Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool

Thomas Plum
Plum Hall, Inc.
3 Waihona Box 44610
Kamuela, HI 96743 USA
+1-808-882-1255
tplum@plumhall.com

David M. Keaton
1630 30th Street #311
Boulder, CO 80301 USA
+1-303-782-1009
dmk@dmk.com

ABSTRACT

We present a set of methods ("SSCC", for "safe, secure C/C++") to eliminate buffer overflows (including wild-pointer stores) in C and C++, using a mixture of compile-time, link-time, and run-time tests, plus some design-time restrictions. A prototype implementation indicates that run-time overhead is much smaller than previous methods. The SSCC methods do not require changes to existing data layouts or object-code representation.

The SSCC methods are applicable to applications written for the ISO/IEC 9899:1999 ("C99") standard [5] and the 14882:2003 (C++) standard [6] (herein, the "Standards"), as well as most commercially-popular extensions to those standards, and the earlier ISO/IEC 9899:1990 ("C90") standard (now essentially out-of-print).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *assertion checkers, class invariants, reliability*;
D.3.4 [**Programming Languages**]: Processors – *code generation, compilers, optimization;*

General Terms

Design, Economics, Reliability, Security, Standardization, Languages, Verification.

Keywords

Static analysis, dynamic analysis, buffer overflow, reliability, code generation, compilers, optimization.

1. INTRODUCTION

Buffer overflows (in C and in C++) are the underlying cause of many vulnerabilities, accounting for up to 50% of vulnerabilities reported by CERT/CC[1]. Completely preventing these weaknesses without sacrificing efficiency would contribute positively to every software security assurance (SSA) approach for C and C++. According to Robert Seacord [11], "vulnerability reports continue to grow at an alarming rate ... To address the growing number of both vulnerabilities and incidents, it is increasingly apparent that the problem must be attacked at the source by working to prevent the introduction of software vulnerabilities during software development and ongoing maintenance."

Ruwase and Lam summarized the situation: "A considerable amount of work has been performed on mitigating the buffer overflow problem using either static analysis or dynamic analysis."[10] However, in SSCC we attack the buffer-overflow problem using static analysis for issues that can be resolved at compile-time (and link-time), plus some amount of dynamic analysis using highly-optimized code sequences, for issues that can only be resolved at run-time. Furthermore, certain design-time restrictions can help eliminate buffer overflows, as described later in this paper.

Modern compilers for C and C++ already perform significant static analysis to understand program semantics for optimizations, especially on vector and super-scalar hardware. Furthermore, in well-written programs the array-bounds information is already maintained in variables defined by the programmer. SSCC provides a method for the compiler to track that bounds information and verify (at compile-time, link-time, or run-time) that fetch-and-store operations are proper.

Whenever possible, we have adopted terminology and concepts that would be reasonably familiar to programmers and compiler implementers. C and C++ are rife with concepts which are intuitive to the programmer but complicated to represent in abstract mathematical logic — and vice-versa. The programmer who understands the concepts behind SSCC will be better prepared to achieve the full safety/security goals of SSCC while minimizing run-time overhead. (We do use the mathematical notation for half-open interval [Lo,Toofar) in contrast to the closed interval [Lo,Hi].)

The SSCC methods generate fatal diagnostic messages in any case where buffer overflow cannot be definitively prevented. However, the SSCC methods do not impose "style rules" or portability considerations upon the compilation. Any particular tool can enhance the basic SSCC methods.

SSCC also applies to the production of software for embedded systems, but there are slightly different design criteria in that arena. This paper primarily addresses the application of the SSCC methods to hosted systems, such as applications written for Linux, or Windows, or the Mac OS.

SSATTM'05, 11/7-11/8/05, Long Beach, CA, USA.
(c) 2005 ACM 1-59593-307-7/05/11

2. BACKGROUND

We use these definitions for some fundamental terms:

Bound of an array – the number of elements in the array.

Lo of an array – the address of the first element of the array.

Hi of an array – the address of the last element of the array.

Toofar of an array – the address of the "one-too-far" element of the array, the element just past the ***Hi*** element.

Target-size (or ***Tsize***) of an array – same as sizeof(array)

Once a pointer is associated with an object, the same terms are defined for that pointer. For an array object, the Tsize of a pointer into that object is the total number of bytes in the array that is accessed by the pointer; i.e. the bound of the array times the number of bytes in (or "sizeof") each element. Furthermore, the same definitions are applied to pointers to non-array objects, consistent with the equivalence between a non-array and an array whose Bound is 1. The terms Lo, Hi, and Toofar can also be applied to integer subscript values when the context allows.

We use "variable" to designate named objects including sub-objects declared with member names. We use "state of an object" exclusively to refer to run-time state, and "attribute of a variable" to designate the compile-time understanding of that state. One simple example is the Nul attribute. On the two flow-control arcs from "if (p!=0)", p has the Nul attribute on the "false" outcome arc and the not-Nul (Nnul) attribute on the "true" outcome arc. Other attributes used in SSCC are as follows: Indir (Indirectable), Ni (Not-indirectable), Qi (Maybe-indirectable, either Nul or Indirectable), Lo (at Lo limit value), Hi (at Hi limit value), Toofar (at Toofar limit value), Ntl (Not-too-low, greater-than-or-equal-to Lo), Nth (Not-too-high, less-than-Toofar), Nullt (Null-terminated), and Unk (Unknown). These attributes are not mutually-exclusive. Besides the attributes of one variable, SSCC makes frequent use of relationships between variables, as follows.

Table 1. Relationships between variables

int n IS_BOUND_OF(p)	n provides the Bound of p
int n IS_LENGTH_OF(p)	n provides Length of p (number of elements before null-terminator)
int n IS_TSIZE_OF((p,q))	n provides Tsize of p and of q
char *p IS_HI_OF (a)	p provides the Hi of a
char *p IS_LO_OF(a)	p provides the Lo of a
char *p IS_TOOFAR_OF(a)	p provides the Toofar of a

The notation above permits representation in the opposite order, with the obvious meaning:

Table 2. Relationships between variables, opposite order

char *p BOUND_IS(n)	n provides the Bound of p.
char *p LENGTH_IS(n)	n provides the Length of p
char *p TSIZE_IS(n)	n provides the Tsize of p
char a[] LO_IS(p)	p provides the Lo of a
char a[] HI_IS(p)	p provides the Hi of a
char a[] LO_IS(p) TOOFAR_IS(q)	p provides the Lo of a, and q provides the Toofar of a

Several of these attributes can be defined using other attributes; e.g. the Bound of an array is equal to the Toofar minus the Lo. (For pointers, C/C++ arithmetic divides difference by sizeof.)

A function's returned value is an unnamed object whose attributes and relationships are often important:

Table 3. Attributes and relationships involving returned value

int n IS_BOUND_OF(return)	n provides the Bound of the function's returned pointer
int n IS_LENGTH_OF(return)	n provides the Length of the function's returned pointer
int n IS_TSIZE_OF(return)	n provides the Tsize of the function's returned pointer
char *p IS_HI_OF(return)	p provides the Hi of the function's returned pointer
char * QI(return) f() {	function f returns a Maybe-Indirectable return value

These tables suggest a representation suitable for notation in source code, but any equivalent representation will do.

Many details of the attributes and relationships used in the SSCC methods will be obvious from the Standards; here we will focus upon some details that might not be obvious. The attributes and relationships are used to express pre-conditions and post-conditions of operators and functions. Whereas some systems of static analysis require manual annotation of pre- and post-conditions, the SSCC methods are targeted at millions of lines of existing code and therefore rely only on pre- and post-conditions inferred automatically by the compiler. To emphasize the distinction, we designate these pre-conditions as "Requirements" and these post-conditions as "Guarantees".

We illustrate the basic definitions with this code snippet:

```
char a[] = "xyz";
char *p = a;
```

The Lo of `p` is the address `&a[0]` (or equivalently, the index 0), the Hi of `p` is `&a[3]` (or the index 3), and the Tsize of `p` is 4. The compiler keeps track of a relationship between the pointer `p` and the array to which it points. The relationship continues through any pointer arithmetic (including increment or decrement) operations on `p`, but is discontinued when an address of a new object is stored into `p`.

3. COMPILE-TIME VERIFICATION

For a simple example of compile-time verification, consider the following.

```
struct spec_fd_t {int m;/*…*/} spec_fd[3];
for (i = 0; i < 3; i++) {
  int limit = spec_fd[i].m; /*…*/
}
```

The Bound of `spec_fd` is 3, the Hi is 2, and the Toofar is 3. The number of iterations is less than or equal to the Bound; since the subscript variable `i` starts at the Lo value, the subscript remains suitable for `spec_fd` throughout the loop. The SSCC methods rely upon recognition by the compiler of certain common loop constucts such as this one.

If a loop manipulates a pointer passed as a parameter, the bound is not provided by the declaration. The compiler can infer the bounds Requirement of a pointer parameter from a loop involving subscript or pointer arithmetic. If the loop performs fetch-or-store up to and including the n-th element then n is the Hi; if the loop stops just before fetch-or-store on the n-th element then n is the Toofar; and similarly for a limiting address (pointer) value. Here is a simple example:

```
void f(char *q, int n) {
  p = q;
  for (int i=0; i<n; i++) {
    *p = ‘\n’;
  } /* … */
```

As written, the compiler infers from this loop that &q[n] (or just n) is the Toofar of p (and q), because the n-th element is not accessed. But if we add another line

```
*p = ‘\0’;
```

after the loop-end, the compiler infers that n is the Hi. (To be more precise, the Requirement is that n is “suitable” for the Hi, i.e., that the “real” Hi of the actual object is greater than or equal to the argument passed to this function. In order to create a simple notation in keeping with the intuition of programmers and implementers, we use the same terms, like “Hi”, to define a “greater-than-or-equal-to” semantics for Requirements, and an “exactly-equal” semantics for the Guarantee provided by a defining declaration.)

A similar rule infers the Nullt (null-terminated) attribute from a loop that searches for a null character; here is a simple example:

```
while (*p++ != ‘\0’)
  ;
```

Note that in these examples, the specified attribute is both a Requirement (pre-condition) and a Guarantee (post-condition). This is usually adequately clear from the context, but a notation for “Pre” and “Post” can be employed when needed. Also note that the attributes and relationships stated for a returned value are always Guarantees and not Requirements (obviously).

SSCC does not require whole-program analysis. Along with each source file (and/or each object file, including object files in libraries) there is a tabulation known as the bounds-data file, specifying Requirements and Guarantees for each function. For example, the bounds data file for memset specifies something like this:

```
memset(p, v, n IS_TSIZE_OF(p) )
```

Having seen this Requirement on the arguments to memset, the compiler can verify that the following invocation clearly meets the Requirement, because the sizeof operator produces the required Tsize:

```
memset(&spec_fd[i], 0, sizeof(spec_fd[i]))
```

Let’s change the example, to pass an integer unknown to the compiler:

```
memset(&spec_fd[i], 0, some_fn() )
```

The SSCC methods are unable to verify this at compile-time. In a later section we describe the methods for run-time verification.

Here we define the Requirements for the basic pointer and array operations in SSCC. The notation “p[0]” will designate an array or pointer-into-array or pointer-to-non-array being accessed by any equivalent form of indirection, including “*p” and “p->member” and “(*p).member”. The notation “p[i]” will designate an array or pointer-into-array being accessed by any form of indexed indirection, including “*(p+i)” and “*p++” and “*++p” and the corresponding forms using minus instead of plus. The notation “p+i” will designate any form of pointer arithmetic, including “p++” and “++p” and the corresponding forms using minus instead of plus.

- Fetch or store indirect via p[0]
  Requires: p Indir (p is Indirectable)
- Fetch or store indirect via p[i]
  Requires: p+i lies within [Lo,Hi],
  i.e., lies within [Lo,Toofar)
- Calculate p+i
  Requires: p+i lies within [Lo,Toofar]

The asymmetry between the Requirements for p[i] and p+i is required by the Standards (see 6.5.6 Additive operators, paragraphs 8 and 9, in [5]); the Toofar value is a valid result for pointer arithmetic, but it cannot be used for fetch or store.

4. LINK-TIME VERIFICATION

After compilation of all source files in the application, the SSCC linker verifies the compatibility of called functions with the calling context, and of uses of external objects with their defining instances, checking all Requirements against all Guarantees. In C and C++, the defining instance of each array will provide definite bounds for the array; moreover, the bounds are constants. Therefore, any Requirements on bounds of external array objects can be verified at link-time.

The discussion of the Requirements for the memset function illustrates the possibility that a bounds-data file may provide Requirements at the time the calling context is compiled. However, two C or C++ source files can each provide calls to a function in the other file, so no scheme of ordering of compilation can guarantee a simple ordering. By requiring a complete traversal of the bounds-data files at link-time, we eliminate ordering-dependencies and verify that the bounds-data files reflect the latest compilation of the corresponding source files.

SSCC specifies “type-compatible linkage for C programs”. This is slightly different from an already-standardized feature of C++ known as “type-safe linkage”, which provides checking between calling functions and called functions to verify that arguments and parameters have (exactly) the same types.

“Type-compatible linkage” is a less restrictive linkage rule which imposes only the C rules of “compatible types” having the “same alignment and representation”. The difference is largely a matter of portability. If int and long have the same alignment and representation on a particular platform, and function f takes one int parameter, and one object file invokes f with a long argument, then type-safe linkage will report a mismatch of types, but type-compatible linkage will accept the linkage on this particular platform. But on a different platform on which int and long have different alignment or representation, then both forms of linkage will complain.

There are several reasons why type-compatible linkage is required for the SSCC methods. First, standard C still permits the “old-style” function definition and declaration, in which no type information is available for compile-time checking; type-compatible linkage ensures that values are passed correctly for this platform. Second, function prototypes might differ between the called and calling contexts, whether by "versioning" changes over time, or by programmer carelessness. Third, C provides the "varargs" calling convention, which is discussed later.

The use of type-compatible linkage is one of several options on an SSCC platform for C. Another option is to require the exact match, as required by the type-safe linkage of C++ (creating a restrictive subset of C). Either linkage is adequate for the requirements of SSCC for C.

Note that in C++ the type-safe linkage rules are also employed to provide function overloading, which is not a feature of C (under either linkage rule).

The type-compatible (or the type-safe) linkage might (or might not) be implemented using name-mangling, a scheme by which a sequence of types is converted by the compiler into a short character string. (For a detailed example of name-mangling, see [13].)

For purposes of traceability and verification, the bounds-data file incorporates checksums for the associated source and object files, to provide a definitive connection between the linked application and the various constituent components.

5. RUN-TIME VERIFICATION

We do not claim that the SSCC compile-time and link-time verification will find all buffer overflows. There will be cases where the compiler has identified the relevant bounds data but cannot verify the values at compile time, requiring run-time verification.

It is well known that run-time verification can be much more efficient than slavishly performing a test at every reference. Loop-limit values need to be tested only once, before starting the loop. Optimizations of the code-hoisting variety can perform verification earlier. Further optimizations are known; for example, see Gupta [4].

Although the general subscript or pointer test implies two bounds, lower and upper, in almost every case the attributes of the pointer or subscript indicate monotonic progress in one direction. Therefore in almost every case the pattern of assembler code introduced into the run-time code sequence is one comparison instruction followed by a conditional branch. Furthermore, the conditional branch is almost never taken. Most modern platforms provide methods either in the hardware itself or in the compiler software whereby the optimization choices will avoid slowdowns for the almost-never-taken branch.

SSCC provides "Keep On Running" modes for embedded (or unattended) systems (including semantics known as "saturation", "modwrap", and "zerobound"). For the purposes of the present Workshop, however, we propose that run-time bounds-check failures must produce either a breakpoint that causes interruption of the running program and an opportunity to debug interactively, or an immediate invocation of the standard `abort()` function. (This choice between two behaviors is called an "abort constraint handler", described in more detail below.)

We created a prototype of the SSCC methods in order to estimate the execution penalty for the run-time tests. Our tools were able to compile, link, and execute seven of the SPEC benchmarks [12]: 164.gzip, 176.gcc, 181.mcf, 197.parser, 256.bzip2, and 300.twolf. Simple static analysis identified declarations and loops that provided bounds, as well as fetch-or-store expressions that required bounds. We instrumented the SPEC benchmark programs to count each execution of a fetch-or-store expression that was not categorized as "compile-time". We hand-estimated the percentage of the counted expressions that should have been recognized as compile-time by a full SSCC implementation, and the percentage of tests which could be eliminated by the various optimization methods described above. The detailed raw data and calculated results from all the tests are provided on the SSCC website [9]. The average estimated run-time overhead was less than 2%, which is significantly better performance than results from other comparable technologies. (For one comparison example, Ruwase and Lam [10] report that by confining their method only to strings, a run-time overhead less than 26% was achieved in most of their samples.)

The SSCC method provides special semantics for "varargs" functions, i.e. functions that accept a varying number of arguments. The C and C++ standards define certain functions which accept a varying number of arguments of heterogeneous types, such as `printf.` The `printf` format string specifies which argument types are expected. If at run-time the actual arguments do not agree with the expected types, undefined behavior results. This is a real vulnerability which has been exploited by hackers, just as buffer overflows have been. Furthermore, this vulnerability can be used to create subsequent buffer overflows. In an SSCC implementation we require two alternative forms of varargs library functions: one which provides no run-time checking of argument types, and one which does provide checking. If the compiler can see that the format-string argument is a constant character string, then at compile-time the compiler can determine whether the actual arguments match the expected types. If successful, the compiler invokes the (faster) alternative without run-time checking. If the compile-time match fails, the compiler can issue a fatal diagnostic so the programmer can fix the problem.

But in some cases it cannot be determined at compile-time whether a varargs function's actual arguments match the expected types. In this situation, the SSCC compiler will add an extra character string argument after the named arguments. The string contains the type-compatible name-mangled list of the types of the actual arguments passed in this function call. Then the called function must also be compiled by the SSCC compiler, which performs a little extra work in the called function as each argument is extracted by the `va_arg` macro from the header `<stdarg.h>`. If the type argument is a scalar type which produces a one-byte encoding in the mangled name string (e.g. `double`, which produces the single character `'d'` in a typical name-mangling), then an invocation such as

```
p = va_arg(ap, double);
```

produces a translated invocation such as

```
p = _va_arg1(ap, double, 'd');
```

The enhanced `_va_arg1` macro tests that the next byte in the argument mangled-name string is the character `'d'`, incrementing the pointer after the test. (This is typically a reasonably fast operation on most hardware: a test and a post-increment.) If the argument has a type which produces a multiple-byte encoding in the mangled name string (e.g. pointer-to-int, which produces the string `"Pi"` in a typical name-mangling), then an invocation such as

```
  p = va_arg(ap, int*);
```

produces a translated invocation such as

```
  p = _va_arg2(ap, double, ‘P’, ‘i’);
```

The _va_arg2 macro tests that the next two bytes in the argument mangled-name string are the characters “Pi”, incrementing the pointer after the test. (Further macros handle more types with longer mangled names. In addition, C has some special rules about varargs type-compatibility.)

The rules for creating the expected-type character, or string of characters, for variable-argument functions permit more matches than the strict type-safe rules of C++. The intent, as described for type-compatible linkage, is to accept C and C++ programs which work reliably in today’s environment, even if some portability problem might be lurking (to be diagnosed if and when the program is compiled on another platform or compiled with further portability-checking options).

If the varargs argument mangled-name characters fail these type-matching rules, an abort constraint handler is invoked (interactive debugger breakpoint, or abort).

6. NEW LIBRARY FOR C

The C standards committee is currently working on one piece of the security puzzle: WDTR 24731, a Technical Report for a new C library [7]. Among other features, the new library provides new APIs which permit, or encourage, the programmer to provide bounds information for all array arguments. Furthermore, arrays-of-characters created by these APIs are always null-terminated.

These functions validate their arguments and the bounds requirements for the arrays they produce; these requirements are known as the “runtime-constraints”. If a requirement is violated, the function invokes a “constraint handler”. The behavior we described above as the “abort constraint handler” is the default behavior in Microsoft’s Visual Studio 2005 which provides a complete implementation of the WDTR 24731 library [8].

The new library provides a new typedef for specifying the sizes of arrays, called rsize_t, and an associated maximum value named RSIZE_MAX. It is recommended that, for implementations targeting machines with large address spaces, RSIZE_MAX be defined as the smaller of the size of the largest object supported or (SIZE_MAX >> 1), even if this limit is smaller than the size of some legitimate, but very large, objects. This way, if a negative number is (incorrectly) given as the size of an array, after the (wraparound) conversion to an unsigned number, it will be recognized as a violation of a runtime-constraint. Before the introduction of RSIZE_MAX, this sort of bug could cause the over-writing of large areas of memory.

The old APIs returned success-or-fail information in an inconsistent variety of conventions that mingled successful returned information with indications of failure. The new APIs consistently return an indicator of “success-or-what-kind-of-failure” using an integer type named errno_t.

Consider the strcpy_s function, which accepts the address where the copied characters will be stored, plus an integer specifying the size of that array.

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

By the explicit provision of bounds information for the target string, this API provides the opportunity to diagnose errors that could have caused buffer overflows with the old strcpy API.

7. EXTENDING TO ALL PROGRAMS

To this point, we have described methods by which SSCC ensures proper fetch-and-store accesses using only the variables defined by the programmer. These methods will in some cases require a fatal diagnostic for situations in which the compiler and linker cannot determine whether a fetch or store introduces undefined behavior. Examples include unusually complex instances of aliased pointers, buffers created by malloc, and interprocedural dependencies. The recent article by Ruwase and Lam [10] has shown another method which can be applied to these most-difficult cases. In this alternative, unverifiable fetch-or-store operations can be checked by requiring that all potential fetched-or-stored objects be entered into run-time tables (i.e. “dynamic tables”).

By this method, hastily-written programs (“one-off jobs”) can be compiled and executed with certainty that, whatever flaws they might contain, they will not execute buffer overflows. In addition, some large legacy applications (“dusty decks”), or portions thereof, might not be worth top-to-bottom remediation to prevent buffer overflows. Adding dynamic tables to the SSCC methods permits a choice based upon cost-benefit considerations.

8. COMMERCIAL IMPLEMENTATION

In order for methods like SSCC to make a significant difference in the reliability of the software infrastructure, we must get the methods into the tools that working programmers are using to build their applications. We suggest that there are two different avenues to adoption; we refer to them as “remediation tools” and “compiler tools”.

Remediation tools are intended to provide assistance when a group has made the decision to spend resources on improving some body of source code (typically hundreds of thousands, or millions, of lines of code). Such decisions are typically prompted by corporate IT management, software QA, corporate standards, etc. There are several commercial software-quality tools which serve this marketplace, including offerings from PolySpace, Coverity, Fortify Software, Secure Software, Klocwork, and others. All of these products provide some assistance with preventing buffer overflows, but to our knowledge none of them provide certification that *all* buffer overflows are detected and prevented (which is the essential feature of the SSCC methods). However, these products do much *more* than check for buffer overflows; they detect bugs, catch other security problems, and enforce corporate coding standards, etc. One or more of the quality-tools producers could add the SSCC methods to their remediation tools to provide assistance to projects attempting to revise their source code to definitively eliminate buffer overflows.

Remediation tools can also perform a one-time conversion from the old C library to the new library [7]. For each (“non-deprecated”) function defined in the new library (such as strcpy_s), there is a corresponding function that lacks some indication of the bounds data of the target (such as strcpy); call that the “corresponding deprecated function”. The set of all the corresponding deprecated functions constitutes the “deprecated functions”. For each invocation of a deprecated function in the program being compiled, the bounds-data Requirements are well-known from the Standards. If the remediation tool employing the SSCC method is unable to determine a corresponding bounds-data Guarantee, then a fatal diagnostic is issued and an expert needs to study the problem. Otherwise, the source code invocation is re-written by the remediation tool to an invocation of the corresponding non-deprecated function, in which the bounds-data Guarantee is explicitly passed as an argument. If the source-code context tests the returned value from the deprecated function, then the remediation tool rewrites the success-or-fail test into a test against the "`errno_t`" returned value from the corresponding non-deprecated function.

These various forms of large-scale remediation should be of interest to the large consultancies that provide skilled talent to clients worldwide, such as Accenture, Bearing Point, IBM Business Consulting, McCabe, Watchfire, and EDS.

Compiler producers constitute a segment of the software production supply chain, one that is quite different from the quality-tools producers. Each hardware company typically maintains some number of compiler groups, as do several of the large software producers. There are several specialized compiler producers. In addition, there is a significant community of individuals and companies that support the open-source Gnu Compiler Collection (gcc). Adding these various groups together, we estimate that there are well over 100 compiler vendors. In order to encourage adoption of the SSCC methods into working compilers, we propose a general-purpose "SScfront" tool, to take the output from the C/C++ preprocessor, perform the SSCC methods (including reading from and writing to the SSCC bounds-data files), and produce a transformed C source code to be compiled by the platform-dependent compiler. Along with the SScfront component, an SSCC "pre-linker" would also be required, to read and process the full collection of bounds-data files from all components of the application being compiled and linked. If or when the SSCC methods become popular in the marketplace, compiler producers can doubtless produce more efficient and better integrated "all-in-one" solutions, just as the initial "cfront" implementation of C++ was replaced by integrated compiler solutions over a period of years.

A third market segment contains the component producers, which provide specialized components to the compiler producers and quality tools producers; see Figure 1 below.

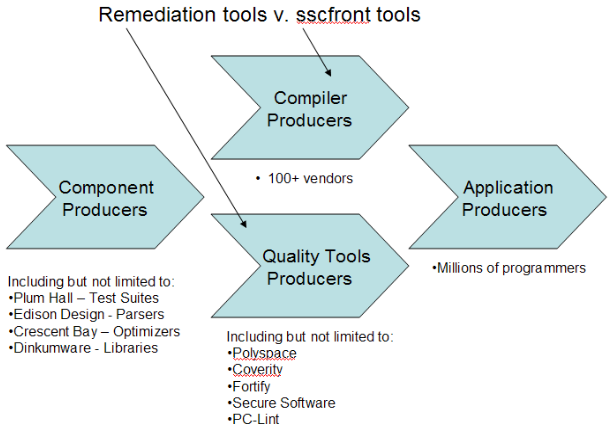

Figure 1. Software Production Supply Chain

In general, component producers don't want to make products that would compete with their customers. A successful adoption strategy for eliminating buffer overflows will need to take account of the unique position of each market segment.

At some point, the compiler or quality tool implementing the SSCC methods will be prepared to certify that the application is free from buffer overflows. Because of the significant costs that buffer overflows have imposed upon the market, certified absence of buffer overflows should provide significant economic value in several market segments.

After demonstrating utility in the marketplace, the SSCC methods should be standardized, with permissions adequate for incorporation into open-source as well as proprietary products. We suggest, however, that too many technologies have been introduced with an emphasis upon market share and insufficient attention paid to requirements of security. We maintain sufficient IPR protection for the SSCC methods to permit taking effective action against "spoofers" that would weaken the expectations of producers, users, and the public.

9. CONCLUSIONS

We itemize the novel features of the SSCC methods:

- Combine static-analysis methods with dynamic-analysis methods, to create a hybrid solution;
- Define an extensive (non-orthogonal) set of attributes and relationships that match the concepts intuitively used by programmers in constructing professional programs, and define their role in preventing buffer overflows;
- Automatically infer the Requirements on the interface of each callable function;
- Supplement the compilation and linking mechanism by producing and using bounds-data files which record Requirements and Guarantees for the defined and undefined symbols in one or more corresponding object files, as well as checksum information;
- Verify C linkage using type-compatible linkage;
- Verify type-compatible behavior of varargs functions, using a name-mangled string at run-time;
- Provide automated remediation of each input source file into a source file which invokes non-deprecated functions in the new C library.

The details involved in SSCC are extensive, but all work together to achieve properties which can be stated simply: Bounds information is kept in parallel with the source and object code, and in particular kept in parallel with each callable function's interface. When a fetch or a store is performed, available bounds information is used at compile time, link time, or run time, to determine the validity of the fetch-or-store operation.

10. ACKNOWLEDGMENTS

Our thanks to the anonymous reviewers, and to Brian Brode, Bruce Galler, David McNamara, Larry O'Brien, Roland Racko, Robert Seacord, Itaru Shimoyama, Youichi Sugiyama, and Steph Walli for their comments on various drafts of this material.

11. REFERENCES

[1] CERT/CC. See http://www.cert.org/stats/cert_stats.html for current statistics.

[2] CERT/CC. US-CERT's Technical Cyber Security Alerts. http://www.us-cert.gov/cas/techalerts/index.html

[3] Dor, N., Rodeh, M., and Sagiv, M. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, June 2003. http://portal.acm.org/citation.cfm?doid=781131.781149

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

[5] INCITS/ISO/IEC 9899-1999. Programming Languages — C, Second Edition, 1999.

[6] INCITS/ISO/IEC 14882-2003. Programming Languages — C++, Second Edition, 2003.

[7] ISO/IEC WDTR 24731. Specification for Secure C Library Functions, 2004. (Options for ordering [5,6,7] are kept updated at http://www.plumhall.com/990216ansi.html.)

[8] Lovell, M. Safe! Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries, MSDN Magazine, May 2005, http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx

[9] Plum Hall, Inc. *The SSCC website.* http://www.plumhall.com/sscc.html (free, requires registration).

[10] Ruwase, O., and Lam, M. A Practical Dynamic Buffer Overflow Detector, In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, February 2004.

[11] Seacord, R. *Secure Coding in C and C++*. Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.http://suif.stanford.edu/papers/tunji04.pdf

[12] System Performance Evaluation Corporation (SPEC). SPEC CPU2000: Component CPU Integer (CINT2000), 2000. http://www.spec.org

[13] Williams, M. et al., "Itanium C++ ABI". http://www.codesourcery.com/cxx-abi/abi.html