute the program. We call a computation that uses this execution strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

One potential objection is that failure-oblivious computing cannot possibly work, because even if the program continues to execute through the memory error, it will generate the wrong result. As discussed below in Section 4, many important programs have characteristics that interact well with continued execution through memory errors *as long as failure-oblivious computing prevents these errors from corrupting the program's address space or data structures.* And in fact, our experimental results show that failure-oblivious computing can substantially improve the availability, robustness, and security of such programs.

Another potential objection is that the program should stop at the first memory error so that a programmer can find and eliminate the error. We agree that in many active development scenarios it is better to stop at the first error. But in the vast majority of usage scenarios debugging is simply not a viable option — very few users have the skills, program knowledge, access to source code, or time required to effectively debug an error in one of the many programs that they use. Moreover, our results indicate that failure-oblivious computing enables many programs to provide perfectly acceptable service to their users even after they attempt to commit one or more memory errors. Given the choice between no execution (because the program stopped or crashed at a memory error), corrupted execution (because the memory error corrupted the address space), and continued failure-oblivious execution, it is clear to us that, for the programs that we discuss in Section 4, almost all users will prefer failure-oblivious execution because it will usually enable them to continue to work productively.

1.2 Servers and Buffer-Overrun Attacks

Consider, for example, a server (such as a web server) that processes incoming requests from a variety of clients. Even if the server processes most requests successfully, some requests may trigger latent memory errors. A common error occurs when the program allocates a fixed-size stack-allocated buffer to hold input strings, then fails to check that each input string actually fits in the buffer. An attacker can exploit this error by passing in a long input string containing executable code. The string overwrites the stack contents, injecting the code into the stack. By including a pointer to this code in the appropriate place in the input string, the attacker can coerce the server into executing the (arbitrary) injected code.

Such attacks are called buffer-overrun attacks. They are currently the most common source of exploited security vulnerabilities in modern networked computer systems [2]. Reasonable estimates place the total cost of such attacks in the billions of dollars annually [3].

Failure-oblivious computing makes a server invulnerable to buffer-overrun attacks: if an input string does not fit into a buffer, the failure-oblivious server simply discards the excess characters, preserving the integrity of the stack. The server typically recognizes the request as invalid, returns some kind of error response, then continues to process the next request. From the perspective of the server, failure-oblivious computing has converted a dangerous, unanticipated attack into a benign, anticipated invalid input. For this class of applications, failure-oblivious computing offers the following advantages:

- **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 in particular and memory-error-based data structure corruption attacks in general.

- **Availability:** The combination of protection against data structure corruption and continued execution in the face of programming errors can dramatically increase the availability of the computation. This combination is especially valuable for servers because it 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 programming errors).

- **Minimal Adoption Cost:** The net adoption cost to the developer is to recompile their program 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 and with almost no effort to today's software infrastructure.

1.3 Multiple Items or Outputs

Many programs (such as mail readers) process multiple items (such as mail messages). Without failure-oblivious computing, a memory error in the computation associated with one of the items can cause the program to fail to process the rest of the items. For unsafe languages the memory error may cause the program to crash or corrupt data structures required to process the rest of the items; for safe languages the errors may generate exceptions that divert the execution from processing subsequent items.

Failure-oblivious computing enables the program to continue to execute through errors to process all of the items. And our experience with such programs indicates that because failure-oblivious computing prevents memory errors from corrupting the data structures, the program is able to process subsequent items successfully. Failure-oblivious computing is therefore able to substantially increase the availability of these kinds of programs. And once again, the adoption cost is minimal: a simple recompile of the existing program with no source code changes.

This basic class of applications generalizes to include applications that generate multiple outputs; in many cases some of the outputs are more important than others. By enabling the computation to execute through errors to generate all of its outputs, failure-oblivious computing can enable the program to produce its important outputs even if the computations that generate other outputs would otherwise fail because of memory errors.

1.4 Overview of Results

We have implemented a C compiler that generates failure-oblivious code. We have obtained several widely-used public-domain programs that contain known memory errors (which usually make them vulnerable to a variety of remote attacks) and used our compiler to obtain failure-oblivious versions of these benchmarks. Our results show that these versions are 1) not vulnerable to the memory errors or the attacks that these memory errors otherwise enable, and 2) continue to correctly service legitimate requests and interactions even after failed attacks trigger the errors. Using a safe-C compiler for these programs is either impractical (because the programs have memory errors during initialization and fail to operate at all) or suboptimal (because it causes the programs to unnecessarily deny service to legitimate users).

1.5 Benefits 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:

- **Increased Resilience:** Current computer systems are notoriously brittle — all too often, a single error can cause an entire system to fail. In some cases, failure-oblivious computing can eliminate the effect of an error entirely; even when it is unable to do so, it often converts outright failure into graceful degradation and enables the system to continue to operate successfully on most of its inputs. The net result is a dramatic increase in the overall resilience of the system and its ability to continue to operate acceptably in the face of errors, unexpected inputs, and attacks.

- **Increased Security:** Failure-oblivious computing eliminates important security vulnerabilities by preventing an attacker from exploiting buffer overruns and other memory errors. Moreover, we expect failure-oblivious computing to be more palatable to end users than safe compilation approaches (which react to memory errors by throwing an exception or terminating the program) because it often enables the program to survive the attacks to continue to execute productively.

- **Reduced Development Costs:** With current execution strategies, a single memory error in any part of the program can cause the entire program to fail. This fact generates enormous pressure to find and eliminate as many such errors as possible. The result is a long, complicated development process characterized by extensive testing (to find the errors), frequent code changes (to fix the errors), and the potential for system disruption (as the effect of the code changes propagate to other parts of the system).

  Because failure-oblivious computing reduces the consequences of memory errors, it can reduce the need for extensive testing and make it feasible to simply leave more errors in place. The result may be a substantial reduction in development costs.

- **Reduced Administration Overhead:** One of the most challenging system administration tasks is ensuring that systems are kept up to date with a constant stream of patches and upgrades; this stream is driven, in large part, by the need to eliminate memory errors (and the associated security vulnerabilities) in otherwise perfectly acceptable programs. Because failure-oblivious computing eliminates this class of errors, it promises to make it possible for system administrators to safely ignore many of these patches and upgrade or 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.

  Detecting and recovering from successful attacks is also a challenging and time-consuming system administration task. Failure-oblivious computing promises to reduce the frequency of successful attacks and therefore reduce the administration effort required to detect and recover from such attacks.

  More generally, other kinds of program failures often generate a similar need for human intervention to detect and recover from the failure. Failure-oblivious computing, because it enables programs to continue to execute through otherwise fatal errors, promises to eliminate many of these failures and therefore to reduce administration costs.

- **Safer Integration:** One of the potential hazards of incorporating a foreign component into a program is the potential that it may contain memory errors that cause the program to fail, either directly within the component itself or indirectly in another component whose data structures it has corrupted. Because failure-oblivious computing reduces the consequences of such memory errors, it lowers the risks associated with the use of such components. It may therefore improve the availability and reliability of systems built out of multiple components from different sources and increase the feasibility of aggressive code reuse.

In our view, the primary drawback of failure-oblivious computing is the potential it has 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 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.

There are several ways that the programmer can ameliorate the impact of this unanticipated execution. First, our results indicate that failure-oblivious computing often has the effect of converting unanticipated situations into anticipated error cases, which put the program back into its anticipated set of executions fairly quickly. Second, the programmer can analyze the characteristics of the program (and the context in which it will be used) and decide if it is appropriate for failure-oblivious computing.

1.6 Scope

We anticipate that failure-oblivious computing will be especially appropriate when users can easily determine if the program is operating acceptably and enabling them to continue to work productively. This is the case for mailers, servers, system administration tools, operating systems, document processing systems, and many other programs that are part of our standard interactive computing environment. We also anticipate that failure-oblivious computing will be appropriate when external intervention is impractical, failure has catastrophic results, and there is little or nothing to lose from continuing to execute.

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) for which the user must simply trust the output because there is no easy way to determine by inspection if the output is correct or not. We also anticipate that it will be less appropriate for safety-critical applications in which it is always safe to terminate the computation and external intervention is readily available.

1.7 Contributions

This paper makes the following contributions:

- **Failure-Oblivious Computing:** It presents 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 programs. Our results show that:

  – **Standard Compilation:** With the standard unsafe C compiler, the programs 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

[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 correctly.

it detects a memory error, the programs exit either during initialization (preventing users from using them at all) or when presented with an input that triggers a memory error (denying the user access to the services that the program is intended to provide).

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

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 4.5. 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 7/2. 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 crashes with a segmentation violation. With safe-C compilers, Mutt exits because of a memory error during initialization 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 which mailbox it wants to open. Without failure-oblivious computing, Mutt crashes or terminates before it issues the request. With failure-oblivious computing, the mailbox name is incorrect and the mail server responds with an error. Mutt correctly handles the error and continues to execute, enabling the user to continue to read mail.

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 is designed to handle correctly.

```
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

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.

3.1 Checking Code

Our implementation uses a checking scheme originally developed by Jones and Kelly [40] and then significantly enhanced by Ruwase and Lam [49]. The scheme is currently implemented as a modification to the GNU C compiler (gcc).

3.1.1 Jones and Kelly's Scheme

Jones and Kelly's scheme maintains a table that maps locations to data units (each struct, array, and variable is a data unit). It uses this table to track intended data units and distinguish in-bounds from out-of-bounds pointers as follows:

- **Base Case:** A base pointer is the address of an array, struct or variable allocated on the stack or heap, or the value returned by `malloc`. All base pointers are in bounds. The *intended data unit* of the base pointer is the corresponding array, struct, variable, or allocated block of memory to which it refers.

- **Pointer Arithmetic:** All pointer arithmetic expressions contain a starting pointer (for example, a pointer variable or the name of a statically allocated array) and an offset. We say that the value of the expression is *derived from* the starting pointer. A derived pointer is in bounds if and only if the corresponding starting pointer is in bounds and the derived pointer points into the same data unit as the starting pointer. Regardless of where the starting and derived pointers point, they have the same intended data unit.

- **Pointer Variables:** A pointer variable is in bounds if and only if it was assigned to in-bounds pointer. It has the same intended data unit as the pointer to which it was assigned.

Jones and Kelly distinguish a valid out-of-bounds pointer, which points to the next byte after its intended data unit, from an invalid out-of-bounds pointer, which points to some other address not in its intended data unit. They implement this distinction by padding each data item with an extra byte. A valid out-of-bounds pointer points to this extra byte; all invalid out-of-bounds pointers have the value ILLEGAL (-2). This distinction is designed to support code that uses valid out-of-bounds pointers in the termination condition of loops that use pointer arithmetic to scan arrays.

Finally, Jones and Kelly instrument the code to check the status of each pointer before they dereference it. Dereferencing an in-bounds pointer returns the referent value. Dereferencing an out-of-bounds pointer causes the program to halt with an error.

3.1.2 Ruwase and Lam's Enhancement

Jones and Kelly's scheme does not support programs that first use pointer arithmetic to obtain a pointer to a location past the end of the intended data unit, then use pointer arithmetic again to jump back into the intended data unit and access data stored in this data unit. While the behavior of programs that do this is undefined according to the ANSI C standard, in practice many C programs use this technique [49]. Ruwase and Lam's extension uses an *out-of-bounds objects* (OOBs) to support such behavior [49].

As in standard C compilation, in-bounds pointers refer directly into their intended data unit. Whenever the program computes an out-of-bounds pointer, Ruwase and Lam's enhancement generates an OOB object that contains the starting address of the intended data unit and the offset from the start of that data unit. Instead of pointing off to some arbitrary memory location outside of the intended data unit or containing the value ILLEGAL (-2), the pointer points to the OOB object. The generated code checks pointer dereferences for the presence of OOB objects and uses this mechanism to halt the program if it attempts to dereference an out-of-bounds pointer. The generated code also uses OOB objects to precisely track data unit offsets and appropriately translate pointers derived from out-of-bounds pointers back into the in-bounds pointer representation if the new pointer jumps back inside the intended data unit. In practice, this enhancement significantly increases the range of programs that can execute without terminating because of a failed memory error check [49].

3.2 Continuation Code

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. 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 by far the most common values in computer programs [24], 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. CASE STUDIES

We have implemented a compiler that generates failure-oblivious code, obtained several widely-used open-source programs with known memory errors, and evaluated the impact of failure-oblivious computing on their behavior. Figure 2 presents the programs in our test suite. These programs implement different system tasks such as reading and distributing email, serving content to remote clients, or supporting a variety of administration tasks. Many of them are are key components of the Linux-based open-source interactive computing environment.

Program	**Purpose**
Pine	Mail User Agent
Midnight Commander	File Manager
Sendmail	Mail Transfer Agent
Mutt	Mail User Agent
Samba	File Server
WsMp3	mp3 Server
Apache	http Server
Gzip	compression utility

Figure 2: Programs in Test Suite

4.1 Methodology

We evaluate the behavior of three different versions of each program: the *standard* version compiled with a standard C compiler (this version is vulnerable to any memory errors that the program may contain), the *safe* version compiled with the CRED safe-C compiler [49] (this version terminates the program with an error message at the first memory error), and the *failure-oblivious* version compiled with our compiler (this compiler is derived from CRED by changing the generated code to discard illegal writes and return a predetermined sequence of values for illegal reads). For each program we chose a representative workload that contains an input that triggers the memory error. In many cases the workloads contain inputs that exploit known security vulnerabilities documented by vulnerability-tracking organizations such as Security Focus [18] and SecuriTeam [17]. We ran all the programs on a Red Hat 8.0 Linux workstation with two 2.8 GHz Pentium 4 processors and 2 GBytes of RAM.

4.2 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [13]. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks. We used 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 legal From fields; it is possible for a remote attacker to exploit this vulnerability to execute arbitrary code on the user's machine [12]. Our workload contains a mail file with a From field that triggers this memory error. In the standard version, the memory error causes Pine to corrupt its heap so that it crashes with an abort message. The safe version detects the memory error and terminates the computation with an error message identifying the memory error.

With both the standard and safe 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 program. 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 again. While the safe version protects the user against injected code attacks, it prevents the user from using Pine to read mail as long as the mail file contains the problematic From field.[2]

The failure-oblivious version, on the other hand, continues to execute through the memory error, enabling the user to process their mail. The user can even read and forward the message with the problematic From field. Moreover, the user can see the entire From field in the user interface — the memory error involves internal data structures that are not visible in the user interface. For this workload, failure-oblivious computing enables Pine to exhibit completely correct behavior with no degradation whatsoever.

Because Pine is an interactive program, its performance is acceptable as long as it provides good interactive responses to its users. There were no perceptible differences in the pause times when using Pine to read, forward, or compose email between any of the versions — all versions responded without perceptible delay during interactive use. The failure oblivious version of Pine incurs a slight pause before starting; the standard version starts almost instantaneously. The startup times are not affected by the size of the mail file.

4.3 Midnight Commander

Midnight Commander is an open source file management tool that allows users to browse their files and archives, copy files from one folder to another, and delete files [8]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [7]. Our workload contains a `tgz` archive designed to exploit this vulnerability.

On our workload, the standard version terminates with a segmentation violation when the user attempts to open the problematic `tgz` archive. Because Midnight Commander has memory errors in its initialization code, the safe version terminates with an error message before it finishes initialization. The result is that the user is unable to use this version of Midnight Commander at all.

The failure-oblivious version, on the other hand, starts with no problems. When the user attempts to open the problematic `tgz` archive, Midnight Commander correctly displays the names of the two symbolic links in the archive. Because these links point off to non-existent files, Midnight Commander correctly displays an error message when the user attempts to open them. Midnight Commander continues to execute successfully throughout the entire session; in particular, the user can continue to use Midnight Commander to browse, copy, or delete other files even after processing the problematic `tgz` archive. For this workload (and for all others that we know of) failure-oblivious computing enables Midnight Commander to exhibit completely correct behavior with no degradation at all.

Like Pine, Midnight Commander is an interactive program. Also like Pine, we observed no perceptible difference in the interactive response times of the different versions — all versions responded without perceptible delay during interactive use.

4.4 Sendmail

Sendmail is the standard mail transfer agent for Linux and other Unix systems [20]. It is typically configured to

[2]Unlike several other of our programs, the safe version of Pine initializes without memory errors and can successfully process many mail files without memory errors.

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 allows a remote attacker to inject and execute arbitrary code on the machine running the Sendmail server [19]. This code executes with the same permissions as the Sendmail server (which is typically root).

We worked with Sendmail version 8.11.6. With standard compilation, we were able to launch an attack that exploited the memory error to provide the attacker with a root shell. The safe version exits with a memory error during initialization and fails to operate at all. The failure-oblivious version is not vulnerable to the attack — the process servicing the remote connection does not provide the attacker with a shell (root or otherwise). It instead prints an error message and continues to execute successfully to process legitimate Sendmail commands and appropriately deliver mail even after the attack has attempted to exploit the vulnerability.

We measured the performance of the different versions by developing a program that uses SMTP to interact with sendmail to deliver 200 messages. We ran the two operational versions (standard and failure-oblivious) five times, observing (each time) the time required to send the 200 messages. The times were comparable at between six and seven seconds for each trial; the differences between the two versions were within the variance of each set of trials, indicating no observable difference between the performance of the two versions.

4.5 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [10]. 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 [9] 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. We were able to develop an attack that exploited this vulnerability to crash Mutt. It is possible for a remote IMAP server to use this attack to remotely crash Mutt; it may also be possible for the IMAP server to exploit the vulnerability to inject and execute arbitrary code.

The standard version of Mutt crashes when it attempts to open a remote mailbox with a name chosen to trigger the memory error. The safe version exits with a memory error during initialization before the user interface comes up. When the failure-oblivious version attempts to open the mailbox, it generates an error message indicating that the mailbox does not exist (because the translated name has been truncated), then continues to execute successfully and allow the user to process mail from other mailboxes.

Because Mutt is an interactive program, the relevant performance metric is the response time. All of the versions of Mutt are very responsive for interactive tasks, exhibiting basically instantaneous response time. In our performance tests, the failure-oblivious version of Mutt loads local mailboxes at the rate of approximately 300 microseconds per message, while the standard version requires approximately 60 microseconds per message. To put these numbers in perspective, the mailbox would need to contain over 3000 messages before the failure oblivious version of Mutt would require more than a second to load the mailbox.

4.6 Samba

Samba is a widely-used file sharing utility that allows machines running a variety of operating systems to share files [41, 16]. We configured our Samba 2.2.5 server to serve files to other Samba clients. This version of Samba has a memory corruption error that (with standard compilation) enables a remote user to obtain a root shell on the Samba server machine [14, 15]. We were able to reproduce this exploit and include it in our workload.

On our workload, the standard version is vulnerable to the exploit and allows the remote attacker to obtain a root shell on the Samba server. The safe version initializes successfully and correctly serves standard Samba requests without memory errors until presented with the attack, at which point the child process serving the connection exits. The failure-oblivious version executes successfully through the attack. Because the Samba server is multithreaded, both the safe and failure-oblivious versions continue to successfully respond to requests on new Samba connections.

We measured the performance of the different versions of Samba by transferring 10 files (comprising 610 MBytes of data) between a Samba server and a Samba client on the same machine. Both versions executed the transfer at approximately 12 Mbytes per second, with no significant difference in the transfer rates between the different versions.

4.7 WsMp3

WsMp3 is a streaming web server designed to distribute MP3 files to remote clients [23]. WsMp3 version 0.0.5 contains a memory-error vulnerability that, according to the vulnerability reports, may enable a remote attacker to execute arbitrary code on the WsMp3 server [22]. Our workload contains a request that triggers this memory error.

On our workload, the standard version terminates with a segmentation violation when it processes the problematic request. The safe version operates correctly without memory errors on normal requests, but terminates with an error message on the request that triggers the memory error. Because the server is single threaded, the entire server terminates and the service is completely unavailable until it is restarted. The failure-oblivious version, on the other hand, executes through the problematic request and continues to correctly serve MP3 files without a problem.

We measured the amount of time it took for the different versions of WsMp3 to serve a 60 Mbyte file 10 times to a client on the same machine. There was no significant difference in the transfer times — all versions transfer data at approximately 40 Mbytes per second. This result is consistent with the results reported in [49]).

4.8 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 (more than all other web servers combined) [11]. To improve performance, it maintains a pool of child processes that it dynamically assigns to process new connections [48]. The Apache 2.0.47

mod_alias implementation contains a vulnerability that, under certain circumstances, allows a remote attacker to trigger a memory error [1]. The vulnerability reports indicate that this error may enable the remote attacker to inject and execute arbitrary code on the Apache server [1]. Our workload contains a request that triggers this vulnerability.

Each Apache request is handled by a child process assigned to service the connection carrying the request. With standard compilation, the child process terminates with a segmentation violation when presented with the attack. The Apache parent process then creates a new child process to take its place. With safe compilation, Apache correctly processes legitimate requests without memory errors until presented with the attack that triggers the error. At this point the child process serving the connection detects the memory and terminates. The parent Apache process then creates a new child process to take its place. With failure-oblivious compilation, the child process redirects the attacking request to a non-existent URL (part of the functionality of the mod_alias module is to redirect web page requests). The child process executes successfully through the attack to correctly process subsequent requests.

We were able to observe no interactive response time differences between the standard and failure-oblivious versions of Apache. We also used measured the time required to transfer a 19 Mbyte file from the Apache server to a client. For a remote client, both versions transferred the file at approximately .8 Mbytes per second. For a local client, both versions transferred at approximately 85 Mbytes per second.

4.9 Gzip

Gzip is a popular compression (and uncompression) utility [6]. Gzip 1.2.4a has a memory error in its file name processing code [5] that an attacker can exploit to execute arbitrary code. This vulnerability may be of concern because many Internet servers and clients silently invoke gzip as part of their normal execution.

Our workload uses gzip to compress several files; one of the files triggers the memory error. The standard version terminates with a segmentation fault when it attempts to process the problematic file; the remaining files are not processed. The safe version processes normal files without memory errors, but terminates when it attempts to process the problematic file. Once again, the remaining files are not processed. The failure-oblivious version prints an error message when asked to process the problematic files, then proceeds through the memory error to successfully process all of the other files.

The standard version compresses files at a rate of approximately 12 Mbytes per second; the failure-oblivious version compresses files at a rate of approximately 1.2 Mbytes per second. We attribute the unusual (for our test suite) performance difference to the fact that gzip spends most of its time in computation rather than processing files or communicating via the network.

4.10 Discussion

We chose the programs in our study largely based on several factors: the availability of source code, the popularity of the application, the presence of memory errors as documented on vulnerability-tracking web sites such as Security Focus [18] and SecuriTeam [17], and our ability to reproduce the documented memory errors. In every case in which we were able to reproduce the memory error, failure-oblivious computing successfully eliminated the negative impact of the error — the programs were, *without exception*, able to execute through the error and continue to successfully service the needs of their users. Moreover, even after executing the code containing the error, we observed no degradation in service or incorrect behavior when the programs continued to process their normal workload.

In our view, this continued execution without memory corruption provides substantial advantages in comparison with the alternatives. The standard versions offer continued execution with memory corruption; the result is a constant stream of memory errors that, at best, crash the program and, at worst, enable attackers to execute arbitrary code. The safe versions prevent memory corruption but terminate the program at the first memory error. This premature termination often unnecessarily prevents users from using the program and enables denial of service attacks.

All of the programs in our study are part of the broad Linux desktop/laptop computing infrastructure. Conceptually, the basic structure of each computation is to accept, then process, a stream of inputs. One reason that failure-oblivious computing works so well for this class of applications is that the computations for different inputs are relatively loosely coupled. Unless one input corrupts the data structures or address space, it has a minimal effect on the next input. While it is less clear how failure-oblivious computing will work for other kinds of programs, several of our programs give some indication of how it might work out in practice. Failure-oblivious computing enables both Pine and Midnight Commander, for example, to correctly process inputs that otherwise trigger fatal memory errors — failure-oblivious computing enables the programs to correctly process inputs even though the computation associated with these inputs has memory errors. These examples provide encouraging evidence that failure-oblivious computing can help computations generate acceptable results even in the presence of otherwise fatal memory errors.

The error checks in failure-oblivious programs have the potential to significantly degrade the performance [40, 53, 49]. But for all of the programs in our test suite except gzip, this potential degradation is irrelevant, either because the programs provide perfectly acceptable interactive response in spite of the checks or because the execution times are dominated by I/O in the form of file accesses or network communication.

5. RELATED WORK

We discuss related work in the areas of memory-safe implementations of unsafe languages, memory-safe languages, traditional error recovery, and data structure repair.

5.1 Safe-C Compilers

Our work builds directly on previous research into implementing memory-safe versions of C [25, 53, 47, 39, 49, 40]. As described in Section 3, our implementation uses techniques originally developed by Jones and Kelly [40], then significantly refined by Ruwase and Lam [49] (in fact, our implementation is derived directly from Ruwase and Lam's CRED compiler). It would be perfectly feasible, however, to implement failure-oblivious computing using any safe-C compiler, specifically by modifying the compiler to discard unsafe writes and manufacture an appropriate value stream for unsafe reads. Building on Ruwase and Lam's implementation enabled us to apply failure-oblivious computing directly to legacy programs without modification (Yong and Horwitz's implementation also has this property); some other implementations require source code changes [29, 42].

One of the key issues for safe-C compilers is the overhead of performing the safety checks. Researchers have developed language design and static analysis techniques that reduce this overhead [36, 26]; the reported results tend to indicate a slowdown factor of usually less than two, with some occasional outliers of a factor of eight or more [53, 49]. The acceptability of this slowdown depends on how the program will be used. This overhead does not perceptibly degrade the response times of the interactive programs in our test suite. It also does not affect the performance of the I/O bound programs. It is visible only for gzip (the one compute-bound program in our test suite).

5.2 Safe Languages

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.

5.3 Traditional Error Recovery

The traditional error recovery mechanism is to reboot the system, with repair applied during the reboot if necessary to bring the system back up successfully [35]. Mechanisms such as fast reboots [50], checkpointing [44, 45], and partial system restarts [28] can improve the performance of the reboot process. Hardware redundancy is the standard solution for increased availability.

Our techniques differ from these standard mechanisms in that they are designed to keep the system operating through errors instead of rebooting the system to restart it from a new, clean state. The advantages include better availability because of the elimination of down time and the elimination of vulnerabilities to persistent errors — restarting Pine as described in Section 4.2, for example, does not enable the user to read mail if the mail file still contains a problematic mail message. Rebooting, on the other hand, may help ensure that the system stays more closely within the anticipated operating envelope.

5.4 Data Structure Repair

Data structure repair, applied either manually [46, 38] or automatically [31], has some conceptual similarity with failure-oblivious computing in that it is designed to enable the program to execute successfully through errors while preserving key data structure consistency constraints. Failure-oblivious computing differs in that 1) its goal is to prevent corruption rather than recovering from corruption and 2) it relies on the normal, uncorrupted operation of the program to preserve the consistency constraints rather than actively enforcing these constraints.

5.5 Static Analysis and Program Annotations

A combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of memory errors. CSSV uses programmer annotations to support an analysis that can statically find all buffer-overrun errors in C programs [32]. Fahndrich and Leino present an extended type system that enables the compiler to statically verify the absence of null pointer dereferences in Java programs [34]. Xi presents a type system that ensures the absence of array bounds errors [52]. 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 and the possibility of conservatively rejecting safe programs).

Researchers have also developed unsound, incomplete analyses that heuristically identify potential errors; the advantage is that such approaches typically require no annotations and scale better to larger programs. Examples of such analyses include Wagner [51], Engler [33], and Prefix [27].

5.6 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 [30] and StackShield [21] 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 [37]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [43].

A key difference between these techniques and failure-oblivious computing 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.

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.

7. REFERENCES

[1] Apache HTTP Server exploit. http://securityfocus.com/bid/8911/discussion/.
[2] CERT/CC. Advisories 2002. http://www.cert.org/advisories .
[3] CNN Report on Code Red. http://www.cnn.com/2001/TECH/internet/08/08/code.red.II/.
[4] ELM. http://www.instinct.org/elm/.
[5] Gzip exploit. http://www.securityfocus.com/bid/3712.
[6] Gzip website. http://www.gzip.org.
[7] Midnight Commander exploit. http://www.securityfocus.com/bid/8658/discussion/.
[8] Midnight Commander website. http://www.ibiblio.org/mc/.
[9] Mutt exploit. http://www.securiteam.com/unixfocus/5FP0T0U9FU.html.
[10] Mutt website. http://www.mutt.org.
[11] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.
[12] Pine exploit. http://www.securityfocus.com/bid/6120/discussion.
[13] Pine website. http://www.washington.edu/pine/.
[14] Samba exploit. http://www.securityfocus.com/bid/7294.
[15] Samba exploit. http://www.netric.org/exploits.htm.
[16] Samba website. http://www.samba.org.
[17] SecuriTeam website. http://www.securiteam.com.
[18] Security Focus website. http://www.securityfocus.com.
[19] Sendmail exploit. http://www.securityfocus.com/bid/7230/discussion/.
[20] Sendmail website. www.sendmail.org.
[21] Stackshield. http://www.angelfire.com/sk/stackshield.
[22] WsMp3 exploit. http://www.securiteam.com/windowsntfocus/6B0071F6AY.html.
[23] WsMp3 website. http://wsmp3.sourceforge.net/.
[24] C. S. Ananian and M. Rinard. Data Size Optimizations for Java Programs. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003.
[25] 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.
[26] 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.
[27] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, 2000.
[34] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.
[35] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[36] R. Gupta. Optimizing array bounds checks using flow analysis. In *ACM Letters on Programming Languages and Systems, 2(1-4):135-150*, March 1993.
[37] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
[38] 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.
[39] 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.
[40] 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.
[41] P. Kelly, R. Eckstein, and D. Collier-Brown. *Using Samba, 2nd Edition*. O'Reilly, 2003.
[42] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference Proceedings*, 1983.
[43] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of 11th USENIX Security Symposium*, August 2002.
[44] 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.
[45] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.
[46] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.
[47] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.
[48] V. S. Pai, P. Druschel, and W. Zwanenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference, General Track*, 1999.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] 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.