Modeling and Analyzing the Interaction of C and C++ Strings

Gogul Balakrishnan\textsuperscript{1}, Naoto Maeda\textsuperscript{2}, Sriram Sankaranarayanan\textsuperscript{3}, Franjo Ivančić\textsuperscript{1}, Aarti Gupta\textsuperscript{3}, and Rakesh Pothengil\textsuperscript{4}

\textsuperscript{1} NEC Laboratories America, Princeton, NJ
\textsuperscript{2} NEC Corporation, Kanagawa, Japan
\textsuperscript{3} University of Colorado, Boulder, CO
\textsuperscript{4} NEC HCL ST, Noida, India

Abstract. Strings are commonly used in a large variety of software. And yet, they are a common source of bugs involving invalid memory accesses arising due to misuses of the string manipulation API. These bugs are often remotely exploitable, leading to severe consequences. Therefore, the static detection of invalid memory accesses due to string manipulations has received much attention, especially for C programs using the standard C library functions. More recently, software development is increasingly being performed in object-oriented languages such as C++ and Java. However, the need to interact with legacy C code and C-based system-level APIs often necessitates the use of a mixed programming paradigm that combines features of high-level object-oriented constructs with calls to standard C library functions. While such programs are commonplace, there has been little research on static analysis of such code. We present heap-aware memory models for C++ programs, with an emphasis on modeling features such as dynamically allocated memory, use of null-terminated buffers as strings, C++ standard template library (STL) classes and interactions between these features. We use standard verification tools such as abstract interpretation and model checking to verify properties over these models to find potential bugs. Our tool can find several previously unknown bugs in open-source projects. These bugs are primarily due to the intricate C++ programming model and subtle interactions with legacy C string functions.

1 Introduction

Buffer overflows are common in systems code. They can lead to memory corruption and application crashes. They are particularly dangerous if they can be exploited by malicious users to deny service by crashing a system or escalate privileges remotely. A large number of overflows are present in deployed commercial as well as open-source software \cite{18}. A significant volume of research on buffer overflow prevention has focused on the detection of overflows in C code.

Software development teams have shifted their development from C to object-oriented languages including C++ and Java. The benefits of using an object-oriented language include reusability, better maintainability, encapsulation and
the use of inheritance. In particular, C++ is often chosen due to its ability to interact with legacy C-based systems, including system-level C libraries. Thus, development in C++ often necessitates a mixed programming style combining object-oriented constructs with lower-level C code. Whereas a large volume of work on verification has focused on C programs, there has been comparatively little work on the verification of C++ programs. The modeling of objects in the heap is a key component of such verification. In this paper, we present heap-aware static analysis techniques that can verify memory safety of C/C++ programs. Our approach focuses on the modeling of strings in C/C++ and buffer overflow errors due to the interaction and misuse of string manipulation functions.

```cpp
class Object {
  // ...}

class Relobj : public Object {
  // ...}

1: void Icf::find_identical_sections(
2:  const Input_objects* input_objects, Symbol_table* systab){
3:     // ...}
4:  for (Input_objects::Relobj_iterator p =
5:      input_objects->relobj_begin();
6:      p != input_objects->relobj_end(); p++) {
7:      // /* (p) is of type Relobj* */
8:      const char* section_name=(*)p->section_name(i).c_str();
9:      // /*(*p)->section_name(.) resolved to Object::section_name()*/
10:     if ( !is_section_foldable_candidate(section_name) )
11:       /* invalid use */
12:     // ...}
```

![Fig. 1. Motivating example from GNU binutils v2.21.](image)

Motivating example. A typical “interaction bug” is shown in Figure 1. The code snippet is taken from the gold project, part of the GNU binutils (binary utilities) package (v2.21). Gold is a linker that is more efficient for large C++ programs than the standard C linker. For convenience, we have added labels to denote line numbers of interest. Consider the call to c_str() in line 6 of the function find_identical_sections. The call (*p)->section_name(i) creates a temporary object (see labels A–C in a class Object). The call to the c_str() method thus obtains a pointer to a C string, pointing into the temporary object.
However, the subsequent uses of that string, stored in the variable `section_name`, are invalid. The temporary object (including the pointed to C string) is destroyed immediately following the call to `c_str()`. Under certain conditions the freed memory may be re-used, leading to segmentation fault or memory corruption. Thus, the call to `is_section_foldable_candidate()`, and further uses of the variable not shown here, produce unexpected behavior.

This example shows some typical C++ code. Note that just considering the call to `c_str()` is not enough to find this bug. If `Object::section_name()` (lines A-C) had returned a reference, this use of `c_str()` would likely have been legal. Due to the hidden side effects in C++ and the interaction with legacy C APIs, such bugs are easy to commit and hard to find. Furthermore, the bug in binutils had gone unnoticed for at least a year in spite of rigorous testing (the bug was introduced before the release of v2.20, which was officially released in October 2009). It is likely that under normal runtime deployment or during unit testing, the pointer assigned to `section_name` still contains the original string even after it is destroyed. However, under large resource constraints, this bug may manifest itself likely through a segmentation fault upon a later use of `section_name`. Finally, note that a static analysis needs to handle numerous C++ specific issues including STL classes, complex inheritance, and iterators.

**Our Approach.** Given a program and the properties to check, we use an abstraction to model the memory used by arrays, pointers, and strings. The memory model abstraction only tracks the attributes and operations that are relevant to the properties under consideration. We focus on providing precise and scalable memory model for the usage of C and C++ strings. In particular, we address the intricate interplay between C and C++ strings.

Instead of providing a universal memory model, we partition the set of potential bugs into various classes, and use different models for the different classes. Tailoring the memory models to the class of bugs makes the analysis and verification more scalable. For instance, while checking for NULL-pointer dereferences and use-after-free bugs, we use an abstraction that only tracks the status of the pointer, and does not keep track of buffer sizes and string lengths. On the other hand, we use a more precise analysis model that keeps track of allocated memory regions and string lengths for checking buffer overflows.

One particular distinguishing feature of our memory models is that we provide a unified framework that addresses correct usage of C-based strings, the C++ STL string class, as well as the interaction between the C++ string class and C strings through conversions from one to the other. Whereas heap aware models for C programs have been well studied [10192022330], our model handles C++ objects including memory allocation using `new/delete`, the string class in STL and the interaction of these features in C++. To deal with the interaction of C and C++ strings, we introduce a notion of non-transferable ownership of a C-string. We utilize this ownership notion to find dangling pointer accesses of C-strings that were obtained through a conversion from a C++ string.

The memory models are woven into the program under consideration and is then verified using various static analysis and model checking techniques. First,
we employ abstract interpretation \cite{16} to prove properties using a variety of numerical abstract domains \cite{15,17,28}. The proved properties are eliminated, which enables us to simplify the model of the program. Then, we use a model checker, in particular a bit-accurate SAT-based bounded model checker \cite{21,13}, to find proofs and violations for the remaining properties. The model checker outputs concrete witnesses that demonstrate (a) the path taken through the program to produce the violation and (b) concrete values for the program variables.

The major contributions of this paper are as follows:

\begin{itemize}
\item We present sophisticated, yet scalable, heap-aware memory models for analyzing overflow properties of C and C++ programs that use features including arrays, strings, pointer arithmetic, dynamic allocation, multiple inheritance, exceptions, casting, and standard library usage.
\item Our approach tackles the interaction of C and C++ strings, thus enabling our tool to discover subtle bugs in the interaction between the different string kinds. We separate the checks into two classes: a pointer-validity-based checking class and a string-length-based checking class. We also introduce a notion of non-transferable ownership or origination of a C-string for strings obtained through conversion from the C++ string class.
\item We implemented our models and demonstrate their usefulness on real code, where we found previously unknown bugs in open-source software. To find these bugs, our tool uses abstract interpretation for proving properties and bit-precise model checking for finding concrete witness traces.
\end{itemize}

\section{Preliminaries}

We provide an overview of our analysis framework for C, and present a taxonomy of bugs related to the usage of C++ strings. This taxonomy will be used to guide our subsequent modeling of the string class and its interaction with C strings.

\subsection{Overview of Analysis Framework}

In the past, we have developed a general analysis framework for C programs called F-Soft \cite{26}. It uses both abstract interpretation and bounded model checking to find bugs in the source code under analysis. F-Soft contains a number of “checkers” for various memory safety issues. These include a memory leak checker (MLC), a pointer validity checker (PVC) and an array buffer overflow checker (ABC). These checkers use different levels of abstraction, and thus, explore different trade-offs between scalability and their ability to reason about intricate pointer accesses. For example, PVC targets bugs such as use-after-free, accesses of a NULL pointer, freeing of a constant string, etc. On the other hand, ABC targets violations that require reasoning about sizes of arrays and strings, and whether strings are null-terminated. To improve scalability of ABC, properties that could be checked using PVC are not considered by ABC. In this paper, we omit discussion of other checkers available in our tool for sake
of brevity. These include checkers for the use of uninitialized memory (UUM) and an exception analysis (EXC) that computes exceptional control flow paths in C++ programs, for example. The EXC checker can also find uncaught exception violations [32]. Ultimately, all checkers generate a model of the program with embedded properties that can be checked by the subsequent analysis engines.

**Figure 2. Main analysis components**

Figure 2 depicts the major analysis modules used in our tool. The overall flow is geared towards maximizing the number of property proofs and concrete witnesses of property violations. After model construction for a given program, we analyze the model using abstract interpretation in an attempt to prove that assertions are never violated. Assertions that can be proved safe are removed from the model, and the final model is sliced based on the checks that remain unresolved. In practice, the sliced model is considerably smaller than the original. The model is then analyzed by a series of model-checking engines, including a SAT-based bounded model checker. At the end of model checking, we obtain concrete traces that demonstrate property violations in the model. These violations are mapped back to the source code and displayed using an HTML-based interface or a programming environment such as *Eclipse*(tm). We briefly describe the major components in the flow:

**Abstract Interpreter** Abstract interpretation [16] is used in our flow as the main proof engine. Our abstract interpreter is inter-procedural, flow- and context-sensitive. Currently, we have implementations of abstract domains such as *constants, intervals* [15], *octagons* [29], and *polyhedra* [17]. These domains are organized in increasing order of complexity. After each analysis is run, the proved properties are removed and the model is simplified using slicing. The resulting model is then analyzed by a more complex domain.

**Model Checker** The model checker creates a finite state machine model of the simplified program after abstract interpretation. Each integer variable is treated as a 32 bit entity, character variables as 8 bits and so on. However, the range information provided by the abstract interpreter for program variables is used to reduce the number of bits significantly. We use bit-accurate representations of all operators, ensuring that arithmetic overflows are modeled faithfully.

The model checker verifies the symbolic model for the reachability of the embedded properties. We primarily use SAT-based bounded model checking [7]. This technique unrolls the program up to some depth $d > 0$ and searches for the presence of a bug at that depth by compilation into a SAT problem. The depth
$d$ is increased iteratively until a bug is found or resources run out. The model checker generates a counterexample (witness trace) which vastly simplifies the user inspection and evaluation of the error.

### 2.2 C++ String Class Usage Issues

C++ STL strings provide a safer alternative to developers when compared to C strings. However, as shown in the motivating example (see Figure 1), mistakes are still easy to make, especially in the interaction with C-based standard library functions. The string class contains a number of built-in features such as modification routines (append, replace, etc), operations such as substring generation, iterators, and others. Additionally, the methods c_str() and data() can be called to obtain a buffer containing a C string, which is null-terminated for c_str() and not null-terminated for data(). Our description focuses on the c_str() method, but is applicable to data() as well. Moreover, the data() method is even more error-prone due to it returning a non-null-terminated string.

We classify common bugs related to the use of strings below:

1. Generic bugs: Memory leaks, uncaught exceptions (e.g., std::bad_alloc) [32].
2. String class manipulation errors:
   a. Out of bounds access. std::out_of_range exception thrown by the at and operator[] methods of the string class.
   b. Use of a string object after it has been destroyed.
   c. Use of a stale string iterator.
3. Interaction between C and C++ strings
   a. Access of C-string returned by string::c_str(), after the corresponding C++ object is destroyed.
   b. Certain C library functions called on strings obtained through c_str().
   c. Manipulation of a C-string returned by string::c_str().
   d. C-based buffer overflows on C-string obtained through string::c_str().

### 3 Program Modeling and Memory Checkers

We now discuss the memory models used in our approach. Our approach supports a hierarchy of memory models ranging from models that simply track few bits of allocation status for each pointer to the full-fledged tracking of allocated bounds, string sizes, region aliasing of arrays, and so on. We describe two models within this spectrum: the pointer validity model that uses simple pointer type states, and the pointer bounds model that attempts to track allocated bounds, positions of various sentinels, and contents of cells accurately.

#### 3.1 Pointer Validity Model

The validity model instruments for each pointer a validity status ptrVal(p) to denote the type of the location pointed-to in memory. These values include null
indicating a null pointer; invalid for a non-NULL pointer whose dereference may cause a segmentation violation; static for pointers to global variables, arrays, and static variables; stack for pointers to local variables, \texttt{alloca} calls, local arrays, formal arguments; heap for pointers to dynamically allocated memory on the heap; and code for code sections, such as string constants.

The validity model does not track addresses of pointers. It also ignores address arithmetic. A pointer expression $p+i$ has the same validity status as its base pointer $p$. A dereference $*p$ yields an assertion check that is violated if $\text{ptrVal}(p)$ is null or invalid. Similarly, relevant checks are done for other operations. We distinguish between null and invalid in order to allow \texttt{delete} \texttt{NULL}, which is allowed per C++ standard, as well as optionally allow \texttt{free}($\text{NULL}$), which is handled gracefully by standard compilers such as gcc. Finally, note that it is easy to extend this model to find invalid de-allocations, such as the case where memory that was allocated using \texttt{new} is released using \texttt{free}. This can be accomplished by separating the validity status heap into sub-regions according to their allocation method, such as heap – malloc, heap – new and heap – new[].

### 3.2 Pointer Bounds Model

The bounds model tracks various attributes for each pointer, including allocation sizes and sentinel positions, which subsumes information tracked by the validity model. For a pointer $p$, the main modeling attributes are as follows (see Figure 3):

1. $\text{ptrLo}(p)$, which corresponds to the base address of a memory region that $p$ currently points to;
2. $\text{ptrHi}(p)$, which corresponds to the last address in the memory region currently pointed to by $p$ that can be accessed without causing a buffer overflow;
3. $\text{strLen}(p)$ which corresponds to the remaining string length of the pointer $p$, which is the distance to the next null-termination symbol starting at $p$.

```
char *s = malloc(10);
if (!s) exit(-1);
strcpy(s,"FeVe008");
char *t = s+4;
```

**Fig. 3.** The memory model for the pointer bounds model after successfully executing the four statements on the left-hand side: The successful allocation returns a pointer to some new address $M$, and the lower bound addresses $\text{ptrLo}(s) = \text{ptrLo}(t) = M$. The higher bound addresses are $\text{ptrHi}(s) = \text{ptrHi}(t) = M + 9$. Finally, the string lengths are determined using the size abstraction, namely $\text{strLen}(s) = 7$ and $\text{strLen}(t) = 3$. The dotted memory region denotes out-of-bound memory regions for the pointers $s$ and $t$. 
For each pointer $p$, we track its “address”, and its bounds $[\text{ptrLo}(p), \text{ptrHi}(p)]$, representing the range of values of the pointer $p$ such that $p$ may be legally dereferenced in our model. If $p \in [\text{ptrLo}(p), \text{ptrHi}(p)]$ then $p[i]$ underflows iff $p + i < \text{ptrLo}(p)$. Similarly, $p[i]$ overflows iff $p + i > \text{ptrHi}(p)$.

**Dynamic Allocation** We assign bounds for dynamic allocations with the help of a special counter $\text{pos}(L)$ for each allocation site $L$ in the code. It keeps track of the maximum address currently allocated. Upon each call to a function such as $p := \text{malloc}(n)$, our model assigns the variable $\text{pos}(L)$ to $p$ and $\text{ptrLo}(p)$. It increments $\text{pos}(L)$ by $n$, and sets $\text{ptrHi}(p) + 1$ to this value.

**C String Modeling** Conventionally, strings in C are represented as an array of characters followed by a special null-termination symbol. String library functions such as `strcat`, and `strcpy` rely on their inputs to be properly null-terminated and the allocated bounds to be large enough to contain the results. We extend our model to check for such buffer overflows using a size abstraction along the lines of CSSV [22]. The major differences are described in Section 5.

For each character pointer $p$, we use an attribute $\text{strLen}(p)$ to track the position of the first null-terminator character starting from $p$. The updates to string length can be derived along similar lines as those for the pointer bounds. For instance, calls to the method `strcat` that append its second argument to the first lead to assertion checks in terms of the pointer bounds and string lengths that guarantee its safe execution. Next, the update to the $\text{strLen}$ attribute of the first argument is instrumented. Our approach currently has instrumentation support for about 650 standard library functions. It provides support for parsing constant format strings in order to model effects of functions such as `sprintf`. We elide the details for lack of space and focus here on the modeling of C++ strings and their interaction with C.

## 4 Modeling the STL String Class

We now present a model for C++ strings that allows us to capture common bugs arising from the misuse of STL strings. Note that, for the sake of brevity, we omit the presentation of string iterator related issues in this paper. Furthermore, we will not discuss issues due to uncaught exceptions when utilizing the C++ string class. Details on our exception handling can be found in [32].

As in Section 3 we separate verification into a light-weight pointer validity-based checker and a more heavy-weight buffer overflow checker tracking accurate string lengths using an extension of the pointer bounds model. Finally, it should be noted that we model a wider class of C++ STL strings than alluded to so far. For example, we also model the templated class `std::basic_string< ? >`, of which `std::string` is just a particular instantiation.

### 4.1 Pointer and String Object Validity

Section 3.1 introduced a memory model that focuses on validity of pointers. Here, we extend it by introducing a new validity status that is used to model
Table 1. Overview of pointer and string object validity model. This table shows the effect of operations on different validity statuses. A potential error is marked using the symbol ⨀. Upon error, the validity status changes to invalid. If the update is safe, the table provides the resulting status after the client code operation. The entry N/A denotes that a particular step is not possible in our model. Operation “∗” denotes a pointer or object read/write, “return” denotes the end of a functional scope, “if-NULL” denotes a pointer equals null check, “”()” denotes a destructor call. Allocation (malloc, new), initialization operations (constructor calls), and other details are omitted for brevity.

<table>
<thead>
<tr>
<th>status</th>
<th>*</th>
<th>free</th>
<th>delete</th>
<th>delete[]</th>
<th>if-NULL</th>
<th>return</th>
<th>−()</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>✗</td>
<td>✗</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>✗</td>
</tr>
<tr>
<td>invalid</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>stack</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>invalid</td>
<td>✗</td>
</tr>
<tr>
<td>global</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>global</td>
<td>✗</td>
</tr>
<tr>
<td>code</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>code</td>
<td>✗</td>
</tr>
<tr>
<td>env.</td>
<td>✗</td>
<td>✗</td>
<td>invalid</td>
<td>invalid</td>
<td>invalid</td>
<td>null</td>
<td>env.</td>
</tr>
<tr>
<td>heap-malloc</td>
<td>Heap-malloc</td>
<td>invalid</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>heap-malloc</td>
<td>✗</td>
</tr>
<tr>
<td>heap-new</td>
<td>heap-new</td>
<td>✗</td>
<td>invalid</td>
<td>✗</td>
<td>N/A</td>
<td>heap-new</td>
<td>✗</td>
</tr>
<tr>
<td>heap-new[]</td>
<td>heap-new[]</td>
<td>✗</td>
<td>✗</td>
<td>invalid</td>
<td>N/A</td>
<td>heap-new[]</td>
<td>✗</td>
</tr>
<tr>
<td>ownerM.</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>N/A</td>
<td>ownerM.</td>
<td>✗</td>
</tr>
</tbody>
</table>

the interaction of C++ strings with C-based strings. We check most issues related to the interaction of C++ and C strings by developing an extended pointer and string object validity checker rather than additionally burdening the pointer bounds model. To do so, we model calls to string::c_str() such that they return C strings whose validity status is set to a new status that behaves roughly like the code status, denoting constant strings. A key difference is that the owning class instance, which returned the string in the first place, is allowed to manipulate this string, while no manipulations are permissible for constant strings.

This naturally leads to a notion of ownership [21][2] of pointers that is a common programming idiom. Thus, we introduce a new status ownerMutable. Prior work used transferable ownership models to find memory leaks in C++ code [23]. However, we only consider C-strings obtained from C++-strings. Thus, we limit ourselves to a non-transferable ownership model, which tracks the relationship between originating C++-string and owned C-string. This allows us to declare such ownerMutable strings as stale (that is, invalid), when the originating C++ object that owns it is modified using a method call.

We summarize the pointer and string object validity checker in Table 1. It shows the effect of various operations in the client code on the defined validity statuses. The handling of many operations including initialization, allocation, destructor calls and so on are omitted from the table in order to avoid clutter.

Figure 1 shows a partial sketch of our custom string object validity model. The internal assertion checks are represented as calls to a member function isValid(operation), which can be thought of as utilizing the information in
class string {
  /* pointer and string object validity model */
  private: char *p;
  public: ...
    string() {
      p = new char[1];   /* assumed to not fail */
    }
    string(const string &s) {
      ASSERT(s.isValid(READ-OP));
      p = new char[1];   /* assumed to not fail */
    }
    ~string() {
      ASSERT(this.isValid(DESTROY));
      delete[] p;
    }
    string substr(size_t p=0, size_t n=MAX) const {
      ASSERT(this.isValid(READ-OP));
      return string();
    }
    void push_back(char c) {
      ASSERT(this.isValid(WRITE-OP));
      delete[] p;    /* used to invalidate stale pointers */
      p = new char[1];  /* assumed to not fail */
    }
    const char *c_str() const {
      ASSERT(this.isValid(READ-OP));
      setValid(p, OWNER_MUTABLE);
      return (const char *) p;
    }
};

Fig. 4. Partial string object validity model sketch

Table [6] The sketch shows the use of a `setValid(void*, status)` method that can be thought of as setting the validity status for arbitrary pointers. The non-constant function `push_back(c)` shows how we invalidate C-strings that may have been obtained through `c_str()` earlier. Finally, note that we separate the issue of allocation failures through `new` from the string validity checking. As mentioned in the comments, we assume that each `new` operation succeeds.

Example 1. Figure [5] shows a simple C++ function that manipulates a C++ string and converts it to a C string. It proceeds to call `strlen` on this C string. A variety of intermediate transformations are performed on the C++ source code including transformations that make calls to constructors and destructors explicit. Figure [5] also shows the result of this transformation for method `cutLen`, which we call `cutLenX`. Note the use of a temporary variable as a result of our transformation, which is initialized using the copy-constructor, and then
// Simple C++ string use
int cutLen(const string &s, size_t i, size_t n) {
    const char *str = s.substr(i, n).c_str();
    return strlen(str);
}

// Simplified representation of cutLen
int cutLenX(const string &s, size_t i, size_t n) {
    const string tmp = string(s.substr(i, n));
    const char *str = tmp.c_str();
    tmp.~string();  // also invalidates str!
    return strlen(str);
}

Fig. 5. A simple example illustrating the interaction of C and C++ strings.

<table>
<thead>
<tr>
<th>stmt</th>
<th>&amp;s</th>
<th>&amp;tmp</th>
<th>str</th>
<th>stmt</th>
<th>&amp;s</th>
<th>s.p</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>substr(...)</td>
<td></td>
<td></td>
<td></td>
<td>initially</td>
<td></td>
<td>heap-new</td>
<td></td>
</tr>
<tr>
<td>C_str()</td>
<td>stack</td>
<td>stack</td>
<td>ownerM</td>
<td>str-s.c_str()</td>
<td>stack</td>
<td>ownerM</td>
<td>ownerM</td>
</tr>
<tr>
<td>tmp. string()</td>
<td>stack</td>
<td>invalid</td>
<td>invalid</td>
<td>s.pushback('a')</td>
<td>stack</td>
<td>heap-new</td>
<td>invalid</td>
</tr>
<tr>
<td>strlen()</td>
<td>stack</td>
<td>invalid</td>
<td>X</td>
<td>strlen(str)</td>
<td>stack</td>
<td>heap-new</td>
<td>X</td>
</tr>
</tbody>
</table>

(a) (b)

Fig. 6. (a) Updates to the validity status for the simplified code shown in Figure 5 assuming that the input string s was initially allocated on the stack. The destruction of the temporary object tmp. string() also invalidates the pointer str through aliasing. (b) Updates to the validity status for another sequence of statements shown in the column labeled stmt. The pushback operation first passes the required assertion, then invalidates the pointer str, and finally resets the internal pointer s.p to a fresh allocated region. The subsequent call to strlen(str) thus raises an error.

destroyed using an explicit call to ~string(). The bug in the code can thus be detected using the model of Figure 4 (see Figure 5(a)).

4.2 Pointer and string bounds model

The array bounds model for C strings is extended by tracking the logical size of each C++ string. This size is used to handle calls to string::c_str() and string::data(). Therein, we create valid C strings of the appropriate string length and allocation size, and null-termination status.

Figure 7 shows a simplified model for the c_str() method. Note that we do not check whether the string object is valid during calls to c_str() in this checker. These checks are already performed in the pointer validity model. Similarly, we do not worry that this model leaks memory for calls to c_str() since it is only used for ABC. It should be highlighted that due to the use of the efficient validity checker, we can simplify the model for the array bound checking model
class string {
  /* array bound model */
private: size_t size ;
public: ...
  const char *c_str const {
    char *res = new[size+1];    /* should not fail */
    char len(res)=size;        /* thus null-terminated */
    return (const char *) res;
  }
};

Fig. 7. Array bound model for the string::c_str method.

to only consider the size abstraction. Issues that are related to failed allocations
are, as mentioned before, relegated to the special purpose exception checker.

5 Experiments

We have implemented our methods in an in-house extension of CIL [31] called
CILpp, which handles C++ programs. We present a number of experiments on
some C and C++ benchmarks, and describe some of the previously unknown
bugs in C++ programs discovered by our analysis.

The models described thus far are able to find a wide variety of memory
related issues in C/C++ source code. Since the focus of this paper is on the
modeling of the interaction of C and C++ strings, we first present experiments
that target only this particular aspect. To do so, we have performed experiments
on open-source software packages that contain such interactions. Our analysis
is performed in a scope-bounded fashion [32,33]. A simple pre-processing tech-
nique is used to identify potential error sites. For the interaction analysis, these
are centered around calls to string library functions and error-prone functions
such as calls to the string::c_str() method. This enables us to choose a set
of objects and methods to be analyzed. We present a number of bugs that have
been uncovered by our experiments, thus far. As our tool is being improved, we
are applying our techniques to more open-source software.

Motivating example Recall the code fragment presented as Figure 1 in Section 4. The released version of the GNU bintools package at the time of the experi-
ments was v2.21 (official releases are available at [ftp.gnu.org/gnu/bintools]),
which was released in December 2010. The bug described earlier was already
present in v2.20 released in October 2009. Our tool discovered the bug in March
2011. The developers of the gold package confirmed this bug. However, the
developers have been aware of this bug internally about a month before our report.
A fix for this bug was finally released with v2.21.1 in June 2011.

Stale uses of c_str-created C-strings In our experiments, we found that
the issue of dangling pointer accesses due to stale uses of C++-to-C converted
strings is the main bug category of interest. We have found many incarnations
void I0::FixSlashes(char *str) {
    for (uint8 i=0; i<=strlen(str); i++) {
        if ((str[i]=='\' || str[i]=='/') &&
            str[i+1]=='\0') {
            str[i] = '\0'; // invalid write */
            return ;
        }
    }
    if (str[i] == '\0') return ;
}

void I0::FixPatches() {
    ... FixSlashes((char *)cfg->mysqlpath.c_str());
    FixSlashes((char *)cfg->wowpath.c_str());
}

Fig. 8. Invalid string manipulation

of this bug pattern in addition to the motivating example, which can be found using the validity-based abstraction model.

We have observed the same issue in a variety of other open-source benchmarks, including in unit tests for ICU4C (see icu-project.org/apiref/icu4c/), which provides portable unicode handling capabilities for software globalization requirements. Similarly, we noticed three uses of a dangling C-string pointer obtained through string::c_str() in Mosh, a fast interpreter for Scheme as specified in R6RS, which is the latest revision of the Scheme standard. After we informed the developers of this actively maintained project about these three dangling pointer violations, they have confirmed the issue and have fixed them in the source repository (see http://bit.ly/godw).

Manipulation of ownerMutable strings We also observed rare cases of direct string manipulation of C-strings obtained through c_str(). As discussed earlier, this is an explicit violation of the STL C++ string specification. Multiple such scenarios occurred in the datatramp project, one of which is shown in Figure 8.

Buffer overflows due to string conversions In our experiments, we have also observed rare cases of potential buffer overflows using strings obtained from a C++ string object. One such example is shown in Figure 3 which is from a library that transliterates text between different representations. Note that this warning awaits confirmation, since in our scope-bounded analysis we are not aware of any global constraint on the maximum size of a string to be converted.

Erlang/OTP Case Study Erlang (see erlang.org) is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Erlang’s runtime system has built-in support for concurrency, distribution and fault tolerance. OTP is a set of Erlang libraries pro-
char *convert(const char *in, mode_t mode,
    parse_func_t parse, output_func_t output) {
    static char buf[4096];
    ...
    std::ostringstream sout;
    (*output)(tokens, sout, mode);
    sout << '\0';
    std::strcpy(buf, sout.str().c_str()); /* buffer overflow */
    return buf;
}

Fig. 9. Potential buffer overflow

We analyzed relevant C and C++ source code of the current Erlang/OTP release R14B01 (December 2010). The Orber application is a CORBA compliant Object Request Broker (ORB), which provides CORBA functionality in an Erlang environment. Essentially, the ORB channels communication or transactions between nodes in a heterogeneous environment.

typedef std::stringstream STRINGSTREAM;
typedef std::stringbuf STRINGBUF;
void InitialReference::createIOR(
    STRINGSTREAM& byte, long length) {
    STRINGBUF *stringbuf;
    STRINGSTREAM string;
    int i;
    const char *c;
    const char *bytestr = byte.str().c_str();
    for(i = 0; c = bytestr; i<length; c++, i++){
        b = *c; /* invalid access */
        ...
    }
    delete bytestr; /* invalid call to delete */
    /* iorString is a member field */
    iorString = (char *)string.str().c_str();
}

Fig. 10. Erlang/OTP Orber application code

Figure 10 shows a part of the C++ source code for the InitialReference class in Orber. The code generates a reference for an Interoperable Object Reference (IOR), which simplifies the initial reference access from C++. However, the
C++ interface contains a number of invalid uses of C-strings from a C++ string object in the central createIOR method. Our analysis discovered the invalid access inside the for-loop, and also reported the invalid call to delete.

We analyzed the complete C++ code inside the Orber module. As is typical for C++, complexity of the analysis is increased due to standard header files. While the Orber module only contained about 300 LOC, the effective LOC after including the relevant headers is about 3k LOC. Our tool analyzed 7 functions of interest, and reported only the above 2 witnesses using the pointer and string object validity checker. The array bound checker did not find any witnesses in this case study. For one of the functions, the analysis using abstract interpretation and bounded model checking timed out (we limit the analysis for each function to 10 minutes). Overall, for 14 function and checker pairs, our tool reported over 40 property proofs, and spent about 20 minutes for the analysis.

However, our tool did not report a third issue, where a dangling pointer is assigned to the forString member field. We discovered this issue when inspecting neighboring code to reported warnings. This likely violation of the object invariant, that all member fields be pointing to valid memory regions, was not discovered since our scope-bounded analysis did not find a read of the forString field. In the future, we would like to extend our analysis to automatically check for object consistency after method invocations, in order to discover such issues.

**The c-icap project** The c-icap project is an open-source implementation of ICAP (Internet Content Adaptation Protocol), a protocol aimed at supporting HTTP content adaptation. ICAP allows arbitrary content-filtering and on-the-fly content modification. A common application running ICAP are anti-virus scanners, for example. The development of the c-icap project started in 2004, and the project is still actively maintained (see c-icap.sourceforge.net).

<table>
<thead>
<tr>
<th>Bug category</th>
<th>Checker</th>
<th>Reported</th>
<th>Known</th>
<th>Fixed</th>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL access</td>
<td>PVC</td>
<td>23</td>
<td>0</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>Memory leak</td>
<td>MLC</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Uninitialized condition</td>
<td>UUM</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Array underflow</td>
<td>ABC</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Partially initialized memory</td>
<td>UUM</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>34</td>
<td>1</td>
<td>32</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2. Experimental results for c-icap for various checkers (see Section 2.1).

We analyzed the complete c-icap project with our tool, by analyzing individual modules separately. The tool analyzed over 24k lines of source code written in C, which includes about 4k lines of header files. The complete analysis, in a scope-bounded fashion, using abstract interpretation and model checking for all checkers completes in a few hours. The full investigation of all witnesses found by the model checker took one expert user about 3 hours.
The experimental results are summarized in Table 2. The investigation yielded 34 unique bugs that were communicated to the developer of c-icap. 32 of the 34 reported issues have been fixed so far. Three of the reported bugs were deemed very important by the developer, including one deep inter-procedural NULL access. The two bugs that have not been fixed yet have been acknowledged as bugs as well, and are to be addressed in future releases. Further details are available at www.nec-labs.com/~ivancic/bugs/c-icap.htm.

The MeCab project The MeCab project provides a customizable Japanese morphological analyzer, which is applied to a variety of natural language processing tasks. Its source code (without any header files) contains 6.6k LOC of C++ code. A verification engineer discovered four bugs using this approach. This includes 3 paths with invalid NULL accesses, which were found using the pointer validity checker. Additionally, one uninitialized memory read was discovered.

6 Related Work

Buffer overflows can cause memory corruption which may be hard to detect instantly. Cowan et al. survey different buffer overflow attacks and some attempts at prevention and detection [18]. Static approaches use pointer analysis, range analysis and constraint solvers at various degrees of precision. Wagner et al. transform the overflow check elimination problem into one of solving interval constraints over integers [36]. Rugina and Rinard provide a powerful summary-based approach that reduces interval analysis problems into linear programming [29]. Many of the early approaches do not completely handle complications involving dynamic memory allocation, heap data-structures, array contents, type-casting, etc. Recently, there has been work on more comprehensive approaches, that handle many of the complications mentioned above [11,23,25]. However, we are not aware of any prior work on addressing buffer overflows due to the interaction of C++ and C string usage. Recently, size-based abstractions for strings have been proposed for other languages, such as PHP, as well [37].

The CSSV tool [22] implements a comprehensive approach to overflow detection of C code. It constructs a memory model that tracks pointer bounds, and string lengths of arrays. A precise region-based points-to analysis handles overlaps between strings. Our memory model is fundamentally similar to that of CSSV. By combining abstract interpretation with SAT-based model checking in a scope-bounded fashion [23], we obtain scalable analysis for programs that are much larger than those reported by Dor et al.

Our approach uses the theory of abstract interpretation [15] along with numerical domains such as Intervals [15], Octagons [28], Polyhedra [17] and other numerical domains of intermediate precision and complexity. Abstract interpretation has been used in tools such as PolySpace [3], Astére [5], and so on. These tools focus on checking embedded applications with special features such as simple aliasing, no dynamic allocation, simple control flow and no recursion. However, our approach is designed to be more general purpose. The CoVerity verifier [1] has also been successfully applied to large industrial and open-source
projects. From published reports, most uncovered defects pertain to static buffers and are intraprocedural. Our effort is more ambitious in nature; we focus on accurate memory modeling to detect more complex bugs. CodeSonar from GrammaTech [2] is another related commercial tool. Recently, the static analysis of STL container classes was proposed [21]. However, we are not aware of any tool that directly targets the interaction of C and C++ strings.

There have been past approaches to model check programs for buffer overflows using various model checking techniques. The CBMC tool due to Clarke et al. [14] uses SAT-based bounded model checking (BMC) to unroll a given program up to a fixed depth into a SAT problem, which is checked for the presence of a violation up to that depth [7]. Our tool uses SAT-based BMC at its backend. However, we also use abstract interpretation up front to vastly simplify the model and obtain a more scalable approach. Predicate abstraction using automatic counterexample-guided abstraction refinement (CEGAR) [23,27] has led to important tools such as SLAM [5], BLAST [21], and many others. These tools have been mainly used to find API usage violations. However, our own experience with predicate abstraction refinement suggests that for properties such as buffer overflows and strings, the automatic refinement leads to a large number of predicates and too many refinement iterations.

References