Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools*

Kendra Kratkiewicz
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108
Phone: 781-981-2931
Email: KENDRA@LL.MIT.EDU

Richard Lippmann
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108
Phone: 781-981-2711
Email: LIPPMANN@LL.MIT.EDU

ABSTRACT
A corpus of 291 small C-program test cases was developed to evaluate static and dynamic analysis tools designed to detect buffer overflows. The corpus was designed and labeled using a new, comprehensive buffer overflow taxonomy. It provides a benchmark to measure detection, false alarm, and confusion rates of tools, and also suggests areas for tool enhancement. Experiments with five tools demonstrate that some modern static analysis tools can accurately detect overflows in simple test cases but that others have serious limitations. For example, PolySpace demonstrated a superior detection rate, missing only one detection. Its performance could be enhanced if extremely long run times were reduced, and false alarms were eliminated for some C library functions. ARCHER performed well with no false alarms whatsoever. It could be enhanced by improving inter-procedural analysis and handling of C library functions. Splint detected significantly fewer overflows and exhibited the highest false alarm rate. Improvements in loop handling and reductions in false alarm rate would make it a much more useful tool. UNO had no false alarms, but missed overflows in roughly half of all test cases. It would need improvement in many areas to become a useful tool. BOON provided the worst performance. It did not detect overflows well in string functions, even though this was a design goal.

Categories and Subject Descriptors
D.2.4 [Software Engineering] Software/Program Verification, D.2.5 [Software Engineering] Testing and Debugging, K.4.4 [Computers and Society] Electronic Commerce Security.

General Terms
Measurement, Performance, Security, Verification.

Keywords
Security, buffer overflow, static analysis, evaluation, exploit, test, detection, false alarm, source code.

1. INTRODUCTION
Ideally, developers would discover and fix errors in programs before they are released. This, however, is an extremely difficult task. Among the many approaches to finding and fixing errors, static analysis is one of the most attractive. The goal of static analysis is to automatically process source code and analyze all code paths without requiring the large numbers of test cases used in dynamic testing. Over the past few years, static analysis tools have been developed to discover buffer overflows in C code.

Buffer overflows are of particular interest as they are potentially exploitable by malicious users, and have historically accounted for a significant percentage of the software vulnerabilities published each year [18, 20], such as in NIST's ICAT Metabase [9], CERT advisories [1], Bugtraq [17], and other security forums. Buffer overflows have also been the basis for many damaging exploits, such as the Sapphire/Slammer [13] and Blaster [15] worms.

A buffer overflow vulnerability occurs when data can be written outside the memory allocated for a buffer, either past the end or before the beginning. Buffer overflows may occur on the stack, on the heap, in the data segment, or the BSS segment (the memory area a program uses for uninitialized global data), and may overwrite from one to many bytes of memory outside the buffer. Even a one-byte overflow can be enough to allow an exploit [10]. Buffer overflows have been described at length in many papers, including [20], and many descriptions of exploiting buffer overflows can be found online.

This paper focuses on understanding the capabilities of static analysis tools designed to detect buffer overflows in C code. It extends a study by Zitser [20, 21] that evaluated the ability of several static analysis tools to detect fourteen known, historical vulnerabilities (all buffer overflows) in open-source software. The Zitser study first found that only one of the tools could analyze large, open-source C programs. To permit an evaluation, short, but often complex, model programs were extracted from the C programs and used instead of the original, much longer programs. Five static analysis tools were run on model programs with and without overflows: ARCHER [19], BOON [18], Splint [6, 12], UNO [8], and PolySpace C Verifier [14]. All use static analysis techniques, including symbolic analysis, abstract interpretation, model checking, integer range analysis, and inter-procedural analysis. Results were not encouraging. Only one of the five tools performed statistically better than random guessing. Not only did the tools fail to detect a significant number of overflows, but they also produced a large number of false alarms, indicating overflows where none actually existed. Equally discouraging were the confusion rates, reflecting the number of cases where a tool reports an error in both the vulnerable and patched versions of a program.

2005 Workshop on the Evaluation of Software Defect Detection Tools
2005,June 12, Chicago, IL

*This work was sponsored by the Advanced Research and Development Activity under Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Given the small number of model programs, and the fact that buffer overflows were embedded in complex code, it is difficult to draw conclusions concerning why the tools performed poorly. This paper describes a follow-on analysis of the five tools evaluated in the previous study. It's simpler but broader, and more diagnostic test cases are designed to determine specific strengths and weaknesses of tools. Although this research evaluated only static analysis tools, it provides a taxonomy and test suite useful for evaluating dynamic analysis tools as well.

2. BUFFER OVERFLOW TAXONOMY

Using a comprehensive taxonomy makes it possible to develop test cases that cover a wide range of buffer overflows and make diagnostic tool assessments. Zitser developed a taxonomy containing thirteen attributes [20]. This taxonomy was modified and expanded to address problems encountered with its application, while still attempting to keep it small and simple enough for practical application. The new taxonomy consists of twenty-two attributes listed in Table 1.

Table 1. Buffer Overflow Taxonomy Attributes

Attribute Number	Attribute Name
1	Write/Read
2	Upper/Lower Bound
3	Data Type
4	Memory Location
5	Scope
6	Container
7	Pointer
8	Index Complexity
9	Address Complexity
10	Length/Limit Complexity
11	Alias of Buffer Address
12	Alias of Buffer Index
13	Local Control Flow
14	Secondary Control Flow
15	Loop Structure
16	Loop Complexity
17	Asynchrony
18	Taint
19	Runtime Environment Dependence
20	Magnitude
21	Continuous/Discrete
22	Signed/Unsigned Mismatch

Details on the possible values for each attribute are available in [11], and are summarized below. For each attribute, the possible values are listed in ascending order (i.e. the 0 value first).

Write/Read: describes the type of memory access (write, read). While detecting illegal writes is probably of more interest in preventing buffer overflow exploits, it is possible that illegal reads could allow unauthorized access to information or could constitute one operation in a multi-step exploit.

Upper/Lower Bound: describes which buffer bound is violated (upper, lower). While the term "buffer overflow" suggests an access beyond the upper bound of a buffer, it is equally possible to underflow a buffer, or access below its lower bound (e.g. `buf[-1]`).

Data Type: indicates the type of data stored in the buffer (character, integer, floating point, wide character, pointer, unsigned character, unsigned integer). Character buffers are often manipulated with unsafe string functions in C, and some tools may focus on detecting overflows of those buffers; buffers of all types may be overflowed, however, and should be analyzed.

Memory Location: indicates where the buffer resides (stack, heap, data region, BSS, shared memory). Non-static variables defined locally to a function are on the stack, while dynamically allocated buffers (e.g., those allocated by calling a malloc function) are on the heap. The data region holds initialized global or static variables, while the BSS region contains uninitialized global or static variables. Shared memory is typically allocated, mapped into and out of a program's address space, and released via operating system specific functions. While a typical buffer overflow exploit may strive to overwrite a function return value on the stack, buffers in other locations have been exploited and should be considered as well.

Scope: describes the difference between where the buffer is allocated and where it is overrun (same, inter-procedural, global, inter-file/inter-procedural, inter-file/global). The scope is the same if the buffer is allocated and overrun within the same function. Inter-procedural scope describes a buffer that is allocated in one function and overrun in another function within the same file. Global scope indicates that the buffer is allocated as a global variable, and is overrun in a function within the same file. The scope is inter-file/inter-procedural if the buffer is allocated in a function in one file, and overrun in a function in another file. Inter-file/global scope describes a buffer that is allocated as a global in one file, and overrun in a function in another file. Any scope other than "same" may involve passing the buffer address as an argument to another function; in this case, the *Alias of Buffer Address* attribute must also be set accordingly. Note that the test suite used in this evaluation does not contain an example for "inter-file/global."

Container: indicates whether the buffer resides in some type of container (no, array, struct, union, array of structs, array of unions). The ability of static analysis tools to detect overflows within containers (e.g., overrunning one array element into the next, or one structure field into the next) and beyond container boundaries (i.e., beyond the memory allocated for the container as a whole) may vary according to how the tools model these containers and their contents.

Pointer: indicates whether the buffer access uses a pointer dereference (no, yes). Note that it is possible to use a pointer dereference with or without an array index (e.g. `*pBuf` or `(*pBuf)[10]`); the *Index Complexity* attribute must be set accordingly. In order to know if the memory location referred to by a dereferenced pointer is within buffer bounds, a code analysis tool must keep track of what pointers point to; this points-to analysis is a significant challenge.

Index Complexity: indicates the complexity of the array index (constant, variable, linear expression, non-linear expression, function return value, array contents, N/A). This attribute applies only to the user program, and is not used to describe how buffer accesses are performed inside C library functions.

Address Complexity: describes the complexity of the address or pointer computation (constant, variable, linear expression, non-linear expression, function return value, array contents). Again, this attribute is used to describe the user program only, and is not applied to C library function internals.

Length/Limit Complexity: indicates the complexity of the length or limit passed to a C library function that overruns the buffer (N/A, none, constant, variable, linear expression, non-linear expression, function return value, array contents). "N/A" is used when the test case does not call a C library function to overflow the buffer, whereas "none" applies when a C library function overflows the buffer, but the function does not take a length or limit parameter (e.g. `strcpy`). The remaining attribute values apply to the use of C library functions that do take a length or limit parameter (e.g. `strncpy`). Note that if a C library function overflows the buffer, the overflow is by definition inter-file/inter-procedural in scope, and involves at least one alias of the buffer address. In this case, the *Scope* and *Alias of Buffer Address* attributes must be set accordingly. Code analysis tools may need to provide their own wrappers for or models of C library functions in order to perform a complete analysis.

Alias of Buffer Address: indicates if the buffer is accessed directly or through one or two levels of aliasing (no, one, two). Assigning the original buffer address to a second variable and subsequently using the second variable to access the buffer constitutes one level of aliasing, as does passing the original buffer address to a second function. Similarly, assigning the second variable to a third and accessing the buffer through the third variable would be classified as two levels of aliasing, as would passing the buffer address to a third function from the second. Keeping track of aliases and what pointers point to is a significant challenge for code analysis tools.

Alias of Buffer Index: indicates whether or not the index is aliased (no, one, two, N/A). If the index is a constant or the results of a computation or function call, or if the index is a variable to which is directly assigned a constant value or the results of a computation or function call, then there is no aliasing of the index. If, however, the index is a variable to which the value of a second variable is assigned, then there is one level of aliasing. Adding a third variable assignment increases the level of aliasing to two. If no index is used in the buffer access, then this attribute is not applicable.

Local Control Flow: describes what kind of program control flow most immediately surrounds or affects the overflow (none, if, switch, cond, goto/label, setjmp/longjmp, function pointer, recursion). For the values "if", "switch", and "cond", the buffer overflow is located within the conditional construct. "Goto/label" signifies that the overflow occurs at or after the target label of a goto statement. Similarly, "setjmp/longjmp" means that the overflow is at or after a longjmp address. Buffer overflows that occur within functions reached via function pointers are assigned the "function pointer" value, and those within recursive functions receive the value "recursion". The values "function pointer" and "recursion" necessarily imply a global or inter-procedural scope, and may involve an address alias. The *Scope* and *Alias of Buffer Address* attributes should be set accordingly.

Control flow involves either branching or jumping to another context within the program; hence, only path-sensitive code analysis can determine whether or not the overflow is actually reachable. A code analysis tool must be able to follow function pointers and have techniques for handling recursive functions in order to detect buffer overflows with the last two values for this attribute.

Secondary Control Flow: has the same values as *Local Control Flow*, the difference being the location of the control flow construct. *Secondary Control Flow* either precedes the overflow or contains nested, local control flow. Some types of secondary control flow may occur without any local control flow, but some may not. The *Local Control Flow* attribute should be set accordingly.

The following example illustrates an `if` statement that precedes the overflow and affects whether or not it occurs. Because it precedes the overflow, as opposed to directly containing the overflow, it is labeled as secondary, not local, control flow.

```
int main(int argc, char *argv[])
{
  char buf[10];
  int i = 10;

  if (i > 10)
  {
    return 0;
  }

  /*  BAD  */
  buf[i] = 'A';

  return 0;
}
```

Only control flow that affects whether or not the overflow occurs is classified. In other words, if a preceding control flow construct has no bearing on whether or not the subsequent overflow occurs, it is not considered to be secondary control flow, and this attribute would be assigned the value "none."

The following example illustrates nested control flow. The inner `if` statement directly contains the overflow, and we assign the value "if" to the *Local Control Flow* attribute. The outer `if` statement represents secondary control flow, and we assign the value "if" to the *Secondary Control Flow* attribute as well.

```
int main(int argc, char *argv[])
{
  char buf[10];
  int i = 10;

  if (sizeof buf <= 10)
  {
    if (i <= 10)
    {
      /*  BAD  */
      buf[i] = 'A';
    }
  }

  return 0;
}
```

Some code analysis tools perform path-sensitive analyses, and some do not. Even those that do often must make simplifying approximations in order to keep the problem tractable and the

solution scalable. This may mean throwing away some information, and thereby sacrificing precision, at points in the program where previous branches rejoin. Test cases containing secondary control flow may highlight the capabilities or limitations of these varying techniques.

Loop Structure: describes the type of loop construct within which the overflow occurs (none, standard for, standard do-while, standard while, non-standard for, non-standard do-while, non-standard while). A "standard" loop is one that has an initialization, a loop exit test, and an increment or decrement of a loop variable, all in typical format and locations. A "non-standard" loop deviates from the standard loop in one or more of these areas. Examples of standard `for`, `do-while`, and `while` loops are shown below, along with one non-standard `for` loop example:

Standard `for` loop:

```
for (i=0; i<11; i++)
{
    buf[i] = 'A';
}
```

Standard `do-while` loop:

```
i=0;
do
{
    buf[i] = 'A';
    i++;
} while (i<11);
```

Standard `while` loop:

```
i=0;
while (i<11)
{
    buf[i] = 'A';
     i++;
}
```

A non-standard `for` loop:

```
for (i=0; i<11; )
{
    buf[i++] = 'A';
}
```

Non-standard loops may necessitate secondary control flow (such as additional if statements). In these cases, the *Secondary Control Flow* attribute should be set accordingly. Any value other than "none" for this attribute requires that the *Loop Complexity* attribute be set to something other than "not applicable."

Loops may execute for a large number or even an infinite number of iterations, or may have exit criteria that depend on runtime conditions; therefore, it may be impossible or impractical for static analysis tools to simulate or analyze loops to completion. Different tools have different methods for handling loops; for example, some may attempt to simulate a loop for a fixed number of iterations, while others may employ heuristics to recognize and handle common loop constructs. The approach taken will likely affect a tool's capabilities to detect overflows that occur within various loop structures.

Loop Complexity: indicates how many loop components (initialization, test, increment) are more complex than the standard baseline of initializing to a constant, testing against a constant, and incrementing or decrementing by one (N/A, none, one, two, three). Of interest here is whether or not the tools handle loops with varying complexity in general, rather than which particular loop components are handled or not.

Asynchrony: indicates if the buffer overflow is potentially obfuscated by an asynchronous program construct (no, threads, forked process, signal handler). The functions that may be used to realize these constructs are often operating system specific (e.g. on Linux, `pthread` functions; `fork`, `wait`, and `exit`; and `signal`). A code analysis tool may need detailed, embedded knowledge of these constructs and the O/S-specific functions in order to properly detect overflows that occur only under these special circumstances.

Taint: describes whether a buffer overflow may be influenced externally (no, argc/argv, environment variables, file read or stdin, socket, process environment). The occurrence of a buffer overflow may depend on command line or stdin input from a user, the value of environment variables (e.g. `getenv`), file contents (e.g. `fgets`, `fread`, or `read`), data received through a socket or service (e.g. `recv`), or properties of the process environment, such as the current working directory (e.g. `getcwd`). All of these can be influenced by users external to the program, and are therefore considered "taintable." These may be the most crucial overflows to detect, as it is ultimately the ability of the external user to influence program operation that makes exploits possible. As with asynchronous constructs, code analysis tools may require detailed modeling of O/S-specific functions in order to properly detect related overflows. Note that the test suite used in this evaluation does not contain an example for "socket."

Runtime Environment Dependence: indicates whether or not the occurrence of the overrun depends on something determined at runtime (no, yes). If the overrun is certain to occur on every execution of the program, it is not dependent on the runtime environment; otherwise, it is.

Magnitude: indicates the size of the overflow (none, 1 byte, 8 bytes, 4096 bytes). "None" is used to classify the "OK" or patched versions of programs that contain overflows. One would expect static analysis tools to detect buffer overflows without regard to the size of the overflow, unless they contain an off-by-one error in their modeling of library functions. The same is not true of dynamic analysis tools that use runtime instrumentation to detect memory violations; different methods may be sensitive to different sizes of overflows, which may or may not breach page boundaries, etc. The various overflow sizes were chosen with future dynamic tool evaluations in mind. Overflows of one byte test both the accuracy of static analysis modeling, and the sensitivity of dynamic instrumentation. Eight and 4096 byte overflows are aimed more exclusively at dynamic tool testing, and are designed to cross word-aligned and page boundaries.

Continuous/Discrete: indicates whether the buffer overflow jumps directly out of the buffer (discrete) or accesses consecutive elements within the buffer before overflowing past the bounds (continuous). Loop constructs are likely candidates for containing continuous overflows. C library functions that overflow a buffer while copying memory or string contents into it demonstrate continuous overflows. An overflow labeled as continuous should have the loop-related attributes or the Length Complexity attribute (indicating the complexity of the length or limit passed to a C library function) set accordingly. Some dynamic tools rely

on “canaries” at buffer boundaries to detect continuous overflows [5], and therefore may miss discrete overflows.

Signed/Unsigned Mismatch: indicates if the buffer overflow is caused by using a signed or unsigned value where the opposite is expected (no, yes). Typically, a signed value is used where an unsigned value is expected, and gets interpreted as a very large unsigned or positive value, causing an enormous buffer overflow.

This taxonomy is specifically designed for developing simple diagnostic test cases. It may not fully characterize complex buffer overflows that occur in real code, and specifically omits complex details related to the overflow context.

For each attribute (except for Magnitude), the zero value is assigned to the simplest or “baseline” buffer overflow, shown below:

```
int main(int argc, char *argv[])
{
  char buf[10];
  /*  BAD  */
  buf[10] = 'A';
  return 0;
}
```

Each test case includes a comment line as shown with the word “BAD” or “OK.” This comment is placed on the line before the line where an overflow might occur and it indicates whether an overflow does occur. The buffer access in the baseline program is a write operation beyond the upper bound of a stack-based character buffer that is defined and overflowed within the same function. The buffer does not lie within another container, is addressed directly, and is indexed with a constant. No C library function is used to access the buffer, the overflow is not within any conditional or complicated control flows or asynchronous program constructs, and does not depend on the runtime environment. The overflow writes to a discrete location one byte beyond the buffer boundary, and cannot be manipulated by an external user. Finally, it does not involve a signed vs. unsigned type mismatch.

Appending the value digits for each of the twenty-two attributes forms a string that classifies a buffer overflow, which can be referred to during results analysis. For example, the sample program shown above is classified as “00000000000000000100.” The single “1” in this string represents a “Magnitude” attribute indicating a one-byte overflow. This classification information appears in comments at the top of each test case file, as shown in the example below:

```
/* Taxonomy Classification: 00000000000000000000 */

/*
 * WRITE/READ                0   write
 * WHICH BOUND               0   upper
 * DATA TYPE                 0   char
 * MEMORY LOCATION           0   stack
 * SCOPE                     0   same
 * CONTAINER                 0   no
 * POINTER                   0   no
 * INDEX COMPLEXITY          0   constant
 * ADDRESS COMPLEXITY        0   constant
 * LENGTH COMPLEXITY         0   N/A
 * ADDRESS ALIAS             0   none
 * INDEX ALIAS               0   none
 * LOCAL CONTROL FLOW        0   none
 * SECONDARY CONTROL FLOW    0   none
 * LOOP STRUCTURE            0   no
 * LOOP COMPLEXITY           0   N/A
 * ASYNCHRONY                0   no
 * TAINT                     0   no
 * RUNTIME ENV. DEPENDENCE   0   no
 * MAGNITUDE                 0   no overflow
 * CONTINUOUS/DISCRETE       0   discrete
 * SIGNEDNESS                0   no
 */
```

While the Zitser test cases were program pairs consisting of a bad program and a corresponding patched program, this evaluation uses program quadruplets. The four versions of each test case correspond to the four possible values of the Magnitude attribute; one of these represents the patched program (no overflow), while the remaining three indicate buffer overflows of one, eight, and 4096 bytes denoted as minimum, medium, and large overflows.

3. TEST SUITE

A full discussion of design considerations in creating test cases is provided in [11]. Goals included avoiding tool bias; providing samples that cover the taxonomy; measuring detections, false alarms, and confusions; naming and documenting test cases to facilitate automated scoring and encourage reuse; and maintaining consistency in programming style and use of programming idioms.

Ideally, the test suite would have at least one instance of each possible buffer overflow that could be described by the taxonomy. Unfortunately, this is completely impractical. Instead, a “basic” set of test cases was built by first choosing a simple, baseline example of a buffer overflow, and then varying its characteristics one at a time. This strategy results in taxonomy coverage that is heavily weighted toward the baseline attribute values. Variations were added by automated code-generation software that produces C code for the test cases to help insure consistency and make it easier to add test cases.

Four versions of 291 different test cases were generated with no overflow and with minimum, medium, and large overflows. Each test case was compiled with gcc, the GNU C compiler [7], on Linux to verify that the programs compiled without warnings or errors (with the exception of one test case that produces an unavoidable warning). Overflows were verified using CRED, a fine-grained bounds-checking extension to gcc that detects overflows at run time [16], or by verifying that the large overflow caused a segfault. A few problems with test cases that involved complex loop conditions were also corrected based on initial results produced by the PolySpace tool.

4. TEST PROCEDURES

The evaluation consisted of analyzing each test case (291 quadruplets), one at a time using the five static analysis tools (ARCHER, BOON, PolySpace, Splint, and UNO), and collecting tool outputs. Tool-specific Perl programs parsed the output and determined whether a buffer overflow was detected on the line immediately following the comment in each test case. Details of

the test procedures are provided in [11]. No annotations were added and no modifications were made to the source code for any tool.

Since BOON does not report line numbers for the errors, automated tabulation cannot validate that the reported error corresponds to the commented buffer access in the test case file. Instead, it assumes that any reported error is a valid detection. Therefore, BOON detections and false alarms were further inspected manually to verify their accuracy, and some were dismissed (two detections and two false alarms) since they did not refer to the buffer access in question.

Special handling was required for PolySpace in cases where the buffer overflow occurs in a C library function. PolySpace reports the error in the library function itself, rather than on the line in the test case file where the function is called. Therefore, the results tabulator looks for errors reported in the called library function and counts those detections irrespective of the associated line number. Additionally, one test case involving wide characters required additional command-line options to work around errors reported when processing wctype.h.

5. RESULTS AND ANALYSIS

All five static analysis tools performed the same regardless of overflow size (this would not necessarily hold for dynamic analysis). To simplify the discussion, results for the three magnitudes of overflows are thus reported as results for "bad" test cases as a whole.

Table 2 shows the performance metrics computed for each tool. The detection rate indicates how well a tool detects the known buffer overflows in the bad programs, while the false alarm rate indicates how often a tool reports a buffer overflow in the patched programs. The confusion rate indicates how well a tool can distinguish between the bad and patched programs. When a tool reports a detection in both the patched and bad versions of a test case, the tool has demonstrated "confusion." The formulas used to compute these three metrics are shown below:

$$\text{detection rate} = \frac{\text{\# test cases where tool reports overflow in bad version}}{\text{\# test cases tool evaluated}}$$

$$\text{false alarm rate} = \frac{\text{\# test cases where tool reports overflow in patched version}}{\text{\# of test cases tool evaluated}}$$

$$\text{confusion rate} = \frac{\text{\# test cases where tool reports overflow in both bad and patched version}}{\text{\# test cases where tool reports overflow in bad version}}$$

As seen in Table 2, ARCHER and PolySpace both have detection rates exceeding 90%. PolySpace's detection rate is nearly perfect, missing only one out of the 291 possible detections. PolySpace produced seven false alarms, whereas ARCHER produced none. Splint and UNO each detected roughly half of the overflows. Splint, however, produced a substantial number of false alarms, while UNO produced none. Splint also exhibited a fairly high confusion rate. In over twenty percent of the cases where it properly detected an overflow, it also reported an error in the patched program. PolySpace's confusion rate was substantially lower, while the other three tools had no confusions. BOON's detection rate across the test suite was extremely low.

Table 2. Overall Performance on Basic Test Suite (291 cases)

Tool	Detection Rate	False Alarm Rate	Confusion Rate
ARCHER	90.7%	0.0%	0.0%
BOON	0.7%	0.0%	0.0%
PolySpace	99.7%	2.4%	2.4%
Splint	56.4%	12.0%	21.3%
UNO	51.9%	0.0%	0.0%

It is important to note that it was not necessarily the design goal of each tool to detect every possible buffer overflow. BOON, for example, focuses only on the misuse of string manipulation functions, and therefore is not expected to detect other overflows. It is also important to realize that these performance rates are not necessarily predictive of how the tools would perform on buffer overflows in actual, released code. The basic test suite used in this evaluation was designed for diagnostic purposes, and the taxonomy coverage exhibited is not representative of that which would be seen in real-world buffer overflows.

Figure 1 presents a plot of detection rate vs. false alarm rate for each tool. Each tool's performance is plotted with a single data point representing detection and false alarm percentages. The diagonal line represents the hypothetical performance of a random guesser that decides with equal probability if each commented buffer access in the test programs results in an overflow or not. The difference between a tool's detection rate and the random guesser's is only statistically significant if it lies more than two standard deviations (roughly 6 percentage points when the detection rate is 50%) away from the random guesser line at the same false alarm rate. In this evaluation, every tool except BOON performs significantly better than a random guesser. In Zitser's evaluation [20], only PolySpace was significantly better. This difference in performance reflects the simplicity of the diagnostic test cases.

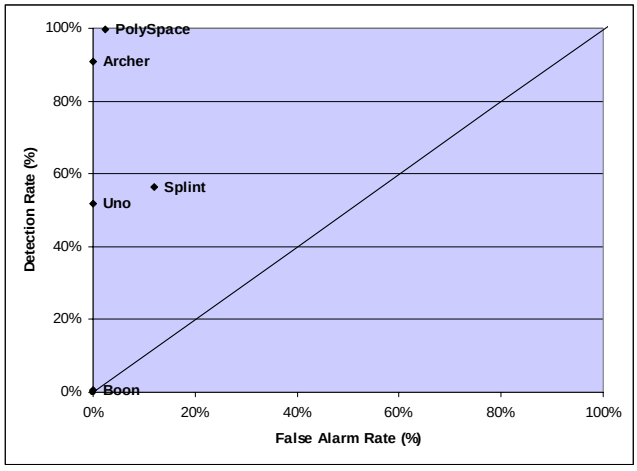

Figure 1. False Alarm and Detection Rates per Tool

Since PolySpace missed only one detection, and three of the other tools did detect the overflow in that test case, one could obtain perfect detection across the evaluation test suite by using PolySpace as the primary authority, and using one of the other tool's results only when PolySpace did not detect an overflow. ARCHER or UNO would be the best choice for this, as neither adds false alarms.

Similarly combining ARCHER and Splint would produce a detection rate of ninety-eight percent. ARCHER missed twenty-seven detections, and Splint detected all but five of those. Unfortunately, using Splint would also add thirty-five false alarms.

Table 3. Tool Execution Times

Tool	Total Time (secs)	Average Time per Test Case (secs)
ARCHER	288	0.247
BOON	73	0.063
PolySpace	200,820 (56 hrs)	172.526
Splint	24	0.021
UNO	27	0.023

Execution times for the five tools were measured as the total time to run each test case, including tool startup time, and are provided in Table 3. PolySpace's high detection rate comes at the cost of dramatically long execution times. ARCHER demonstrated both the second highest detection rate and the second highest execution time. Splint and UNO, with intermediate detection rates, had the two fastest execution times. BOON's slightly longer execution time did not result in a higher detection rate.

Some general observations can be made from inspecting the results as a whole. Missed detections and false alarms tend to group in certain attribute sets and follow logical patterns. If one tool missed a detection on a particular test case, usually some of the other tools missed it as well. For five test cases, only PolySpace did not miss detections in the bad programs. No attribute sets and no individual test cases have perfect detections across all five tools, but eight attribute sets contain no false alarms at all (Upper/Lower Bound, Data Type, Pointer, Alias of Buffer Index, Loop Structure, Loop Complexity, Asynchrony, and Signed/Unsigned Mismatch). Without the BOON results, looking exclusively at the results of the other four tools, three of the attribute sets (Write/Read, Data Type, and Alias of Buffer Index) and 108 individual test cases had perfect detections across the four tools. Complete and detailed results are presented in [11].

6. Detailed Tool Diagnostics

The following paragraphs discuss each tool's performance in detail, especially compared to the tools' design goals.

ARCHER's strategy is to detect as many bugs as possible while minimizing the number of false alarms. It is designed to be inter-procedural, path-sensitive, context-sensitive, and aware of pointer aliases. It performs a fully-symbolic, bottom-up data flow analysis, while maintaining symbolic constraints between variables (handled by a linear constraint solver). ARCHER checks array accesses, pointer dereferences, and function calls that take a pointer and size. It is hard-coded to recognize and handle a small number of memory-related functions, such as malloc [19].

The authors discuss many limitations of the current version of ARCHER. It does not handle function pointers, and imposes a five second limit on the analysis of any particular function. Furthermore, it loses precision after function calls, as it does not perform a proper inter-procedural side effects analysis and has a very simple alias analysis. It does not understand C library string functions, nor does it keep track of null pointers or the length of null-terminated strings. Its linear constraint solver is limited to handling at most two non-linearly related variables. Finally, some of the techniques it uses to reduce false alarms will necessarily result in missed detections. For instance, if no bounds information is known about a variable used as an array index, ARCHER assumes the array access is trusted and does not issue a warning. Similarly, it only performs a bounds check on the length and offset of a pointer dereference if bounds information is available; otherwise it remains quiet and issues no warning [19].

With close to a 91% detection rate and no false alarms, ARCHER performs well. Most of its twenty-seven missed detections are easily explained by its limitations. Twenty of these were inter-procedural, and this seems to be ARCHER's main weakness. The twenty inter-procedural misses include fourteen cases that call C library functions. While the authors admit to ignoring string functions, one might have expected memcpy() to be one of the few hard-coded for special handling. The other inter-procedural misses include cases involving shared memory, function pointers, recursion, and simple cases of passing a buffer address through one or two functions. Of the remaining seven misses, three involve function return values, two depend on array contents, and two involve function pointers and recursion.

While some of the missed detections occurred on cases whose features may not be widespread in real code (such as recursion), the use of C library functions and other inter-procedural mechanisms are surely prevalent. Indeed, ARCHER's poor performance in [20] is directly attributable to the preponderance of these features. ARCHER detected only one overflow in this prior evaluation, which was based on overflows in real code. Of the thirteen programs for which ARCHER reported no overflows, twelve contained buffer overflows that would be classified according to this evaluation's taxonomy as having inter-procedural scope, and nine of those involve calls to C library functions. To perform well against a body of real code, ARCHER needs to handle C library functions and other inter-procedural buffer overflows correctly.

BOON's analysis is flow-insensitive and context-insensitive for scalability and simplicity. It focuses exclusively on the misuse of string manipulation functions, and the authors intentionally sacrificed precision for scalability. BOON will not detect overflows caused by using primitive pointer operations, and ignores pointer dereferencing, pointer aliasing, arrays of pointers, function pointers, and unions. The authors expect a high false alarm rate due to the loss of precision resulting from the compromises made for scalability [18].

In this evaluation, BOON properly detected only two out of fourteen string function overflows, with no false alarms. The two detected overflows involve the use of strcpy() and fgets(). BOON

failed to detect the second case that calls strcpy(), all six cases that call strncpy(), the case that calls getcwd, and all four cases that call memcpy(). Despite the heavy use of C library string functions in [20], BOON achieved only two detections in that evaluation as well.

PolySpace is the only commercial tool included in this evaluation. Details of its methods and implementation are proprietary. We do know, however, that its approach uses techniques described in several published works, including: symbolic analysis, or abstract interpretation [2]; escape analysis, for determining inter-procedural side effects [4]; and inter-procedural alias analysis for pointers [3]. It can detect dead or unreachable code. Like other tools, it may lose precision at junctions in code where previously branched paths rejoin, a compromise necessary to keep the analysis tractable.

PolySpace missed only one detection in this evaluation, which was a case involving a signal handler. The PolySpace output for this test case labeled the signal handler function with the code "UNP," meaning "unreachable procedure." PolySpace reported seven false alarms across the test suite. These included all four of the taint cases, shared memory, using array contents for the buffer address, and one of the calls to strcpy(). The false alarm on the array contents case is not too surprising, as it is impractical for a tool to track the contents of every location in memory. PolySpace does not, however, report a false alarm on the other two cases involving array contents. The other six false alarms are on test cases that in some way involve calls to C library or O/S-specific function calls. Not all such cases produced false alarms, however. For instance, only one out of the two strcpy() cases produced a false alarm: the one that copies directly from a constant string (e.g., "AAAA"). Without more insight into the PolySpace implementation, it is difficult to explain why these particular cases produced false alarms.

PolySpace did not perform as well in Zitser's evaluation [20]. Again, without more knowledge of the tool's internals, it is difficult to know why its detection rate was lower. Presumably the additional complexity of real code led to approximations to keep the problem tractable, but at the expense of precision. The majority of the false alarms it reported in Zitser's evaluation were on overflows similar to those for which it reported false alarms in this evaluation: those involving memory contents and C library functions.

PolySpace's performance comes with additional cost in money and in time. The four other tools were open source when this evaluation was performed, and completed their analyses across the entire corpus in seconds or minutes. PolySpace is a commercial program and ran for nearly two days and eight hours, averaging close to three minutes of analysis time per test case file. This long execution time may make it difficult to incorporate into a code development cycle.

Splint employs "lightweight" static analysis and heuristics that are practical, but neither sound nor complete. Like many other tools, it trades off precision for scalability. It implements limited flow-sensitive control flow, merging possible paths at branch points. Splint uses heuristics to recognize loop idioms and determine loop bounds without resorting to more costly and accurate abstract evaluation. An annotated C library is provided, but the tool relies on the user to properly annotate all other functions to support inter-procedural analysis. Splint exhibited high false alarm rates in the developers' own tests [6, 12].

The basis test suite used in this evaluation was not annotated for Splint for two reasons. First, it is a more fair comparison of the tools to run them all against the same source code, with no special accommodations for any particular tool. Second, expecting developers to completely and correctly annotate their programs for Splint seems unrealistic.

Not surprisingly, Splint exhibited the highest false alarm rate of any tool. Many of the thirty-five false alarms are attributable to inter-procedural cases; cases involving increased complexity of the index, address, or length; and more complex containers and flow control constructs. The vast majority, 120 out of 127, of missed detections are attributable to loops. Detections were missed in all of the non-standard for() loop cases (both discrete and continuous), as well as in most of the other continuous loop cases. The only continuous loop cases handled correctly are the standard for loops, and it also produces false alarms on nearly all of those. In addition, it misses the lower bound case, the "cond" case of local flow control, the taint case that calls getcwd, and all four of the signed/unsigned mismatch cases.

While Splint's detection rate was similar in this evaluation and the Zitser evaluation [20], its false alarm rate was much higher in the latter. Again, this is presumably because code that is more complex results in more situations where precision is sacrificed in the interest of scalability, with the loss of precision leading to increased false alarms.

Splint's weakest area is loop handling. Enhancing loop heuristics to more accurately recognize and handle non-standard for loops, as well as continuous loops of all varieties, would significantly improve performance. The high confusion rate may be a source of frustration to developers, and may act as a deterrent to Splint's use. Improvements in this area are also important.

UNO is an acronym for uninitialized variables, null-pointer dereferencing, and out-of-bounds array indexing, which are the three types of problems it is designed to address. UNO implements a two-pass analysis; the first pass performs intra-procedural analysis within each function, while the second pass performs a global analysis across the entire program. It appears that the second pass focuses only on global pointer dereferencing, in order to detect null pointer usage; therefore, UNO would not seem to be inter-procedural with respect to out-of-bounds array indexing. UNO determines path infeasibility, and uses this information to suppress warnings and take shortcuts in its searches. It handles constants and scalars, but not computed indices (expressions on variables, or function calls), and easily loses precision on conservatively-computed value ranges. It does not handle function pointers, nor does it attempt to compute possible function return values. Lastly, UNO does not handle the setjmp/longjmp construct [8].

UNO produced no false alarms in the basic test suite, but did miss nearly half of the possible detections (140 out of 291), most of which would be expected based on the tool's description. This included every inter-procedural case, every container case, nearly every index complexity case (the only one it detected was the simple variable), every address and length complexity case, every address alias case, the function and recursion cases, every

signed/unsigned mismatch, nearly every continuous loop, and a small assortment of others. It performed well on the various data types, index aliasing, and discrete loops. Given the broad variety of detections missed in the basic test suite, it is not surprising that UNO exhibited the poorest performance in Zitser's evaluation [20].

7. CONCLUSIONS

A corpus of 291 small C-program test cases was developed to evaluate static and dynamic analysis tools that detect buffer overflows. The corpus was designed and labeled using a new, comprehensive buffer overflow taxonomy. It provides a benchmark to measure detection, false alarm, and confusion rates of tools, and can be used to find areas for tool enhancement. Evaluations of five tools validate the utility of this corpus and provide diagnostic results that demonstrate the strengths and weaknesses of these tools. Some tools provide very good detection rates (e.g. ARCHER and PolySpace) while others fall short of their specified design goals, even for simple uncomplicated source code. Diagnostic results provide specific suggestions to improve tool performance (e.g. for Splint, improve modeling of complex loop structures; for ARCHER, improve inter-procedural analysis). They also demonstrate that the false alarm and confusion rates of some tools (e.g. Splint) need to be reduced.

The test cases we have developed can serve as a type of litmus test for tools. Good performance on test cases that fall within the design goals of a tool is a prerequisite for good performance on actual, complex code. Additional code complexity in actual code often exposes weaknesses of the tools that result in inaccuracies, but rarely improves tool performance. This is evident when comparing test case results obtained in this study to results obtained by Zitser [20] with more complex model programs. Detection rates in these two studies are shown in Table 4. As can be seen, the two systems that provided the best detection rates on the model programs (PolySpace and Splint) also had high detection rates on test cases. The other three tools performed poorly on model programs and either poorly (BOON) or well (ARCHER and UNO) on test cases. Good performance on test cases (at least on the test cases within the tool design goals) is a necessary but not sufficient condition for good performance on actual code. Finally, poor performance on our test corpus does not indicate that a tool doesn't provide some assistance when searching for buffer overflows. Even a tool with a low detection rate will eventually detect some errors when used to analyze many thousands of lines of code.

Table 4. Comparison of detection rates with 291 test cases and with 14 more complex model programs in Zitser [20].

Tool	Test Case Detection	Model Program Detection [20]
ARCHER	90.7%	1%
BOON	0.7%	5%
PolySpace	99.7%	87%
Splint	56.4%	57%
UNO	51.9%	0.0%

The test corpus could be improved by adding test cases to cover attribute values currently underrepresented, such as string functions. It may also be used to evaluate the performance of dynamic analysis approaches. Anyone wishing to use the test corpus should send email to the authors.

8. ACKNOWLEDGMENTS

We would like to thank Rob Cunningham and Tim Leek for discussions, and Tim for help with getting tools installed and running. We also thank David Evans for his help with Splint, David Wagner for answering questions about BOON, Yichen Xie and Dawson Engler for their help with ARCHER, and Chris Hote and Vince Hopson for answering questions about C-Verifier and providing a temporary license.

9. REFERENCES

[1] CERT (2004). CERT Coordination Center Advisories, http://www.cert.org/advisories/, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA

[2] Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs, *Proceedings of the 2nd International Symposium on Programming*, Paris, France, 106--130

[3] Deutsch, A. (1994). Interprocedural may-alias analysis for pointers: beyond *k*-limiting, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, Florida, 230--241

[4] Deutsch, A. (1997). On the complexity of escape analysis, *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 358--371

[5] Etoh, H. (2004). GCC extension for protecting applications from stack smashing attacks, http://www.trl.ibm.com/projects/security/ssp/

[6] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis, *IEEE Software*, 19 (1), 42--51

[7] GCC Home Page (2004). Free Software Foundation, Boston, MA, http://gcc.gnu.org/

[8] Holzmann, G. (2002). UNO: Static source code checking for user-defined properties, Bell Labs Technical Report, Bell Laboratories, Murray Hill, NJ, 27 pages

[9] ICAT (2004). The ICAT Metabase, http://icat.nist.gov/icat.cfm, National Institute of Standards and Technology, Computer Security Division, Gaithersburg, MD

[10] klog (1999). The frame pointer overwrite, *Phrack Magazine*, 9 (55), http://www.tegatai.com/~jbl/overflow-papers/P55-08

[11] Kratkiewicz, K. (2005). Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code, Master's Thesis, Harvard University, Cambridge, MA, 285 pages

[12] Larochelle, D. and Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities, *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, 177--190

[13] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. (2003). The Spread of the Sapphire/Slammer Worm, http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html

[14] PolySpace Technologies (2003). PolySpace C Developer Edition, http://www.polyspace.com/datasheets/c_psde.htm, Paris, France

[15] PSS Security Response Team (2003). PSS Security Response Team Alert - New Worm: W32.Blaster.worm, http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/alerts/msblaster.asp, Microsoft Corporation, Redmond, WA

[16] Ruwase, O. and Lam, M. (2004). A practical dynamic buffer overflow detector, *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, 159--169

[17] Security Focus (2004). The Bugtraq mailing list, http://www.securityfocus.com/archive/1, SecurityFocus, Semantec Corporation, Cupertino, CA

[18] Wagner, D., Foster, J.S., Brewer, E.A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities, *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 3--17

[19] Xie, Y., Chou, A., and Engler, D. (2003). ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors, *Proceedings of the 9th European Software Engineering Conference/10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Helsinki, Finland, 327--336

[20] Zitser, M. (2003). Securing Software: An Evaluation of Static Source Code Analyzers, Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, 130 pages

[21] Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open-source code, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, CA, 97--106