A Buffer Overflow Benchmark for Software Model Checkers (Short Paper)

Kelvin Ku[*], Thomas E. Hart[†], Marsha Chechik
Department of Computer Science
University of Toronto
{kelvin,tomhart,chechik}@cs.utoronto.ca

David Lie
Dept. of Electrical and Computer Engineering
University of Toronto
lie@eecg.utoronto.ca

ABSTRACT

Software model checking based on abstraction-refinement has recently achieved widespread success in verifying API conformance in device drivers, and we believe this success can be replicated for the problem of buffer overflow detection. This paper presents a publicly-available benchmark suite to help guide and evaluate this research. The benchmark consists of 298 code fragments of varying complexity capturing 22 buffer overflow vulnerabilities in 12 open source applications. We give a preliminary evaluation of the benchmark using the SatAbs model checker.

Categories and Subject Descriptors: D.2.4 [**Software / Program Verification**]: Model Checking

General Terms: Security, Verification, Performance, Measurement.

Keywords: Buffer overflow, array bounds checking, benchmark, model checking.

1. INTRODUCTION

Buffer overflows are widespread in both legacy and modern systems, accounting for nearly half of all known security vulnerabilities [15]. They affect the security of programs written in languages that are neither type- nor memory-safe, such as C, enabling an attacker to gain control of a program and execute arbitrary code with elevated privileges. As a result, there has been much recent interest in developing static analysis tools to detect buffer overflows in C code, e.g., [8, 15, 17]. However, many of these tools suffer from poor precision and thus either report many false alarms or miss errors.

[*]Supported by MITACS.

[†]Supported by an NSERC Canada Graduate Scholarship and MITACS.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

Software model checking (SMC) is a family of automated techniques which have achieved success in finding errors in critical systems [14] and in verifying their absence [1]. A primary feature of SMCs is that they aim to provide sound analysis in both cases. As a result, a well-implemented SMC will neither give false alarms nor miss errors, but may instead fail to terminate on programs that are too complex for it to analyze. Software model checkers based on abstraction-refinement [1] (we call them CEGAR SMCs) have recently achieved widespread success in verifying API conformance in device drivers, and we believe this success can be replicated for the problem of finding buffer overflows or proving their absence; however, CEGAR SMCs typically do not yet perform well when checking code for buffer overflows [12].

To help us guide our own work in developing CEGAR SMCs for buffer overflows, we looked for a suitable benchmark. Such a benchmark can provide a set of examples on which tool developers can test new algorithms and a language for communicating research results, enabling quantitative measures as to how much improvement a new algorithm brings. In order to be useful, such a benchmark should be *realistic* (so that testcases correspond to buffer overflows found "in the wild") and at least partially solvable by existing tools, while still leaving room for improvement.

Zitser et al. [18] present a benchmark suite derived from real programs containing buffer overflow vulnerabilities. Their benchmark is realistic and contains a combination of correct and faulty programs. However, the technical, or "front-end", limitations prevent existing CEGAR SMC tools such as BLAST [9] and SatAbs [5] from running on the benchmark programs; the tools crash or behave unexpectedly on too many cases to yield useful results. In addition, algorithmic, or "back-end", limitations prevent the tools from analyzing the rest of the cases within reasonable resource limits. While the Zitser suite could be parameterized (for example, by buffer size, which significantly affects tool performance), the size and complexity of the examples make it difficult to determine which parts of a testcase are the analysis bottlenecks.

Kratkiewicz et al. [11] present a benchmark consisting of small synthetic programs with systematically varying syntactic constructs. Similarly, the benchmark of Wilander et al. [16] is comprised of a set of yet smaller programs, each containing a single call to a standard C library func-

tion. Both suites were designed as correctness tests for lightweight static analyses. Whereas Zitser's suite is too hard for current CEGAR SMCs, these suites are too easy, as we saw in our internal evaluation of BLAST and SatAbs on the Kratkiewicz benchmark; Chaki [2] reports similar results running ComFoRT on it. Barring implementation errors, each tool quickly terminated with a correct answer on every example. These benchmarks thus fail to challenge current CEGAR SMCs and, furthermore, lack realism and configurability.

Since neither approach was suitable for CEGAR SMCs, we have developed a special-purpose benchmark for buffer overflows, presented in this paper, in which we also report on a preliminary application of the benchmark to SatAbs.

The rest of the paper is organized as follows. After a short background on CEGAR SMCs in Section 2, we discuss and illustrate the benchmark construction methodology in Section 3, evaluate it in Section 4, and conclude in Section 5 with a discussion and an outline of future research directions.

2. SOFTWARE MODEL CHECKING

Model checking takes a state transition system representing some behaviors of a system, in our case, a program, and a property P to be checked against the system. In the case of buffer overflows, P is a property indicating that every access to a specified buffer is safe. Using standard compiler analyses, all pointer dereferences and array accesses are instrumented with bounds checks, so that the system goes to a line labelled `ERROR` if a bound is violated. For example, the instrumentation for the program in Figure 1(a) appears on lines 6 and 7. The buffer overflow problem then reduces to checking the reachability of the line labelled `ERROR`.

Models must be finite-state. *Counterexample-guided abstraction-refinement* (CEGAR) [3] is an automated technique for iteratively constructing a finite *predicate abstraction* [7] of a program in which concrete data (variable values) are replaced with *predicates* (boolean expressions) over program variables. It begins with an initial overapproximation of the system without any predicates. For example, Figure 1(b) shows the initial abstraction for the program in Figure 1(a); unknown values are represented by the special variable `NONDET`. The abstraction is then checked to reveal a *counterexample*, i.e., a path to `ERROR`, going through lines 1–7. Simulating this path on the concrete program shows that the counterexample is spurious: the first iteration of the loop cannot overflow `dest[]`. Learning from this, CEGAR adds a predicate `q = dest`. A new model tracking this predicate is built, and the process continues until either an answer is found or the system runs out of resources. The answer can be that `ERROR` is reachable, indicating an error, or that `ERROR` is not reachable, which corresponds to a proof that the program is free of buffer overflows. The system can run out of resources because the size of each successive model and the cost of its corresponding check are effectively exponential in the number of predicates used to build it.

For the example in Figure 1(a), to show that the `ERROR` is reachable, the model checker needs a separate predicate to track the relationship between `q` and the upper bound, `dest + 4`, in each of the five iterations of the loop: `q = dest`, `q = dest+1`, ..., `q = dest+4`. This is effectively unrolling the loop, so in this particular case, the running time of a CEGAR SMC is exponential in the size of the buffer, `dest`. This is not always the case, as we will see in Section 4.

```
(a)
1  char *src = "TOOBIG";
2  char dest[4];
3  char *p = src;
4  char *q = dest;
5  while (*p != '\0') {
6    if (q >= dest+4)
7      ERROR:;
8    *q++ = *p++;
9  }
```

```
(b)
1  char *src;
2  char dest[4];
3  char *p;
4  char *q;
5  while (NONDET) {
6    if (NONDET)
7      ERROR:;
8    ;
9  }
```

Figure 1: Basic buffer overflow example augmented with a bounds check. (a) The original program; (b) Initial CEGAR abstraction.

Program	Domain	# Vulns	# Testcases
Apache	Server	2	36
edbrowse	App	1	6
gxine	App	1	2
LibGD	Library	1	8
MadWifi	Driver	1	6
NetBSD libc	Library	1	24
OpenSER	Server	2	102
Samba	Server	1	4
SpamAssassin	App	1	2
BIND	Server	2	22
WU-FTPD	Server	3	24
Sendmail	Server	7	63

Table 1: Suite Composition.

3. BENCHMARK DESIGN

The benchmark is composed of testcases derived from a variety of buffer overflow vulnerabilities in open source programs, summarized in Table 1. We analyzed 22 vulnerabilities in 12 programs, producing 298 testcases (half of these are faulty versions and the other half are patched). Most of the vulnerabilities come from the Common Vulnerabilities and Exposures (CVE) database [6] while the rest appear in prior publications [12,18]. Different types of programs use buffers in different ways, so we selected programs from a variety of domains.

Scope. Our benchmark is designed to evaluate the ability of CEGAR SMCs to detect buffer overflows and verify their absence. Ultimately, we aim to use our benchmark to evaluate a variety of CEGAR tools employing a range of core algorithms and optimizations, e.g., [9, 10, 12, 13]. In the short term, however, we were limited by the availability of effective tools. For example, we found that the currently available releases of two popular CEGAR SMCs, SLAM [1] and BLAST [9], do not soundly model arrays. Aside from CBMC [4], we were unable to find any non-CEGAR tools, such as explicit-state SMCs, with adequate support for buffer overflow analysis. Our experiments in this paper are therefore limited to SatAbs [5]. However, we have done preliminary experiments, not reported in this paper, with CBMC and ComFoRT [2].

Methodology. We first examine the source code for each vulnerability and, after understanding the reason for the error and the corresponding patch, we identify the function in which the overflow occurs and slice away code outside its calling context. We parameterize all buffer size declarations so that we can control the bounds of buffer-dependent loops.

CEGAR generates predicates in order to track dependencies between data and control-flow. As such, we derive testcases from the source material by removing and simplifying these dependencies. Briefly, data-dependencies arise when a variable is used (evaluated) whereas control-dependencies arise from branch constructs.

Data-dependency simplifications include, for example, replacing pointer expressions with array indexing, thereby avoiding the generation of predicates relating a pointer to the buffer(s) into which it points. Other simplifications in this category include inlining functions, removing assignments, and performing constant and variable propagation.

Control-dependency simplifications include, for example, removing branch statements from a program to avoid the generation of branch condition predicates. Other such simplifications involve removing subexpressions in branch conditions and replacing functions whose return values are used in branch conditions with nondeterministic stubs.

The choice of simplifications to apply to a given program comes from our observations, as CEGAR SMC users and developers, of source code constructs that affect tool performance. However, we restricted the simplifications to those that preserve the intrinsic nature of the vulnerability. That is, the simplifications retain existing attack inputs, but may create new ones. For example, removing a branch which aborts execution if the input is malformed allows previously rejected input to produce an overflow. The simplification process continues until the programs are reduced to a form which we believe current CEGAR SMCs can effectively handle. We then review the generated testcases and remove redundant and uninteresting ones. Finally, we apply the official source code patch, possibly modified for compatibility with our simplifications, to obtain a safe variant of each testcase.

Each vulnerability is accompanied by the following documentation: a link to the original source code of the associated program, the file(s) in the original source code containing the vulnerability, the names of the source files of our testcases (listed in order of complexity), an explanation of how the vulnerability works and how the patch removes the vulnerability, and definitions of the simplifications used in each testcase. In general, each vulnerability required between one and four days for a single person to understand, slice, and process into testcases.

Example. Figure 2 shows an example buffer overflow we extracted from a module of the Apache web server (CVE-2006-3747). The function takes as input a Uniform Resource Indicator (URI), checks it for valid syntax, and extracts some tokens from the URI if the URI starts with `ldap://` (Lightweight Directory Access Protocol). The out-of-bounds write is on line 16: `c` is used to index into the array `token[]`, but the bounds checking is incorrect. Since `c` is incremented after the check `(c < TOKEN_SZ)`, `c` can be equal to `TOKEN_SZ` when it is subsequently used to index into `token[]`, thus exceeding array bounds. The patched version changes the check `(c < TOKEN_SZ)` on line 14 to `(c < TOKEN_SZ-1)`.

We constructed six cases containing vulnerabilities from this program, along with the corresponding six patched testcases. The first testcase is identical to Figure 2, modulo formatting. The second has a data-dependency simplification, replacing the pointer `cp` with an explicit integer index into the array `uri[]`. The four other testcases also use explicit array indexing and furthermore have control-dependency simplifications. The simplest testcase includes only the `while` loop in lines 14–19. The next simplest adds line 10. The other two omit line 10, but include line 3, and lines 3 and 6–9, respectively. All testcases preserve the original attack inputs while allowing more inputs to produce an overflow.

```
1  void escape_absolute_uri (char *uri, int scheme) {
2    char *cp; char *token[TOKEN_SZ];
3    if (scheme == 0 || strlen(uri) < scheme) return;
4    /* Skip past http://, mailto://, etc. */
5    cp = uri + scheme;
6    if (cp[-1] == '/') {
7      while (*cp != '\0' && *cp != '/') ++cp;
8      if (*cp == '\0' || *(++cp) == '\0') return;
9      scheme = cp - uri;
10     if (strncmp(uri, LDAP, LDAP_SZ) == 0) {
11       int c = 0;
12       /* Extract tokens into token[] */
13       token[0] = uri;
14       while (*cp != '\0' && c < TOKEN_SZ) {
15         if (*cp == '?') {
16           token[++c] = cp + 1; /* UNSAFE */
17           *cp = '\0';
18         } ++cp;
19       } return; } } }
```

Figure 2: Buffer overflow in Apache.

4. PRELIMINARY EVALUATION

The testcases were constructed to obtain a wide variation in performance when applied to a CEGAR SMC. To test how well they meet this requirement, we ran SatAbs [5] on each testcase in the suite. We chose SatAbs because it provides automatic instrumentation of potential buffer overflows and thorough handling of the C language, particularly arrays and pointer arithmetic which are heavily used in our testcases. We used version 1.6 with the default model checker, Cadence SMV. For the evaluation, we configured SatAbs to check all relevant buffer overflow assertions. We used 600s timeouts.

The tests were run with minimal buffer sizes, 1 and 2, in order to separate the effect of the simplifications from that of the loops. Analysis time quickly explodes as the buffer size increases, so in evaluating the testcases it was essential to limit the bounds of the buffer-dependent loops.

Some of the results are presented in Figure 3. Each bar indicates the difficulty of a testcase as the average number of predicates generated per overflow assertion (a testcase can contain more than one such assertion). The results for buffer size 1 are shown by the bottom bar; results for buffer size 2 are shown by the top bar and are cumulative. For example, the rightmost bar indicates that the most complex of the testcases shown for NetBSD generated 68 predicates at buffer size 1, and 108 predicates at buffer size 2. The bars are clustered by vulnerability and sorted from left to right by order of testcase difficulty. We only include the results for at most four of the simplest unpatched testcases from each vulnerability. Cases which failed to produce useful results (due to crashes and runtime exceeding the 600s timeout) are omitted from the chart. In total, at base buffer size 1, SatAbs found an overflow in 71.4% of our testcases.

The results show that, for each vulnerability, the testcase construction methodology produces a series of cases

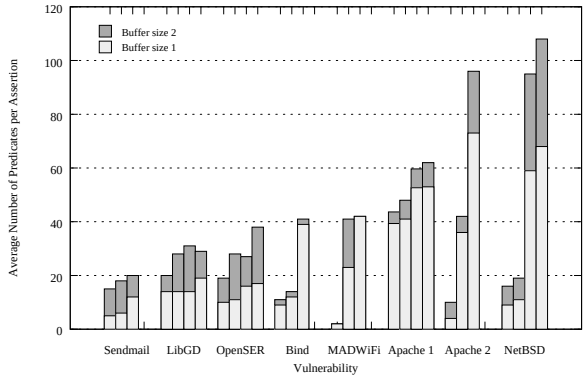

Figure 3: SatAbs performance variation over testcases with vulnerabilities, buffer sizes 1 and 2.

of gradually increasing difficulty. They also show that different vulnerabilities generate testcases with distinct difficulty bounds. For example, at buffer size 1, the most difficult case in MadWiFi requires only 42 predicates, whereas the comparable case in NetBSD needs 108. Finally, as expected, buffer-dependent loop bounds are shown in many cases to have a significant impact on analysis difficulty.

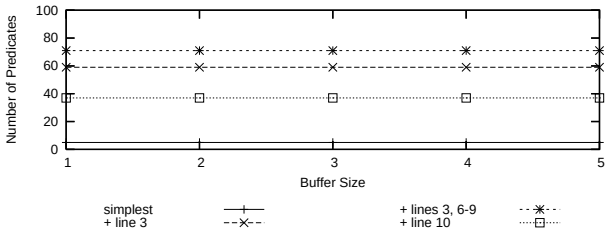

Figure 4: Plots for patched versions of testcases from the Apache example (Figure 2).

Interestingly, dependency on buffer size is less pronounced for the patched versions of the testcases which check a tool's ability to prove that a buffer overflow cannot occur. Figure 4 shows the number of predicates generated for the patched testcases from the Apache example in Figure 2, with one line representing each testcase, plotting the number of predicates generated as a function of the base buffer size. The lines are flat: once CEGAR finds the predicate `(c < TOKEN_SZ-1)`, it is able to prove that accesses to `token[]` are safe, regardless of the size of the buffer. However, the analysis remains sensitive to our simplifications, with the more complex testcases resulting in an increase in the number of predicates. The most complicated testcases (not shown) exceed the timeout even at small buffer sizes.

Unfortunately, this trend only holds for patched testcases in which buffer accesses are protected by inequality checks on pointers or array indices. In programs such as those using `strcpy()` to (safely) copy the contents of one array into a sufficiently large target array, the number of predicates needed to prove safety roughly equals the number needed to find an error, and grows linearly with target buffer size.

5. CONCLUSION

We have described a buffer overflow benchmark for CEGAR SMCs. In examining 22 buffer overflow vulnerabilities, we found that the code examples in which these overflows appear differ considerably—we could not have come up with similar examples synthetically. We believe this diversity will be very useful in evaluating new techniques.

Prior to starting this project, our naïve intuition had been that `strcpy()`-type loops were the "base" case for evaluating buffer overflow analysis, and CEGAR's poor performance on them discouraged us. We were pleasantly surprised to find, in the wild, buffer overflows in array accesses meant to be protected by inequality checks, and to see that CEGAR could efficiently verify their safe equivalents.

We plan to test our benchmark with other CEGAR SMCs, and with explicit-state and bounded model checkers. Since our simplifications were made with CEGAR in mind, they would need to be re-examined to appropriately evaluate these paradigms. We are continuing to add to the suite and we encourage others to add their examples as well. More information about the benchmark is available at `http://www.cs.toronto.edu/~kelvin/benchmark`.

6. REFERENCES

[1] T. Ball and S. Rajamani. "The SLAM Toolkit". In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 260–264, 2001.

[2] S. Chaki and S. Hissam. "Certifying the Absence of Buffer Overflows". Tech. Report CMU/SEI-2006-TN-030, SEI, 2006.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement". In *Proc. CAV'00*, volume 1855 of *LNCS*, pages 154–169, 2000.

[4] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In *Proc. TACAS'04*, volume 2988 of *LNCS*, pages 168–176, 2004.

[5] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. " SATABS: SAT-based Predicate Abstraction for ANSI-C ". In *Proc. TACAS'05*, volume 3440 of *LNCS*, pages 570–574, 2005.

[6] CVE — Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[7] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 72–83, 1997.

[8] B. Hackett, M. Das, D. Wang, and Z. Yang. "Modular Checking for Buffer Overflows in the Large". In *Proc. ICSE'06*, pages 232–241, 2006.

[9] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy Abstraction". In *Proc. POPL'02*, pages 58–70, 2002.

[10] R. Jhala and K. McMillan. "Array Abstractions from Proofs". In *Proc. CAV'07*, LNCS. Springer, 2007. to appear.

[11] K. Kratkiewicz and R. Lippmann. "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools". In *Proc. BUGS'05*, 2005.

[12] D. Kroening, A. Groce, and E. Clarke. "Counterexample Guided Abstraction Refinement via Program Execution". In *Proc. Int. Conf. on Formal Eng. Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 224–238, 2004.

[13] D. Kroening and G. Weissenbacher. "Counterexamples with Loops for Predicate Abstraction". In *Proc. CAV'06*, volume 4144 of *LNCS*, pages 152–165, 2006.

[14] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. "Model Checking Programs". *Journal of Automated Software Engineering*, 10(2), April 2003.

[15] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities". In *Proc. NDSS'00*, pages 3–17, 2000.

[16] J. Wilander and M. Kamkar. "A Comparison of Publicly Available Tools for Static Intrusion Prevention". In *Proc. 7th Nordic Workshop on Secure IT Systems*, pages 68–84, 2002.

[17] Y. Xie, A. Chou, and D. R. Engler. "ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors". In *Proc. ESEC/FSE-11*, pages 327–336, 2003.

[18] M. Zitser, R. Lippmann, and T. Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". In *Proc. SIGSOFT'04/FSE-12*, pages 97–106, 2004.