SAFC: Selective Awareness of Failures in C

Jason Rickwald, Eric Firestone, Jeremy Seeba
Dept of Computer Science
California Polytechnic University, San Luis Obispo
{jrickwal, efiresto, jseeba}@calpoly.edu

Abstract

There are a number of programming languages commonly used by software developers, C being one of them, that enforce weak or no type checking and allow access to any location in a process's address space. These features make the language flexible and programs written in it fast, but those programs are also more vulnerable to programmer errors and attacks from malicious outsiders. The most prevalent such attack is the buffer overflow attack. Previous solutions for managing, deterring, or removing the risk of a buffer overflow attack from programs have been inadequate, inflexible, or inappropriate in all situations. This paper briefly discusses some of these previous techniques, focusing mainly on the solutions that add bounds checking into C to remove the possibility of buffer overflow vulnerabilities. The SAFC - Selective Awareness of Failures solution is then presented, which builds on bounds checking in C[13] as well as on the failure-oblivious[12] concept of Rinard, et. al. The SAFC (pronounced "Safe C") compiler, gives developers failure-oblivious bounds checking in their programs by default, but also allows them to specify error functions that can be called in the event of a memory reference error (out of bounds read or write). Additionally, with a provided header file, the SAFC specific functionality can be ignored in non-SAFC enabled compilers. The advantages to such a flexible error-handling solution are discussed in depth.

1. Introduction

"Unsafe" languages, particularly C and C++, are to blame for a great deal of security flaws in software today. For this paper the term "unsafe," when applied to a programming language, means that it provides some mechanism that allows for accidental or intentional (malicious) reading of or writing to memory locations in a way that could expose sensitive data or abnormally affect a running process. One of the most common attacks on software, the buffer overflow, is directly related to the fact that C provides only weak type checking and allows for arbitrary pointer creation, each of which make it possible to read and write freely to any memory location in a process's address space.

Sections 2 and 3 further discusses buffer overflow attacks and methods that have been developed to try to better cope with memory-related security issues. Of particular interest is how bounds checking, and pointer validity checks in general, are added to C. This paper also discusses failure-oblivious bounds checking versus the standard abort-on-failure bounds checking, as it is the default behavior of our solution. In failure-oblivious bounds checking, writes to bad memory locations are ignored and reads from bad memory locations are provided with a made-up value. There are some alternatives to this scheme which will also be discussed, but the general idea is that the program does not abort on a memory error.

Section 4 describes our particular enhancement to bounds checking and failure-obliviousness. This enhancement, dubbed SAFC (or, Selective Awareness of Failures in C), adds failure-oblivious bounds checking to C by default. It also provides a means for programmers to opt-in to receive notifications about memory reference errors, giving more flexibility than just abort-on-failure or failure-oblivious approaches.

Sections 5 and 6 evaluate SAFC, particularly its performance and usability compared to existing solutions. Section 7 goes over future work.

2. Background

SAFC is a flexible bounds checking solution for programs written in C. It is designed to guard against most programmer memory errors and all buffer overflow attacks, which will be described in this section.

2.1 Programmer Errors

No programmer is perfect, and human-introduced errors account for a large portion of incorrect behavior in programs. [4] and [7] show that even well-established program such as the Apache web server, or well-established operating systems such as Linux or OpenBSD can contain programmer errors, even after numerous revisions. If such errors were made more noticeable to the programmer, such as by a bounds-checking compiler, they would be more likely to be fixed, and therefore not produce incorrect behavior or leave security vulnerabilities. The larger problem which is addressed by memory checking, however, is malicious attacks, such as the buffer overflow attack.

2.1 Buffer Overflow Attacks

Buffer overflow attacks account for the majority of security holes according to the United States Computer Emergency Readiness Team (CERT), with more than 57% of the 2003 security holes being related to a buffer overflow vulnerability[3, 13]. The typical goal of a buffer overflow attack is to take control of a running program that has some degree of privilege – either it is running at some higher privilege level or it is on a machine that the attacker cannot normally access[6]. Two steps must be taken before this goal may be achieved. First, the attacker must ensure that useful code is in the privileged program's address space. Next, the attacker must get the privileged program to jump to that code.

"Buffer overflow" refers to the usual way of accomplishing these two steps. Programs that receive input need to place the input data somewhere in memory. A buffer of some fixed size is allocated in the program's address space and the input is read into that buffer. If the buffer is not an adequate size for the input data and the programmer did not put checks into the code, then the data will overflow the buffer and overwrite whatever memory was beyond the end of the buffer. This vulnerability can cause accidental failures due to errors in the program or it may be exploited by an attacker.

The first step towards executing a successful buffer overflow attack, ensuring that there is some useful code in the program's address space, may be accomplished by simply using existing code in the target program. The attacker would only need to know the location of this code in the program's address space, which may be easy to determine using utilities like nm. If there isn't code to do the desired work, then the attacker may inject code into the buffer.

The second step, causing the program to jump to the desired code, can be done in a number of ways. This is normally where overflowing the buffer comes into play – the attacker is going to overwrite some adjacent memory location. If the buffer is on the stack, then the attacker can overwrite the return address for the current stack frame with the address of the code to execute. When the currently executing function returns, the program jumps to the attacker's code. This is commonly referred to as a "stack smash." Function pointers can also be overwritten by overflowing buffers anywhere in memory. When the program attempts to call the function using the pointer, it will instead execute the attacker's code. Lastly, an attacker can overwrite `longjmp` buffers. C contains `setjmp` and `longjmp`, which can be used for simple checkpoint/rollback operations. These commands use a buffer in memory. An attacker can overflow an input buffer and overwrite the value of the `longjmp` buffer so that when `longjmp` is called it jumps to the attacker's code.

Finally, it should be mentioned that not all buffer overflow attacks try to affect the target program by executing malicious code. Some buffer overflow attacks will simply try to modify the target program's normal behavior in undesirable ways by overwriting non-executable variables adjacent to the buffer. For example, an attacker may overflow an input buffer of a web application to overwrite the username associated with his session. The attacker could then have access to the given user account.

3. Preventing Buffer Overflow Attacks

This section describes existing techniques for dealing with buffer overflow attacks, including previous work in adding bounds checks to C. It also summarizes previous work in failure-oblivious computing, and the reasons why it may be more advantageous than aborting on failures for many applications.

3.1 Using a "Safe" Language

If unsafe languages are a problem, then start using "safe" languages. Such languages will typically use protections such as running the process in a virtual machine and/or interpreting the program in order to catch invalid memory accesses, thereby preventing buffer overflows. Java uses these protections, as well as using references instead of pointers. Pointers can be manipulated to point anywhere in memory, whereas references are locked to a specific object in memory. Safe languages might also enforce strong type checking or do bounds checking on array accesses (both are true of Java as well).

So why don't we just leave it at "don't use unsafe languages?" One reason is that, despite tremendous improvements to the performance of programs written in unsafe languages (like the improved performance of JIT for Java), they still typically run slower than their unsafe counterparts. A small decrease in performance may not be an issue for most programs, but it may matter for some (like servers).

There is also a very large amount of code written in unsafe languages that is infeasible to abandon or rewrite in safe languages due to time and resource restraints. In contrast to this expensive conversion, SAFC requires only a recompile of existing code, with optional modifications for increased functionality.

3.2 Checking Code

If a common mistake in programs is to allocate a buffer but not check that the input data fits in the buffer, then one solution might be to just look for such mistakes in the code and correct them. Examining code for errors or even using the test/fix method of error detection is probably the most common way that programmers try to make their code "safe." This is probably because it is the most natural way for people to do error checking. This method would be analogous to a writer proof reading his or her own document. Carrying the analogy further, the writer may then give the document to an editor for review. In the programming world, this would compare to a programmer submitting code to a code review by peers in his or her own organization or to outside auditing companies. Also, the writer may utilize spelling and grammar checkers built into his word processor. Likewise, there are tools available for the static analysis of programs to detect errors and vulnerabilities and, in some cases, take steps to deal with these problems.

Checking code for errors and vulnerabilities has the advantage that it does not adversely affect the final performance of the running program. Unfortunately, there can be no guarantee made that the checked code is, indeed, free of all errors and vulnerabilities. Buffer overflow vulnerabilities can be very subtle and may go unnoticed by many sets of eyes. Static analysis tools, too, cannot point out all problems with a program. The semantics of C make it very hard to lay out exact rules of what is allowable. Therefore, more general heuristics are often used to find errors and vulnerabilities, sometimes leading to false-positives and undetected errors.

3.3 Stack Protection

Dealing with all buffer overflow vulnerabilities in C can be an enormous task. It may be easier to focus on a smaller subclass of the vulnerability. This idea has found success in StackGuard, a compiler technique for detecting the most common form of buffer overflow attack – the stack smash. StackGuard protects against stack smash attacks by placing a known, "canary", value on the stack between the local variables and the return address. At runtime, before returning from a function, the canary value is checked to make sure it is correct. It will likely be incorrect if it was overwritten by a buffer overflow attack.

StackGuard cannot guarantee invulnerability from stack smash attacks as there are a discrete number of values that can be used as the canary value – enough tries may strike upon the correct one. However, it does make the program less vulnerable to the most common buffer overflow attack with a small performance hit.

The idea of using canary values to check for overwrites can also be extended to protect against other buffer overflow attacks, as it is in PointGuard. The idea is to place canary values next to other code pointers, like function pointers and `longjmp` buffers, and checking them before dereferencing the code pointer. This has the disadvantage of possibly causing incompatibility with existing programs and libraries as the canary values will change the size of data structures.

Canary values can also be taken further to protect other variables, not just code pointers. If we place a canary value next to every variable and check the canary before using the variable we can protect against even buffer overflow attacks that don't affect code jumps. However, checking canary values for all variables can degrade performance and memory usage. The authors of PointGuard, therefore, allow the programmer to specify which variables should get canary values. Unfortunately, this solution now has the programmer making non-portable modifications to the code and it has the programmer deciding which variables are “important” enough to get canary values (again exposing the program to human mistakes). It should also be noted again that these canary values only deter attacks. With enough determination and luck an attacker could find the correct canary value.

3.4 Memory Reference Checks

The most robust solution to a problem is to remove the root cause of that problem. With buffer overflows the problem is that a programming language does not check that data fits into its associated memory space, so a solution to the problem is to add in mechanisms that check memory references to insure they are within their bounds. There have been a number of prior projects to add array bounds checking and, more generally, memory reference validity checks to C.

The idea behind these checks is actually fairly simple. For each pointer into memory chunk of addresses is associated with valid accesses. This “chunk” is represented as a base address and an extent. When we dereference the pointer we check to see if the value of the pointer variable is between the base and extent for the pointer variable. This concept has been implemented in a variety of ways, each with its own advantages and drawbacks.

One such implementation is CCured [10]. CCured transforms programs written in C by adding memory safety. The first step that it takes is to perform static analysis on the program. Essentially, this static analysis tries to enforce strong typing on the program. The parts of the program that fail to adhere to the strong typing rules are deemed unsafe and are modified for runtime checking. The reason for this first step is to limit the performance hit incurred from runtime checks. If the tracking of a pointer shows that it is never modified, casted, or used in pointer arithmetic, then there is no reason to do time consuming runtime checks on it.

CCured's approach to runtime bounds checking is rather unique. For most unsafe pointers the representation of the pointer is actually modified depending on how the pointer is classified during the static analysis phase. The new representation contains information like type, base, and extent. Placing this information right with the pointer creates a “wide pointer” on which checks can be performed quickly. The downside to this approach is that it limits compatibility with precompiled libraries that use the standard pointer representation [5].

This difficulty is managed in a couple of ways. The primary way is to create wrapper functions to act as the interface between the CCured program and the linked in functions. These functions contain special-purpose code for modifying pointer representations as they are passed through. CCured comes with many such wrapper functions already implemented for standard libraries, but the programmer may still need to write his own wrappers for other precompiled libraries used by the program. Another approach that is used less often is to store the pointer metadata in a separate structure. This approach further degrades performance during checking, and is therefore only used when complex structures must be passed between a CCured program and a precompiled library, avoiding the need to copy such structures between representations.

Another tool, called CRED[13], uses the approach of storing metadata separate from the pointer as its common approach. This is done because the primary objective of the authors was to create a tool that could be used on any C program without needing to modify the code. A runtime table structure is used to map pointers to their associated static, heap, and stack objects. These objects are generally represented as their bases and extents in memory. The CRED compiler adds runtime code to the C programs that it is compiling for adding these objects to the table at load-time (for static variables), when functions are called (for stack variables), and when heap space is allocated (requiring modified versions of `malloc`, `realloc`, etc. be linked in). Steps are also taken to make sure that new pointers returned by precompiled libraries, which don't add object information to the table, don't cause the program to fail.

CRED has the advantage that it works with a large number of programs without needing to modify the program, and it works effortlessly with precompiled libraries. However, it performs checks on all pointers, with each check being made up of rather slow lookups and comparisons. This can result in a sizable slowdown – greater than 30% in some cases. CRED may benefit from first doing static analysis to determine what pointers are unsafe, as in CCured, but it doesn't appear that work was put into investigating this.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    int array[5];
    int var;
    int i;

    for (i=0; i<=5; i++) {
        array[i] = 1;
    }

    var = 2;

    for (i=0; i<5; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }

    printf("var=%d\n", var);

    return 0;
}
```

Figure 1: A C program with an inconsequential error

Another tool, presented in *Protecting C Programs From Attacks Via Invalid Pointer Dereferences*, also uses static analysis to determine what pointers are safe and unsafe. The program is then modified so that, at runtime, the unsafe pointers are tracked. To avoid the issues that come up when trying to store bases and extents for each object in memory, this tool actually uses a single bitmap to keep track of all locations in memory that are "appropriate" and "inappropriate." This bitmap start out with all memory marked as inappropriate. When space in memory is allocated to the tracked pointers, that space gets marked as appropriate. Checking a pointer dereference consists of doing a quick lookup of that location in the bitmap. If it is appropriate, then it is allowed.

For the most part, this tool works a lot like CRED, but using a single bitmap makes pointer validity checks before dereference much faster. The disadvantage to this approach is that all "appropriate" areas in memory apply to all tracked variables. This means that it is still possible for inappropriate use of one pointer to modify the object of another pointer, but it is less likely (with the likelihood going up as more variables get marked as "tracked"). This tool also doesn't handle precompiled libraries as well as CRED and deals with them using wrapper functions as CCured does.

3.5 Failure-Oblivious Computing

Each of the aforementioned runtime tools may look for memory reference errors in different ways, but they each deal with errors in the same way – they abort the process. This course of action is the most natural because it is the safest. If the program is doing something bad, then terminate it before it can do any more bad. However, this may not be the ideal action in all cases. Some programs may want to terminate on an error, but only after performing some clean-up. Also, there are some errors in programs that are very minor and may hardly affect the execution of the program (if at all). Consider the example C program in Figure 1. When compiled and run this program runs just as expected – it prints out 1's for each `array` element and a 2 for the `var`. Yet it contains a very common programming error. The while loop that assigns the 1's to the `array` elements writes one element past end of the array because of the use of the less-than-equals instead of less-than. It doesn't fail because the `var` memory is located after the array, and writing there is allowed. There is no problem because `var` does not get assigned a value until after the loop executes. A bounds-checked version of this program would abort even though there is no real problem. A complex program may contain many such inconsequential errors.

Aborting on a bad memory reference may also be unnecessary for applications such as web servers and other client-server programs. Servers often use separate threads to handle each client session. If one thread is attacked by a malicious client, it would be better to terminate that thread instead of terminating the entire server process. This was the main reasoning behind the "failure-oblivious" bounds-checking compiler presented in *Enhancing Server Availability and Security Through Failure-Oblivious Computing[12]*. This work started with the CRED compiler and is therefore susceptible to the same slowdown. The authors modified CRED so that memory errors do not abort. Instead, writes to invalid memory locations are ignored and values are manufactured for reads from invalid memory locations. Some care was taken with the

```
#include "safc.h"
void *error_func(size_t size, const char *msg, const char *filename, int line, void *pointer, const object * obj);
set_error_func(error_func);
```

Figure 2: API for SAFC including header for compatibility

manufactured read values so that programs would hopefully not get stuck – random values were often returned, but most manufactured values were 1's and 0's (as these are the most common loop termination values).

Programs compiled with the failure-oblivious compiler will not abort due to inconsequential errors, or other programming errors that would normally cause problems for that matter. Consider Figure 1 again, but imagine the consequences of placing the assignment to `var` before the first loop. After compiling the program with a normal compiler we would see unexpected behavior. Compiling it with the failure-oblivious compiler would not show the unexpected behavior, because the write to `array[5]` is ignored.

The *Failure-Oblivious* authors showed that the failure-oblivious approach was useful in many situations; particularly with servers. Input data formed for an attack on a server is changed so that it is simply bad input, which is handled in normal ways by the server's user input checks. Even if the input cannot be handled in a normal way, the attack was not allowed to corrupt critical structures like the call stack. Also, execution paths for servers handling client requests tend to be short, so that bad data will not propagate far and should not affect other user sessions.

There is a variation to the approach of ignoring bad writes, which is also mentioned in the original paper, where writes to bad memory locations are not ignored, but are stored in a data structure. Later, if there is a bad read from the same memory location, then, rather than manufacturing a value, the value is taken out of the data structure. This approach takes more work, time, and space, but it is useful for programs that have come to rely on the presence of particular memory errors.

4. SAFC: Selective Awareness of Failures in C

SAFC is an extension of the failure-oblivious approach that adds a degree of flexibility by allowing an error handler to be defined in certain sections of code. Through the use of opt-in error handling, programs can be written with or without knowledge of SAFC yet still benefit from the failure-oblivious aspects that have been added in. Programs that have defined error handlers and take advantage of what SAFC has to offer will be given useful debugging information into their custom error handler that would not be available otherwise and have the decision about what to do with that information. Our solution focuses on usability and usefulness of SAFC and therefore uses a simple runtime API for programmers and provides useful information for their custom error handlers.

4.1 Our Solution

Our solution combines the fault-tolerance that comes from failure-oblivious computing with fine-grain control through the use of the runtime API. Our solution provides functions to handle different types of errors, such as array index out of bounds on a read from a memset() to a pointer location. By allowing different types of errors to be defined by different error functions, certain types of errors can be left as failure-oblivious while others that are either critical to handle or be notified of can be handled by writing an error handling function as part of the program. By providing this solution, a program can handle any type of memory error as is appropriate, such as printing the error to the screen, aborting the program, or even emailing the developer with information about the error.

4.2 Our Implementation

A focus of our implementation is on making a usable solution. For this reason, instead of handling individual cases with different error functions, we use the very simple API shown in figure 2; a single error function that handles all types of failures.

```c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "safc.h"

void * debug_error_handler(size_t size, char * msg,
        char * filename, int line, void * pointer)
{
   /* Send mail to a specified recipient on each error */
   char subject[300];
   char recipient[] = "efiresto@calpoly.edu";
   char message[1000];
   int n, fd[2];
   pid_t pid;

   printf("In error handler\n");

   sprintf(subject, "\"Memory Bounds Error: %s\"", msg);

   sprintf(message, "Memory boundary error occurred:\n"
                    "Error: %s\n"
                    "File: %s\n"
                    "Line: %d\n"
                    "Pointer: 0x%x\n",
                    msg, filename, line, pointer);

   if (pipe(fd) < 0)
      /* Error occurred, just do FOB */
      return (void *)new_fob_pointer(size);
   if ( (pid = fork()) < 0)
      return (void *)new_fob_pointer(size);
   else if (pid > 0) {
      close(fd[0]); /* Close read end */
      /* Write the message to the pipe */
      write(fd[1], message, strlen(message));

      close(fd[1]); /* Close write end */
      waitpid(pid, NULL, 0);

      if (size == 0) {
         return NULL;
      } else if (size == -1) {
         return (void *)new_fob_pointer(sizeof(int));
      } else {
         return (void *)new_fob_pointer(size);
      }
   } else {
      close(fd[1]); /* close write end */
      if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
         /* Error occurred, just do FOB */
         return (void *)new_fob_pointer(size);
      close(fd[0]);
      execl("/bin/mail", "mail", "-s", subject,
         recipient, NULL);
   }
}
```

```c
int main(int argc, char ** argv)
{
   int a[3];
   int b[3];

   set_error_func(debug_error_handler);

   /* Reference */
   a[3] = 5;
   printf("%d\n", a[3]);

   /* Memory Functions */
   memset(&a[3], 9, sizeof(int));
   memcpy(&a[3], &b[2], sizeof(int));
   mempcpy(&a[3], &a[2], sizeof(int));
   memmove(&a[3], &a[2], sizeof(int));
   memchr(&a[3], 'c', 2);
   bcopy(&a[3], &a[1], sizeof(int)*2);
   bzero(&a[3], sizeof(int)*2);
   printf("memcmp returned %d\n",
      memcmp(&a[3], &a[2], sizeof(int)));
   printf("bcmp returned %d\n",
      bcmp(&a[3], &a[2], sizeof(int)));

   return 1;

}
```

Figure 3: A sample SAFC application that e-mails the developer on each invalid memory reference.

In an error handling function, the user is provided with a large amount of information relevant to the invalid memory reference. Specifically, the user is provided with the size of the memory chunk being referenced. This is important as the pointer returned from the error handling function will be used in place of the invalid address, and so the returned pointer must point to a sufficiently large area of allocated memory to avoid an additional reference error. Currently, write operations are made consistent with failure oblivious computing and are skipped silently. This means that the pointer returned by an error handler is not used. To indicate this to the programmer, the size parameter is given as zero. See the future work section for possible expansions on this implementation.

Additional information is given about the error for reporting purposes. This includes a message describing the type of error which has occurred, such as whether it was an array out of bounds error, or an increment on an invalid pointer. The file name and line number where the error occurred are also provided, as is a copy of the invalid pointer. Finally, a constant reference to the actual in-memory bounds checking object is provided. This can provide additional information if desired, such as the base and extent addresses of the original allocation, as well as the file name and line number of where the allocation was made.

Because a user may want to have a program remain failure oblivious to the user, but add additional error handling, we provide a "pass-through" method for obtaining valid failure oblivious replacement pointers. The function, new_fob_pointer(size_t), returns a void pointer to an area of memory of a size at least as large as the parameter specifies.

As with the failure oblivious project, our work is derived from the CRED source code. To implement our modifications, we started by adding simple failure oblivious operation to the bounds checking compiler, and then exchanged this automatic failure oblivious behavior with the ability to call an optional handler. Unlike the failure oblivious paper, our failure oblivious implementation does not necessarily return ideal values to allow for continued operation, although values are based upon information provided by the failure oblivious paper. When using the default failure oblivious behavior of SAFC, or when using the new_fob_pointer() convenience function, there is a 25% chance that the returned pointer will point to an address containing a "0", a 25% chance that it points to an address containing a "1", and a 50% chance that the address points to a random number.

To enable the ability to compile SAFC code under non-SAFC compatible compilers, we provide a header file for use in SAFC programs. By accessing the error handler "set" function and failure oblivious convenience function through a provided header file, the calls to these functions will be turned to NO-OPs in compilers which don't recognize SAFC. This allows programs to compile correctly with no adverse side-effects. The strength of our solution and implementation is seen in how SAFC can be useful in the programming and debugging process.

4.3 SAFC Application

We decided to write a program to show a useful application of SAFC in the debugging process. When making a very complex program, there is bound to be some memory errors, whether a simple less than equals where a less that sign should be or a complex chain of references that are not handled correctly. Figure 3 shows a program that simulates memory errors and has an error handler that sends a mail message to the developer giving the type of error, line number of the error, file that contained the error, etc.

This model could allow developers to release a debug version of their software with error reporting enabled that is completely transparent to the user in sections of code that they want to debug and run in the default failure-oblivious mode in other sections. In fact, because an error function should be able to reasonably recover from the failure, the program will not even crash but continue to run as if it was working correctly. When the developer receives the messages, they will know what line and file an error came from and should be off to a good start to fixing the error without the software users even knowing that there is a problem.

While this shows one possible use of SAFC in the debugging process any number of possible uses could be devised because of the extreme flexibility in error handling used by SAFC.

5 Performance

Because our work is based on the CRED bounds-checking compiler, we expect to incur similar performance penalties. The bounds checking mechanism is essentially unchanged, and any performance differences would only occur if and when an error handling function is executed. In error-free programs, performance should be identical; in programs with errors, the additional overhead will depend upon the complexity of the user defined error handler and the frequency of errors.

Based upon the experiments in [3], we expect a performance decrement of about 30% for most programs, with a significantly higher penalty for certain applications such as ssh or enscript. As with CRED's original compiler, applications compiled with the SAFC compiler would show performance benefits from disabling string checking (and therefore reducing the protection provided), and from static program analysis such as that in CCured [10].

6. Usability

The major advantage of the SAFC compiler over other bounds-checking and failure oblivious solutions is its flexibility. It was a key design goal that this flexibility did not come at the cost of usability. This section contrasts the features and usability of the SAFC compiler with other similar solutions.

6.1 No Handler

SAFC is only beneficial if it provides improvements over no memory handling at all. The default case of no handler does provide a number of options, namely speed because no memory checks are done, and ease of implementation, since nothing special needs to be done.

SAFC provides a large number of benefits over no memory checking. These benefits are largely described above, with the additional security and the addition of failure obliviousness being some of the most significant.

6.2 SIGSEGV Handler

Similar to a SAFC solution, implementing a SIGSEGV handler allows the user to display a more informative, customized error message. It also gives the programmer the potential to recover from the error, though not easily.

The use of a SIGSEGV handler provides two noticeable benefits over using a SAFC compiler. First, using such a handler is fast. Because no additional memory checks are done (there is only a different function called when a segmentation fault occurs), there is no performance decrease. A second benefit is that the ability to add a SIGSEGV handler is widely available. Unlike SAFC which requires a special compiler, the ability to handle SIGSEGV signals is mandated by the POSIX standard [11], and therefore available in a large number of existing C compilers.

A SIGSEGV handler does provide a few of the features of a SAFC compiler, though in a more limited fashion. As in a SAFC compiled program, a special error handling function is used, which can allow the programmer to terminate with a more informative message for the user. Such a technique is employed by GCC itself. Unlike a SAFC error handler, a SIGSEGV signal handler is not provided with any additional information, such as the filename and line number of where variable was allocated, or where the error occurred. It also lacks the detailed error message describing the error which is provided to a SAFC error handler.

A SIGSEGV handler could also potentially recover from a segmentation fault, however this is difficult. Upon returning from a SIGSEGV handler, the program counter will return to the instruction that caused the signal to be generated and attempt to re-execute that instruction. This will create an infinite loop unless cumbersome stack or return address manipulations are made.

A third similarity in usability is that both a SIGSEGV and a SAFC error handler require that the user implement a custom handler to produce specialized behavior. The methods for setting the custom handler are also similarly simplistic – a function pointer must be passed to a single "set" function. To remove the handler the function is called again with some "default" value. Unlike the SIGSEGV method, SAFC provides the additional benefit of failure obliviousness when no user error handler is defined.

A segmentation fault handler also lacks many of the advantages of a SAFC error handler as well. Unlike a SAFC handler, a SIGSEGV handler is called only when a segmentation fault occurs, which as mentioned, does not include referencing valid memory from the wrong memory structure. Because the default POSIX behavior is to terminate on a SIGSEGV signal, an application that generates a segmentation fault would likely have quit anyway, and therefore adding a handler does not provide any additional security benefit. Furthermore, it does not help catch programmer memory errors which do not generate a SIGSEGV signal.

6.3 Bounds Checking

Bounds checking compilers such as CCured [10] or CRED [13] provide many of the benefits of the SAFC compiler; however they provide few benefits which a SAFC compiler does not include.

Bounds checking compilers share the security of a SAFC compiler as invalid memory references cause program abortions (and therefore do not allow malicious attacks to proceed). As discussed, their memory checking also causes them to share the performance deficit of the SAFC compiler.

Any benefits unique to bounds checking compilers lie in the fact that by default errors in a bounds-checked program will cause a program to terminate, whereas a SAFC program is failure oblivious by default. As a design decision we default to failure obliviousness; in programs where this is not desired, bounds-checked only program would have a small advantage in that they would require no extra implementation. To change to bounds-checking type behavior in a SAFC program requires simply writing an error handler function that aborts and to set this function as the handler early on in execution.

A bounds checking only solution does lack a number of benefits present in a SAFC compiler. The most important is the ability to do any sort of failure obliviousness. Additionally, a bounds checked program cannot create custom error notifications, and is forced to use either the default handling or no handling at all for the entire program execution.

6.4 Purify

Purify[9] is a commercially available application that performs similarly to bounds checking. A run of an application is done while linked to the Purify runtime library and any memory errors are listed in a final output log.

Purify differs slightly in its usability from current bounds checking GCC solutions. The biggest difference is a single final report which lists all memory errors and mismatches as opposed to bounds checking's abort-on-failure approach. Unfortunately, this report is cumbersome and somewhat difficult to follow, so it is debatable as to which reporting method is superior. Additionally, Purify will catch references to uninitialized variables.

Purify does not, however fully encompass all of the checking capabilities present in bounds checking [1]. Purify will only work reliably on heap (malloc'ed) variables, and does not work reliably through certain pointer casts for structs.

Because of the cost involved with purchasing the software, and its limited checking capabilities, using the SAFC compiler provides a number of usability improvements over Purify.

6.5 Failure Oblivious Computing

Failure obliviousness implements behavior similar to SAFC, but lacking a few usability features. Like SAFC, failure oblivious computing allows for continued execution even after a memory error, and all memory references are checked, making it safe.

A key advantage of SAFC over failure obliviousness is its ability to provide a custom error handler on error. This means that the programmer is notified on errors, and can therefore create cumulative logs or other enhanced error handling techniques.

A second advantage is that a SAFC error handler allows the programmer to tailor the memory address which is referenced instead of the invalid one. By returning a specific pointer from the error handler, the programmer can specify what memory will be read or written instead of the invalid memory which caused the error handler to be invoked.

A final advantage is that the error handler behavior can be changed for different areas of the application code. Unlike failure obliviousness which inherits its on/off ability from its CRED roots, SAFC can provide an unlimited number of different error handler functions for different areas of the code, with each handling an error in different ways.

6.6 SAFC

The SAFC compiler aims to improve on the usability flaws of the previously mentioned solutions. This paper did not focus on improving the performance hit introduced by the base bounds checking compilers, and so this flaw remains common.

Because security is a key feature of any of these solutions, the SAFC compiler carries over the memory checking of many of the mentioned solutions. Unlike these solutions, it does not mandate what action will be taken if the bounds are violated. An extra degree of flexibility is added which allows the user to specify this action if desired using a custom function. Additionally, the helpful information which is present in the CRED bounds checking compiler is passed to this function so that it may be utilized by the programmer.

The improved user experience of failure

oblivious computing is also incorporated into the SAFC compiler. To provide this functionality with the least hassle, failure obliviousness is the default behavior when no error handler is defined. Additionally , the convenience function new_fob_pointer() provides the user access to a predefined area of memory which may be used to implement failure obliviousness in addition to other error handling actions.

The method for setting an error handler mirrors that of setting a signal handler. We feel that this is a simple, elegant, and easily understood method for doing so. It requires understanding only a single function, and is very easily understood if signal handlers are already understood.

There are additional benefits of SAFC which are not simply carried over from previous solutions. As mentioned previously, SAFC allows for customized error handling, as well as safe compilation under non-SAFC compilers.

7 Future Work

In our current implementation, standard library calls that are linked in instead of being part of GCC like gets() or getc() are not checked for memory errors. In the current state, functions like memset() work because they have been rewritten to take advantage of bound checking. By creating a safe version of different libraries that can be linked in or wrapping these calls, SAFC would then be able to check other functions that are out of its scope now.

Another area that we would like to see extended is passing the error function an enum which indicates the type of error that occurred. Currently, to determine the type of error which occurred programmatically, the error handling function must do string parsing on the error message passed in. This is cumbersome, and it is relatively trivial, but useful, addition to pass an enum value along with each error message. This would also expand the flexibility of handling for write operations. Currently, write operations are usually aborted. The user is notified of the error in the error handler, and operation will continue in a failure oblivious fashion, but the pointer returned by the user is not actually written to. The programmer may want the write to occur to his returned pointer in some cases. An example application of this would be to implement the expanded failure oblivious functionality mentioned in the original paper, in which case invalid writes are made in additional memory and reads from that same invalid location are redirected to the additional memory. If an enum were provided which clearly indicated if the error occurred in a write or a read (along with additional information), then this type of implementation would be feasibly simple to implement for the programmer.

An exception to our “skip writes” policy is in variable assignments. Due to the complexity of handling this case, and time limitations, when an assignment to an invalid reference is encountered, the write proceeds to the user returned pointer. Instead of the way assignments are currently handled, through allocating memory and letting the write happen, our implementation should be changed to be consistent with other write operations.

Finally, being able to pick between the default behaviors of failure-oblivious computing versus normal bounds checking, that exits when an error occurs, should be a compiler option. Some programs probably should not continue to run on manufactured values or tolerate any memory errors so failure-oblivious computing would be the wrong default behavior where bounds-checking would be a much better choice.

8 Conclusions

SAFC is an evolution of existing memory check solutions in the C programming language. It provides the safety of bounds checking, the user experience of failure obliviousness, and the flexible ability to be changed as with a signal handler. As demonstrated, SAFC allows for very flexible error handling, such as an e-mail notification to the developer on error, which are not possible with any existing solution.

Unfortunately SAFC suffers from the same performance limitations of other bounds checking solutions, however like other solutions, it will benefit from enhancements such as static analysis and code optimizations.

The benefits of SAFC over other bounds checking solutions, particularly the ability to create a custom error handler and to create different handling methods for different areas of the code, are significant. Additionally, there are no significant benefits which existing solutions have that SAFC does not. This combination makes SAFC a viable replacement in existing bounds checking and failure oblivious implementations. Also, as demonstrated, there are many additional uses for SAFC which are not possible with existing memory checking solutions.

References

[1] Bounds Checking GCC – User Guide <http://www.lrde.epita.fr/~akim/compil/doc/bounds-checking.html>

[2] Brugge, H.. SourceForge.Net: Boundschecking. SourceForge.Net. The Stanford SUIF Compiler Group. 11 May 2006 <http://sourceforge.net/projects/boundschecking>.

[3] CERT/CC Advisories 2002. CERT. 4 June 2006 <http://www.cert.org/advisories>.

[4] Chen, H., Dean, D., Wagner, D. Model Checking One Million Lines of Code. *NDSS*, The Internet Society, Dec 2004

[5] Condit, J., et. al.. CCured in the Real World. *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003. 11 May 2006. <http://www.cs.berkeley.edu/~smcpeak/papers/ccured_pldi03.pdf>

[6] Cowan, C., et. al. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of DARPA Information Survivability Conference and Exposition*, Jan. 2000. 11 May 2006 <http://dx.doi.org/10.1109/DISCEX.2000.821514>.

[7] Engler, D., et. al. Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code. *ACM SIGOPS Operating System Review.* ACM Press, 2001.

[8] Gnu Gcc Team. GCC Online Documentation - GNU Project - Free Software Foundation (FSF). *GCC Home Page*. 22 Apr. 2006. 11 May 2006 <http://gcc.gnu.org/onlinedocs/>.

[9] Hastings, R., Joyce, B. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at http://www.rational.com/support/techpapers/fast_detection/.

[10] Necula, G. C., McPeak, S., Weimer, W. CCured: Type-Safe Retrofitting of Legacy Code. *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*. ACM Press, 2002

[11] The POSIX standard. http://www.posix.com/posix.html

[12] Rinard, M, et. al. Enhancing Server Availability and Security Through Failure-Oblivious Computing. *OSDI* 2004. 11 May 2006 <http://www.usenix.org/events/osdi04/tech/full_papers/rinard/rinard.pdf>.

[13] Ruwase, O., Lam, M. S.. A Practical Dynamic Buffer Overflow Detector. *11th Annual Network and Distributed System Security Symposium*, Feb. 2004. 11 May 2006 <http://suif.stanford.edu/papers/tunji04.pdf>.

[14] Yong, S. H., Horwitz, S. Protecting C Programs From Attacks Via Invalid Pointer Dereferences. *9th European Software Engineering Conference Held Jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003. 11 May 2006 <http://pag.csail.mit.edu/reading-group/yong03protecting.pdf>.