Submitter: Robert C. Seacord
Submission Date: February 3, 2019
Group: ISO/IEC JTC 1/SC 22/WG14
Document No.: N2336
Title: Bounds-checking Interfaces: Field Experience and Future Directions

Summary

Annex K of C11, Bounds-checking interfaces, introduced a set of new, optional functions into the standard C library with the goal of mitigating the security implications of a subset of buffer overflows in existing code. The bounds-checked interfaces originated as an ISO/IEC technical report in 2007 before being incorporated in C11 as the optional but normative Annex K. Field experience with the bounds-checked interfaces has been hampered by a lack of adoption by compiler implementations, despite these interfaces being available for over a decade. This lack of adoption resulted largely from unfounded criticism of the API as well as actual flaws. As a result, the international standardization working group for the C programming language is evenly divided between repairing and eliminating Annex K for the next major release of the C language (C2X). This paper examines the arguments both for and against the bounds-checked interfaces, evaluates possible solutions for actual problems with Annex K, and propounds the repair and retention of Annex K for C2X.

1. Introduction

C11 Annex K, "Bounds-checking interfaces", introduced a set of new, optional functions into the standard C library to mitigate certain classes of security vulnerabilities. Annex K defines copying functions, concatenation functions, file operations, file access functions, formatted input/output functions, general utilities, search functions, date and time functions, and extended multibyte and wide character utilities. The design of the interfaces has an extensive history that dates back to security pushes by Microsoft in the 1990s.

The annex originated as ISO/IEC TR 24731-1 — Extensions to the C library — Part 1: Bounds-checking interfaces, published in 2007 [3]. The technical report was incorporated into C11 [16] with only minimal changes.

TR 24731-1 is the result of a four-year effort by ISO/IEC JTC1/SC22/WG14 (the international standardization working group for the programming language C). This effort started with the original Proposal for Technical Report on C Standard Library Security submitted on February 24, 2003 [1], and culminated by a successful publication of the technical report in 2007.

The Rationale for TR 24731 Extensions to the C Library Part I: Bounds-checking interfaces [4] explains the goals for Annex K. The most important goal was to mitigate certain classes of security vulnerabilities including (1) buffer overflows, (2) formatted output vulnerabilities, and (3) default protections associated with program-created files. For (1), the goal was that the bounds-checking interface~~s~~ should not store data outside of its intended target. Whenever data is stored to an array, a bounds is used to verify that other storage is not being modified. Additional goals include not producing unterminated strings, not unexpectedly truncating strings, and preserving the null-terminated string datatype.

In addition to these goals, there were also usability goals to make it easier to adopt and use the bounds-checking interfaces with existing code. For example, the bounds-checked interfaces were designed to only require local edits to programs, allowing the replacement of existing functions with "only a line or two of code".

WG14 is currently considering the future of Annex K. WG14 is evenly split between eliminating and repairing the annex [10]. This paper incorporates field feedback as well as providing an assessment of the current state of Annex K and recommendations for future directions.

Much of the stigma associated with the bounds-checked interfaces is the result of poor guidance and practices associated with the adoption and application of these interfaces. The bounds-checked interfaces are best suited for developing new code that might receive invalid inputs. Legacy functions are preferred when developing performance-intensive code that can be proven correct by other means. Existing code that is not being actively improved (e.g., refactored or modernized) should continue to use the legacy APIs. As with any functions, the incorrect adoption and use of the bounds-checked interfaces, can have an unsatisfactory outcome.

This paper examines the utility of the bounds-checking interfaces compared with the legacy functions and examines use and misuse cases. Section 2 describes the usability of Annex K functions and mechanisms; Section 3 discusses specific security concerns; Section 4 describes the use cases and misuse cases for adopting the bounds-checked interfaces; Section 5 contrasts

the performance of these functions with the performance of the legacy functions; Section 6 enumerates existing implementations; Section 7 discusses the future of Annex K; and Section 8 summarizes the paper.

2. Usability

Usability is critically important in the design of a secure interface. It is important to remember that, with few exceptions (such as the `gets` function—deprecated in C99 and eliminated from C11), there are no inherently insecure standard functions. Some functions, however, are more usable than other functions meaning that they are less prone to misuse and resulting defects and vulnerabilities. Exact data on defect rates is difficult to obtain, although there is wide industry consensus that many of the legacy functions defined in the C standard library [25] are prone to misuse [26]. The remainder of this section contrasts the bounds-checking interfaces mechanisms and functions with legacy capabilities.

2.1. Interface Design

To ease their adoption in existing code bases, the names and signatures of the APIs were deliberately chosen to be close to those of the corresponding standard functions. These APIs have a `_s` suffix, commonly take an additional argument, and return a value of type `errno_t` rather than a pointer to the destination buffer. For example, the standard function `strcpy` declared as follows:

```
char *strcpy(
  char * restrict s1,
  const char * restrict s2
);
```

corresponds to the Annex K `strcpy_s` function whose declaration is:

```
errno_t strcpy_s(
  char * restrict s1,
  rsize_t s1max,
  const char * restrict s2
);
```

Syntactic differences are in **bold** font. The additional `s1max` argument specifies the size in characters of the destination buffer pointed to by `s1`. The function return type is `errno_t`, which is a `typedef` for `int`. The function returns zero to indicate success and a non-zero value on error.

2.2. Array Size

While superficially similar, `strcpy_s` has different behavior than `strcpy`. The additional `s1max` argument was added to require developers to specify the size of the destination array. Because of array decay in the C and C++ languages, it is not otherwise possible for the invoked function to determine the size of the object referenced by a pointer to an array. Historically, this has been the cause of numerous vulnerabilities in C and C++ programs [26]. The idea of adding these extra arguments is not new to Annex K, but originated with the OpenBSD functions `strlcpy` and `strlcat` [14].

When replacing an invocation of `strcpy` with a call to `strcpy_s` it is necessary to determine the size of the destination array. If the array is declared in the scope of the invocation, the size can be statically determined by the `sizeof` operator. If the storage is dynamically allocated or declared in a different scope, it may be advantageous to save the result of the size computation in a new variable of type `size_t` or `rsize_t`.

O'Donell and Sebor [24] identify specific programmer errors using the bounds-checking interfaces observed in code bases but without any specific statistics as to how common these errors are. The first of these errors is that the programmer incorrectly specifies the size of the source array instead of the destination array:

```
void func(void) {
  char source[] = "...";
  char dest[N];
  ...
  strcpy_s(dest, sizeof source, source);

}
```

While such mistakes are possible, they are easily identifiable by static analysis. GCC, for example, detects this bug in uses of the standard functions.

The use of the `strcpy_s` function here is no less secure than the use of the legacy function. If a buffer overflow could result from this code, it would also result from a similar call to the `strcpy` function. On the other hand, if the source array were smaller than the destination array, no buffer overflow would occur and the error would be detected at runtime.

Another possible error can occur when the size of the destination buffer is not readily available at the call site to the function (for example, the destination is not an array). A programmer may use the wrong computation (or reuse an inappropriate existing computation) to determine the size of the destination buffer. A typical mistake of this sort is to pass the result of `strlen(dest)` as the size of the destination buffer, for example:

```
void func(const char *dest) {
  char src[N];
  ...
  strcpy_s(dest, strlen(dest), src);
  ...
}
```

In the absence of a previous error, `dest` must be a pointer to a string (meaning that a null character is present before the bound of the array referenced by `dest`). Provided that `dest` refers to a valid string, the size passed to the function is too small (by at least one). This code may fail to copy a string for which adequate storage was available, but it is unlikely to result in a buffer overflow.

2.3. Error Handling

Although `errno` itself is considered outdated, the concept of a set of `errno` values to indicate failure conditions is used by many Annex K functions. These functions return a value of type `errno_t` that would be the value to which the functions set `errno`, if the functions did set `errno`. Although ISO/IEC 9899:1999 [25] defines only three different specific values for `errno`, other standards (such as POSIX) and conventions define many more. Annex K defines no new values for `errno`.

Because of the usefulness that the set of `errno` values represents, Annex K defines a `typedef`, `errno_t`, to represent this set of values. As stated earlier, the type of `errno_t` is required to be `int`, which is also the type of `errno` itself.

Returning a value of type to `errno_t` to indicate the status of the returning function is a best practice in C language programming because the language lacks a general exception handling mechanism.

The legacy `strcpy` function returns a pointer to the destination string. Programs typically ignore this return value except in cases where it is chained with other API calls in a single expression.

```
strcat(strcpy(d, a), b);
```

In these cases, there is an implicit assumption that the call to `strcpy` succeeds. As a consequence, secure coding standards such as ISO/IEC TS 17961 [12] and The CERT C Coding Standard [15] have no requirement to diagnose a failure to check the return value from `strcpy`.

Invoking the `strcpy` or `strcat` functions can easily result in a buffer overflow if the programmer has not ensured adequate storage has been allocated to store the resulting string.

In contrast, the `strcpy_s` Annex K function returns a value of type `errno_t`. Checking for potential errors requires compromise as function nesting is easily implemented when the return value is used to indicate status.

Consequently, the above code must be rewritten as two separate statements:

```
strcpy_s(d, sizeof d, a);
strcat_s(d, sizeof d, b);
```

And because these functions may fail and return an error, the return value needs to be tested and appropriately handled:

```
if (strcpy_s(d, sizeof d, a))
```

```
  return -1;
if (strcat_s(d, sizeof d, b))
  return -1;
```

The migration from functions whose return values are typically ignored to functions that need to be checked to determine if an error occurred should be aided by static analysis that generates a warning when a programmer neglects to check a return value. This analysis can be implemented using the `nodiscard` attribute supported by C++ and proposed for C2X [9]. The `nodiscard` attribute allows an API developer to indicate that ignoring the results of a function call is a programmer error.

Annex K supports an additional error handling feature. When a function detects an error (such as invalid arguments or not enough room in an output buffer) a special runtime-constraint handler function is called. The constraint handler might print a message to `stderr` and/or abort the program. The programmer has control of the handler function called via the `set_constraint_handler_s` function, and can have the handler simply return if desired. If the handler returns, the function that identified the *runtime-constraint violation* and invoked the handler indicates a failure to its caller using its return value.

Runtime-constraints are violations of the runtime requirements of a function that the implementation must detect and diagnose by a call to a handler and, if the handler returns, by a failure indicator returned to the caller of the failed function call.

The implementation is required to enforce runtime-constraints. Typically, this is performed by bounds-checked interface functions checking the conditions immediately upon entry, or as they perform their tasks and gathers sufficient information to determine if a runtime-constraint has been violated. The runtime-constraints of the bounds-checked interfaces are conditions that would otherwise be undefined behavior for C Standard functions.

An implementation has a default constraint handler that is used if no calls to the `set_constraint_handler_s` function have been made. The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort. The default behavior of the handler is implementation-defined to allow implementations to provide reasonable behavior by default. This allows compilers customarily used to implement safety-critical systems, for example, to not abort by default. Implementation-defined behavior can be eliminated by invoking the `set_constraint_handler_s` function before invoking any bounds-checked interfaces or using any mechanism that invokes a runtime-constraint handler.

The `abort_handler_s` and `ignore_handler_s` functions represent two common cases and are provided for convenience. The implementation default handler need not be either of these handlers.

In general, applications are responsible for setting the error handling policy, while libraries are not. A library which sets an error handling policy, by setting the `abort_handler_s` for example, may now be unusable in safety-critical applications.

Applications that establish a policy of not returning from errors (e.g., by setting the `abort_handler_s` or similar non-returning handler) are free to ignore the return value from Annex K functions. This significantly reduces the amount of code that must be written to test the results of calls to the bounds-checked interfaces and handle errors.

An application policy of not returning on error *can* be appropriate for security-critical systems where errors may indicate an attack and that continued execution may result in a security compromise. In safety-critical systems (such as an airplane that must provide robust, continuous operations in the presence of unexpected inputs) forcing a system to abruptly terminate may be the goal of an attack. The system architecture must decide on a consistent, system-wide approach to error handling that makes sense for the particular application.

Setting policy in library or middleware code is usually avoided because it limits the types of applications in which this code can be used. This means that library code that wishes to avoid policy decisions must be able to work with application code that installs the `ignore_handler_s` or similar handler that returns and requires the library code to evaluate the return values from the function for errors.

O'Donell and Sebor [24] are not the first to point out that adding code to handle errors increases the size and complexity of code. *Provably correct code* can and should use legacy functions to avoid this additional overhead, for example:

```
size_t size = strlen(source) + 1;
dest = (char *)malloc(size);
if (dest) {
  memcpy(dest, source, size);
```

```
} else {
  /* handle error   */
  ...
}
```

However, as most programmers do not write provably correct code it is frequently beneficial to use the bounds-checked-interfaces to detect potential errors.

2.4. Single Pass

String operations are typically implemented as single pass algorithms. For example, when implementing the `strcpy_s` function, a naive algorithm might first determine the length of the string before copying. This requires two passes over the array; the first to locate the null character and the second to actually copy the character data. Requiring two passes over the array significantly impacts the performance of a successful copy on long strings.

To perform the string copy in a single pass, a typical `strcpy_s` function implementation retrieves a character from the source string and copies it to the destination array until the string has been copied or the destination array is full. If the entire string cannot be copied, the `strcpy_s` function sets the first byte of the destination array to the null character, creating an empty string. The remaining bytes in the destination array will likely contain the partially copied source string.

The `strcpy` function does not detect undefined behaviors at runtime.

There are potential security concerns with the `strcpy_s` function. If the original source string contained sensitive information, the programmer must take care not to leak it, for example, by using a `memcpy` operation instead of a `strcpy` operation.

Creating an empty string on failure can also have security implications. Consider the following function that creates a path name by copying a safe root directory to a fixed length array followed by a user-specified directory and file name. The function then allows the user to perform a security sensitive operation on the file, provided it is within the safe root directory:

```
void create_path(char * dir, char * fn) {
  char pn[2048];
  strcpy_s(pn, sizeof pn, "/safe_dir/");
  strcat_s(pn, sizeof pn, dir);
  strcat_s(pn, sizeof pn, fn);
  security_sensitive_op(pn);
}
```

An attacker might exploit this function by passing a very long string for `dir` causing the first `strcat_s` operation to fail. If the runtime constraint handler returns, the string referenced by `pn` will be an empty string. The next call to `strcat_s` can allow the attacker to specify any pathname and escape the secure directory:

```
char longstr[2048] = {0};
memset(longstr, 'a', sizeof(buffer) - 1);
create_path(
  longstr,
  "/unsafe_dir/sensitive_file"
);
```

This call invokes `security_sensitive_op` with the pathname `"/unsafe_dir/sensitive_file"`.

The defect in the `create_path` function is the failure to test for and handle errors from the bounds-checked interface functions and to continue execution without considering the downstream consequences of these failures.

2.5. memcpy

The `memcpy` function can be used instead of `strcpy` to copy strings under certain conditions. Both functions are defined in Section 7.24 of the C Standard, “String handling `<string.h>`”. The `memcpy` function can be used to copy memory, but also to copy strings provided the size of the destination array is equal to or larger than the `size` argument to `memcpy`, the source

array contains a null character before the bound, and that the string length is at least one less than the `size` so that the resulting string will be properly null-terminated.

There are a variety of recommendations as to when to use `strcpy` vs. `memcpy`. The most conservative of these is to use `strcpy` or `strcpy_s` when copying a string, and `memcpy` or `memcpy_s` when copying memory. Another common recommendation is to use `memcpy` to copy strings when the prerequisite conditions described in the previous paragraph are met. This recommendation is based on the theory that `memcpy` is faster than `strcpy`.

Most `memcpy` and `strcpy` implementations are designed to efficiently handle large amounts of data. This often requires additional startup overhead such as determining alignment, setting up SIMD, cache management, and so forth. That makes these implementations slow for copying small amounts of data. Consequently, a `memcpy` implementation optimized for large amounts of data may be significantly slower than a `strcpy` implementation that was optimized for small amounts of data (when copying small amounts of data).

2.6. String Copy with Truncation

The `strncpy` function has had a problematic history. The `strncpy` function was designed for inserting text into the middle of strings and was then repurposed as a secure replacement for `strcpy` although it is not. The most significant problem with the `strncpy` function is that it does not properly null-terminate the resulting string if the source string is longer than the number of characters being copied. In other words, if there is no null character in the first n characters of the source array, the destination array will not be null-terminated.

The `strncpy_s` function solves this problem by setting the `dest[n]` to a null character if no null character is copied from the source array. This also means this function is useless for copying into the middle of a string. Of course, the `strncpy` function could also insert a null byte if the source string is shorter than n.

The `strncpy` function appends null characters to the copy in the destination array when the source string is shorter than n characters, until n characters have been written. A common source of errors when converting existing code to the bounds-checked version is to assume that the `strncpy_s` function zeroes out the destination buffer past the first null character in a similar manner to `strncpy`. This functionality was deliberately not carried over to `strncpy_s` because the zeroing out was criticized as an inefficiency. The difference in behavior has led to unintended information disclosure vulnerabilities in networking code where the previous contents of the buffer beyond the first null character were sent to the client. For example, naively replacing the call to `strncpy` in the following function with `strncpy_s` leaves the bytes past the first null character in the destination unchanged when `strlen(in)` is smaller than `out_len`:

```
void secure_copy_buffer (char *out,
  const char *in, size_t out_len) {
    strncpy (out, in, out_len);
    // ...
}
```

This function could potentially leak sensitive data, although it seems unlikely that a developer concerned with leaking information would rely on `strncpy_s` to clear sensitive information.

```
void secure_copy_buffer (
  char *out, const char *in, size_t out_len
) {
  strncpy_s(out, , out_len, in, out_len);
  // ...
}
```

The decision to leave the bytes past the first null character in the destination unchanged was the consensus view of WG14. If WG14 ever reconsiders this decision, changing the behavior of the `strncpy_s` function to zero out the destination buffer past the first null character should have no impact on existing program behavior outside the possibility of some limited performance regressions. The current specification is not unreasonable. In cases where the developer needs to ensure that the original contents of the buffer are overwritten, the call to the `strncpy_s` function should be preceded by a call to `memset_s`.

A further issue with the `strncpy_s` function was documented by defect report 468 [17]. As originally specified by C11, the `strncpy_s` function was allowed to clobber characters in the destination array past the terminating null character to allow for

efficient, single-pass implementations. However, because the latitude granted went beyond what was required and because the possibility of unspecified values resulting from a successful operation raises security concerns about information leakage, the standard was modified to only allow the destination array to take unspecified values when `strncpy_s` returns a non-zero value.

2.7. The strerror Function

The legacy function `strerror` is not thread safe and has a serious usability problem [2]. It requires the caller to supply a fixed buffer for the result, but there is no way to determine how large this buffer needs to be. The only way to make this work in general is to supply an initial buffer, check for overflow, reallocate, and try again until succeeding or all available memory is used. For example:

```
size_t buflen;
char *buf;
buflen = 100;
while (0 != (buf = malloc(buflen))) {
  if (0 == (strerror(errno, buf, buflen)) )
    break;
  free(buf); buflen++;
}
```

Annex K solves both these problems by introducing two functions. The `strerror_s` function that can be used to avoid data races and the `strerrorlen_s` function that calculates the length of the (untruncated) locale-specific message string that the `strerror_s` function maps to `errnum`.

The resulting solution could be implemented as follows:

```
size_t bufle = strerrorlen_s(errno) + 1;
char * buf = malloc(buflen);
if (buf) strerror_s(errno, buf, buflen);
```

The `strerror_s` function is used to avoid data races. In a single threaded application, the `strerror` function could be used to eliminate unnecessary checks because the size of the destination array is known to be of sufficient size.

While using the `malloc` function has a performance impact and is also not permitted in safety-critical applications that conform to MISRA [13], applications are not required to invoke `malloc`. If simple truncation is acceptable for the application, then the `strerror_s` function is sufficient because it will always result in a null terminated string and will gracefully truncate the error string if there is a sufficiently large buffer.

3. Security

This section examines aspects of Annex K that affect the security of systems that use it.

3.1. Pointer Subterfuge

A frequent target of exploits is to overwrite the address of a function to which execution will eventually be transferred with the address of malicious code injected by an attacker (a.k.a shell code) or normal code already present in the code segment that is repurposed for malicious purposes. These attacks can be accomplished through any indirect call where the target of the indirection can be overwritten by an attacker. Examples of such targets include function pointers, the global offset table (GOT), the `.dtors` section, virtual pointers, the `at_exit` and `on_exit` functions, and the `longjmp` function.

Most implementations use a function pointer in their implementation of the `set_constraint_handler_s` function to hold the address of the currently registered handler. This introduces an additional address that can be altered by an attacker to execute malicious code. WG14 decided that the benefit of a user-settable runtime-constraint handler justified providing another function pointer that might be overwritten by an attacker.

Each of these addresses can be protected from being overwritten with the address of malicious code. One mechanism for protecting these is the `encode_pointer` and `decode_pointer` functions proposed by Plum and Bijanki [19]. These functions are similar in purpose to the `EncodePointer` and `DecodePointer` functions used by Visual Studios' C runtime libraries. Instead of storing a function pointer, the program can store an encrypted version of the pointer. An attacker would need to

brute-force the secret key used to encode the pointer to redirect the pointer to other code [11]. The Plum and Bijanki proposal was rejected by WG14 because the committee decided that it would be better for implementations to provide this functionality automatically without the need for programmer intervention and was consequently considered a quality-of-implementation issue. In the 11 years since this proposal was presented to WG14, no implementations have implemented this feature. Additionally, it is not possible for C programmers to implement their own encode/decode function in a portable manner because C does not give programmers a portable way to treat function pointers as data.

In the absence of an execution environment with protected function pointers, the addition of another attack vector is largely irrelevant because any address to which control is transferred can be overwritten, including the address of the `abort` function in the GOT table. In the case of an implementation where the address of system functions is protected, but other addresses are not, abrupt termination of the program by calling the `abort` or `_Exit` functions may prevent the execution of malicious code. The programmer must also consider the possibility that the attacker's goal *is* to force a program to terminate abruptly.

Other paths to termination, such as calling `exit` or `quick_exit`, result in execution of functions registered by calls to the `atexit` and `at_quick_exit` functions, respectively. It is possible, when running in an insecure execution environment, that the address of functions to be executed have been overwritten by an attacker to divert execution to malicious code.

The Annex K runtime-constraint mechanism is both a security benefit and weakness in that it can be used to easily implement a policy of exiting abruptly (for example, by installing the `abort_handler_s`) but can also provide an attack vector. Systematically, it makes little difference if this additional attack vector is present. An implementation that protects function pointers can protect the current runtime-constraint handler from being overwritten using the same mechanism. An implementation that does not protect function pointers will have other locations that can be overwritten by an attacker to execute malicious code.

4. Use Cases

There is a wide variety of advice when it is appropriate or inappropriate to use the bounds-checking interfaces; much of which is contradictory. This section describes some of the existing guidance for adopting the bounds-controlled interfaces.

4.1. Banned Functions

Microsoft, who initially developed these APIs, completely bans the use of *replaced* C Standard functions such as `strcpy` and `strcat` as part of the Security Development Lifecycle [Howard 2006]. Use of the Annex K functions is supported by static analysis to ensure the right buffer size values are used. Microsoft's experience is that, without static analysis, bounds-checking interfaces are easily misused resulting in many of the same issues. Similar static analysis can, of course, also improve the usability of legacy functions.

The conversion process at Microsoft was aided by modifications to the compiler to automatically migrate code to the bounds-checking interfaces if the length of the destination buffer could be determined at compile time. Automation is most effective by limiting it to changes which were guaranteed to not introduce defects. It is also worth noting that the introduction of additional runtime-constraint tests during the conversion process at Microsoft resulted in the discovery of previously undetected undefined behaviors and vulnerabilities [6].

4.2. Failure to Implement

Strict banishment of C Standard string handling and other functions is a position at one extreme of this argument; the other extreme is disallowing the use of bounds-checked interfaces functions (possibly by failing to implement them).

O'Donell and Sebor [24] argue that such changes are often unnecessary and increase the opportunities to introduce defects. This is a widely-held expert view that changes to "working code" generally increase the risk to the system because of the possibility of injecting new defects. This view has even been expressed by the safety-critical community [18] and elsewhere.

Disallowing the use of bounds-checked interfaces goes beyond this and prevents the use of these functions in new code and during repairs where they can be used advantageously. Failing to implement Annex K functionality forces programmers to use legacy functions which decades of practice has demonstrated are prone to misuse and vulnerabilities.

4.1. Selective Use

A compromise between these two extremes is to introduce Annex K functions selectively in new code or as required to mitigate existing vulnerabilities. In cases when the invoked handler does not return, this approach has the advantage that the maintainer does not require a deep understanding of the affected code. The repair can be implemented by programmers unfamiliar with the code with little chance of introducing additional defects.

In situations where the handler may return to the caller, modifications to existing code may need to be more extensive. This is particularly true of code that was not designed to handle error conditions.

Using Annex K functions in new code leads to the most straightforward use case as new code can be designed with the use of these functions in mind. For example, all three calls to string handling functions in the following code can result in an undetected buffer overflows:

```
int main(int argc, char *argv[]) {
  char name[2048];
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
  ...
}
```

Repairing this code while retaining the legacy functions requires some effort and additional overhead.

The first (obvious) problem is that `argv[1]` may be null or `argv[1]` may refer to a string and `argv[2]` may be null. These problems can be eliminated by testing `argc` and returning with an error (and optionally a usage message) if insufficient arguments were passed to the command. Assuming the solution continues to use a statically allocated buffer, the programmer needs to determine if the combined sized of the three strings can exceed the length of the buffer. The size of both `argv[1]` and `argv[2]` are unknown but can be determined by a call to `strlen` as shown:

```
int main(int argc, char *argv[]) {
  char name[2048];
  if (argc < 3) return 1;
  if (strlen(argv[1]) +
      strlen(argv[2]) +
      sizeof " = " > 2048) return 1;
  strcpy(name, argv[1]);
  strcat(name, " = ");
  strcat(name, argv[2]);
  ...
}
```

This of course, requires an additional pass to calculate the length of both argument strings. The following code eliminates the possibility of undefined behavior by allowing the Annex K functions to invoke a non-returning constraint handler if a runtime-constraint violation is detected:

```
char name[2048];
strcpy_s(name, sizeof name, argv[1]);
strcat_s(name, sizeof name, "=");
strcat_s(name, sizeof name, argv[2]);
```

Using the bounds-checked interfaces in this case means that this code is now secure and only needs to make a single pass over each command line argument. Consequently, the solution using the bounds-checked interfaces is more concise and theoretically faster. This code can also be written to work in the case when the runtime-constraint handler returns:

```
constraint_handler_t oconstraint =
  set_constraint_handler_s(
    ignore_handler_s
  );
  ...
char name[2048];
if (strcpy_s(name, sizeof name, argv[1]))
  _Exit(EXIT_FAILURE);
if (strcat_s(name, sizeof name, "="))
```

```
  _Exit(EXIT_FAILURE);
if (strcat_s(name, sizeof name, argv[2]))
  _Exit(EXIT_FAILURE);
```

Writing and testing this code is no different than developing any other code that detects and securely recovers from error conditions.

Mandating the use of these functions can result in non-optimal code in cases where there is no possibility of error conditions. Developers should be allowed to combine Annex K functions with standard string handling functions to efficiently handle these cases.

5. Performance

While the performance of bounds-checked interfaces can be compared to the performance of the replaced legacy functions, this is not a fair comparison because these functions do not do the same thing. Specifically, bounds-checked interfaces test for runtime-constraint violations including verifying that pointers are not null, that pointers do not refer to overlapping regions of memory, and that the size of the destination buffer is sufficient to hold the result of the operation. A function that perform these checks, such as `strcpy_s`, is necessarily slower than `strcpy`, which does not. The absence of these tests in `strcpy` does not mean that they do not need to be performed, although they can frequently be omitted from user code when redundant. Implementing these tests within the body of the bounds checked interfaces means that this code only needs to be implemented in one location which can lead to smaller, faster code bases.

Modern optimizing compilers provide *intrinsics* or *built-ins* as highly efficient equivalents of the traditional C library string manipulation functions and expand calls to the functions inline. Besides avoiding the overhead of a branch instruction to jump to the library implementation of the functions, the intrinsics have the important benefit of enabling other optimizations across multiple calls to the same function that are not possible otherwise.

Although the intent of the original proposal authors was that the APIs would be implemented in compilers in the form of efficient intrinsics, to date this has not happened. Until this work is completed, each call to one of the APIs incurs the overhead of a function call, and redundant tests for runtime-constraint violations are not eliminated. The following section provides performance measures for specific implementations.

6. Implementations

Despite the bounds-checked-interface specification having been around for over a decade only a handful of implementations exist, with varying degrees of completeness and conformance. This section provides a survey of known implementations and their status.

While two of the implementations below are available in portable source code form as Open Source projects, popular Open Source distributions such as BSD or Linux have not made either available to their users. At least one (GNU C Library) has repeatedly rejected proposals for inclusion citing the Austin Group review of WDTR 24731 [2]. Conversely, the Clang and GCC mailing lists have seen multiple requests for Annex K support [20]. It is unclear if (and when) these APIs will be provided by future versions of these distributions.

6.1. Microsoft Visual Studio

Microsoft Visual Studio is the prototype implementation of the bounds-checked interfaces. Unfortunately, the implementation does not conform to C11 or TR 24731-1. This is because Microsoft failed to update their implementation based on changes to the APIs that occurred during the standardization process. For example, Visual Studio does not provide the `set_constraint_handler_s` function but instead retains the previous function with similar behavior but a different and incompatible signature:

```
_invalid_parameter_handler
 _set_invalid_parameter_handler(
 _invalid_parameter_handler)
```

It also does not define the `abort_handler_s` and `ignore_handler_s` functions, the `memset_s` function (which is not part of the TR), or the `RSIZE_MAX` macro. The Microsoft implementation also does not treat overlapping source and destination sequences as runtime-constraint violations and instead has undefined behavior in such cases.

Table 1 lists the performance of `memcpy_s` vs. `memcpy` for Microsoft Visual Studio in millions of cycles. Performance was measured by Office Profiler. Both `memcpy` and `memcpy_s` were invoked from a function which was declared `__declspec(noinline)` to prevent the compiler from inlining either function, and compiled at `-O2`. If the amount of data copied is small (16 bytes or less), `memcpy` is faster. The `memcpy_s` function is faster for 32 and 64 bytes. For sizes greater than 64 bytes, the difference in performance is negligible.

Table 1: Performance of `memcpy_s` vs `memcpy` in millions of cycles.

SIZE	MEMCPY_S	MEMCPY
4	175.7	28.4
8	58.2	19.8
16	40.3	16
32	15.7	21.8
64	8.8	15.9
128	16.2	15.6
256	15.1	19.8
256	16.7	15
512	24.9	15.8
1024	26.8	17.7
2048	25	15.5
4096	26	21

The test app consumed around 1.38 billion cycles. Even with a stripped-down test executable that was not doing much else, 1.5-1.8% of the test case's run time was spent in memory copy operations. Unless an application is in a tight loop performing a large number of copies of less than 16 bytes or less, the difference in performance is insignificant. In the overall context of a binary that is doing a lot of other things, the difference in performance should be unnoticeable. In this test case, initializing the two source and destination buffers (1 page each) on startup consumed 30 million cycles. The `printf` call used to prevent the memory operations from being eliminated by dead store removal consumed 600 million cycles.

As a result of deviations from the specification, the Microsoft implementation cannot be considered conforming or portable.

6.2. Open Watcom

Starting with version 1.5, the Open Watcom compiler [23] ships with an implementation of TR 24731-1. Open Watcom Version 1.9 defines the `__STDC_LIB_EXT1__` macro to `200509L`, indicating that it conforms to the final draft of the technical report [3].

Because the final draft of TR 24731 is essentially identical to the published technical report, which is close to Annex K (although not identical because the `memset_s` function specified by the Annex does not appear in the technical report, the Open Watcom implementation can be considered a nearly conforming implementation.

6.3. Safe C Library

The Safe C Library [21] is an efficient, portable, and complete implementation of Annex K with many extensions. It also allows search for Unicode strings by adding the `wcsfc_s` and `wcsnorm_s` extensions supporting Unicode standard 11.0, with plans to update to 12.0 in May, 2019.

The performance of `memcpy_s` vs `memcpy` was measured using the `perf_memcpy_s.c` test.[1] All tests were performed with the `-march=native` and `--disable-constraint-handler` flags. The results from these tests are shown in Table 2. Additional flags used are shown in the table.

Table 2: Performance of `memcpy_s` vs `memcpy`.

COMPILER	OVERHEAD	ARCH	FLAGS
`clang-7`	5-20%	32 bit	

[1] https://github.com/rurban/safeclib/blob/master/tests/perf_memcpy_s.c

clang-4	5-20%	32 bit	
clang-3.9	87%	32 bit	
clang-3.8	86%	32 bit	
clang-3.7	84%	32 bit	
clang-3.4	89%	32 bit	
clang-3.3	88%	32 bit	
Apple/cc	87%	32 bit	
gcc-7	89%	32 bit	-Wa,-q
gcc-5	88%	32 bit	-Wa,-q
gcc-4.9	86%	32 bit	-Wa,-q
gcc-4.8	89%	32 bit	-Wa,-q
gcc-4.6	89%	32 bit	-Wa,-q
gcc-4.3	86%	32 bit	-Wa,-q
clang-7 -Ofast	-2%	64 bit	
clang	-2 - 5%	64 bit	
gcc	77%	64 bit	Wa,q

6.4. Slibc

Slibc is a complete, open source implementation of Annex K designed to be used with the GNU C library typically distributed with Linux [22]. The implementation claims to be complete and to fully conform to C11. An inspection of the implementation reveals that it is inefficient and consequently unsuitable for production use without considerable changes. However, it does provide a good reference implementation of the library. A proposal to incorporate slibc into the GNU C library was rejected by the GNU C library community in 2012.

7. Future Directions

WG14 is currently evenly divided between eliminating Annex K and repairing it [10]. There are valid arguments on both sides, although various issues complicate the removal of Annex K from the C2X standard.

7.1. memset_s

The `memset_s` function was proposed after ISO/IEC TR 24731-1 had been added to C11 as Annex K [5]. Unlike `memset`, any call to `memset_s` is evaluated strictly according to the rules of the abstract machine. This function was added to address the concern that operations meant to overwrite memory for security purposes were being removed by the compiler using *dead store removal* optimizations. This proposal was adopted by WG14 but incorporated into Annex K because it at least superficially resembled other bounds checked interfaces. As defined, the function makes use of the runtime-constraint mechanism. Consequently, if Annex K were eliminated but this function retained it would need to be altered or the runtime-constraint mechanism would need to be repaired and retained as well.

7.2. Annex L Analyzability

Annex L specifies optional behavior to aid in the analyzability of C programs. Annex L states that “If the program performs a trap, the implementation is permitted to invoke a runtime-constraint handler.” The runtime-constraint mechanism is defined by Annex K, so its removal would require revisiting Annex L.

7.3. Thread-local Storage

There is general consensus among WG14 experts that the runtime-constraint mechanism is incorrectly specified for multi-threaded programs [7].

The `set_constraint_handler_s` function sets a process-wide runtime-constraint handler. A frequent use case is that the handler is set appropriately for some sequence of calls after which the original handler is restored. However, the process-wide handler is shared among all threads in a program. This can lead to changes in one thread having inadvertent consequences in a separate thread. Implementations can be allowed to make the current handler state in the `set_constraint_handler_s` function thread-local if they desire to do so, as long as the handler state is inherited from the current thread at the time of creation [8]. Overall, this is an easily remedied problem and not in itself sufficient reason to abandon Annex K.

7.4. Visual Studio Alignment

The C Standards committee requires existing implementations before considering standardization. It is possible that WG14 went too far in reinventing the interfaces from the original Microsoft proposal [1] with the unintended consequence of causing the Microsoft implementation to be nonconforming. A solution is to revert the Annex K specification to more closely resemble the existing Visual Studio implementation. This would result in the Visual Studio implementation conforming to the standard but potentially make Annex K less likely to be adopted elsewhere. Visual Studio solves the problem of a single, process-wide runtime-constraint handler by adding the `set_thread_local_invalid_parameter_handler` function to define thread-specific handlers.

7.5. Forward References

There are a number of forward references throughout the main body of the standard to Annex K functions which resolve issues with the legacy APIs.

The `strtok` function has a forward reference to the `strtok_s` function that can be used instead to avoid data races.

The `strerror` function has a forward reference to the `strerror_s` function that can also be used to avoid data races.

These functions, along with the `strerrorlen_s` function, are strong candidates to become non-optional parts of the standard. Additional Annex K functions also make strong candidates although a thorough examination of industry practice is required to make a complete list.

8. Summary

The Annex K Bounds-checking interfaces have largely functioned as anticipated by WG14. The introduction of Annex K in C11 was not intended to be an impetus for changing large amounts of working production code and guidance in the proper use of these functions is poor or lacking. These interfaces are best suited for developing new code that might receive invalid inputs. Legacy functions are appropriate when developing performance-intensive code that can be proven correct by other means. Existing code that is not being actively improved should continue to use the legacy APIs.

Experience with these functions has been somewhat limited by the availability of implementations. The largest body of experience in implementing the bounds-checked interfaces comes from Microsoft, whose experience has been largely positive. Microsoft claims, for example, replacing legacy functions with bounds-checked interfaces in their Office product resulted in defect reductions. Microsoft determined that the application of static analysis is necessary to aid in the correct and secure use of the bounds-checked interfaces.

One obvious problem area requiring repair is the use of the runtime-constraint handlers in multithreaded environments. There are multiple proposals to resolve this issue, the simplest being the proposal by Florian Weimer [8] to allow the effective constraint handler to be either the thread-local constraint handler or the global constraint handler, depending on which approach is chosen by the implementation.

The field experience report by O'Donell and Sebor [24] is critically deficient in a number of significant aspects which negates the usefulness of their findings. First, the report does not contain any frequency data involving the number of modifications made to the code base and defect insertion rates. Consequently, their report only shows the kinds of errors which can be made and fails to provide any useful inferences as to the overall usability of these interfaces. A second deficiency is that the report focuses on a small number of Annex K functions from sections K.3.7.1, “Copying functions” and K.3.7.2, “Concatenation functions” and ignores the majority of functions defined in the other sections of Annex K. The report recommends the elimination of the entirety of Annex K, whereas only a small number of these functions were critically examined.

Eliminating Annex K from future major revisions of the C Standard as a result of an easily resolved defect would be a major overreaction and is uncharacteristic of the C Standards Committee which took decades to deprecate then remove the `gets` function. Eliminating Annex K without first providing an alternative solution to the security problems inherent in the use of the standard C handling functions would be irresponsible and would largely roll back C language security to C99.

Acknowledgements

Thanks to my colleagues who reviewed or otherwise contributed to this paper: Ollie Whitehouse, Jennifer Fernick, Thomas Plum, Aaron Ballman, Jeff Dileo, David LeBlanc, Michael Howard, Dan Jump, Florian Weimer, Reini Urban, and Martin Sebor. Thanks to Simon Harraghy for superb technical editing.

References

[1] N997 Proposal for Technical Report on C Standard Library Security, February 24, 2003. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n997.pdf

[2] N1106 — Austin Group Review of ISO/IEC WDTR 24731 Specification for Secure C Library Functions, March 2005 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1106.txt

[3] ISO/IEC TR 24731-1:2007 Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces http://www.iso.org/iso/catalogue_detail.htm?csnumber=38841

[4] N1173 — Rationale for TR 24731 Extensions to the C Library Part I: Bounds-checking interfaces http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1173.pdf

[5] David Svoboda. `memset_s` to clear memory, without fear of removal April, 2009. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1381.pdf

[6] Michael Howard and Steve Lipner. 2006. The Security Development Lifecycle. Microsoft Press, Redmond, WA, USA.

[7] N1866. Martin Sebor. Thread safety of `set_constraint_handler_s`. September 19, 2014. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1866.htm

[8] N2225. Florian Weimer. Multi-threading behavior of strtok, getenv, set_constraint_handler_s. 2018-03-26 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2225.htm

[9] Aaron Ballman. N2267 The `nodiscard` attribute. July, 2018. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2267.pdf

[10] Rajan Bhakta, "Draft Minutes for 15-18 October, 2018 MEETING OF ISO/IEC JTC 1/SC 22/WG 14 AND INCITS PL22.11". http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2307.pdf

[11] Moritz Jodeit. Look Mom, I don't use Shellcode: Browser Exploitation Case Study for Internet Explorer 11. Blue Frost Security. https://labs.bluefrostsecurity.de/files/Look_Mom_I_Dont_Use_Shellcode-WP.pdf

[12] ISO/IEC TS 17961:2013/Cor 1:2016. Information Technology—Programming Languages, Their Environments and System Software Interfaces—C Secure Coding Rules. Geneva, Switzerland: ISO, 2016.

[13] MISRA (Motor Industry Software Reliability Association). MISRA C: 2012 Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1, ISBN 978-1-906400-11-8 (PDF), March 2013.

[14] Todd C. Miller and Theo de Raadt. 1999. `strlcpy` and `strlcat`: consistent, safe, string copy and concatenation. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 41-41.

[15] Seacord, R.: The CERT C Coding Standard, Second Edition: 98 Rules for Developing Safe, Reliable, and Secure Systems (Addison-Wesley, 2014).

[16] ISO/IEC. Programming Languages—C, 3rd ed (ISO/IEC 9899:2011). Geneva, Switzerland: ISO, 2011.

[17] Martin Sebor. N1872 `strncpy_s` clobbers buffer past `null` http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2148.htm#dr_468

[18] MISRA Compliance:2016 Achieving compliance with MISRA Coding Guidelines, April, 2016.

[19] Thomas Plum and Arjun Bijanki. N1332 Encoding and Decoding Function Pointers. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1332.pdf

[20] Implement C11 annex K? https://sourceware.org/ml/libc-help/2019-01/msg00035.html

[21] Safe C Library — A full implementation of Annex K https://github.com/rurban/safeclib/

[22] slibc https://code.google.com/archive/p/slibc/

[23] Watcom C Library Reference Version 1.8. Open Watcom. 2008. ftp://ftp.openwatcom.org/manuals/current/clib.pdf

[24] Carlos O'Donell, Martin Sebor N1967 Updated Field Experience With Annex K — Bounds Checking Interfaces. September, 2015. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm

[25] ISO/IEC. Programming Languages—C, 4th ed (ISO/IEC 9899:2018). Geneva, Switzerland: ISO, 2018.

[26] Robert C. Seacord. 2013. *Secure Coding in C and C++* (2nd ed.). Addison-Wesley Professional.