PtYasm: Software Model Checking with Proof Templates

Thomas E. Hart[*], Kelvin Ku[†], David Lie[‡], Marsha Chechik
Department of Computer Science
University of Toronto
{tomhart,kelvin,lie,chechik}@cs.toronto.edu

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University
arie@sei.cmu.edu

Abstract

We describe PtYasm*, an enhanced version of the* Yasm *software model checker which uses proof templates. These templates associate correctness arguments with common programming idioms, thus enabling efficient verification. We have applied* PtYasm *to the problem of verifying the safety of array accesses in programs derived from the Verisec suite.* PtYasm *is able to verify this property in the majority of testcases, while existing software model checkers fail to do so due to loop unrolling.*

1. Introduction

Software model checking based on predicate abstraction and refinement is a powerful and commercially successful [1] verification technique. Tools using this paradigm iteratively prune infeasible paths from a model of a program, in order to find an abstraction of the program in which the property of interest can be proven to hold.

Since software verification is undecidable in general, software model checkers (SMCs) are not guaranteed to find "good" abstractions, even when the program being analyzed uses common and well-understood idioms. In our companion paper [4], we advocate the use of *proof templates* in these instances. A proof template associates a correctness argument with a program fragment which uses a common programming idiom, thus helping an SMC find an efficient abstraction.

We have implemented PtYasm, an SMC which uses proof templates in the domain of array bounds checking. Existing SMCs often perform poorly in this domain, getting stuck "unrolling" loops in order to prove safety [5]. Using proof templates, PtYasm verifies the safety of common loops which keep an index in-bounds using (1) numerical comparisons, (2) sentinel null characters, or (3) by correlating updates with those of a second variable.

[*]Supported by an NSERC Canada Graduate Scholarship and MITACS
[†]Supported by MITACS
[‡]Department of Electrical and Computer Engineering

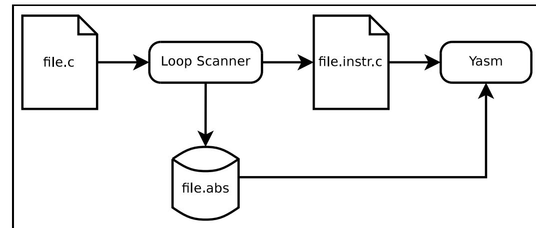

Figure 1. Architecture of PtYasm.

```
1  void foo () {
2    char src[1024], dest[1024];
3    char ch; int i=0, j=0;
4
5    src[1023] = '\0';
6    if (src[i] == '*') i++;
7
8    while (1) {
9      ch = src[i];
10     if (ch == '\0' || ch == ',') break;
11     assert (j < 1024);
12     dest[j] = ch;
13     j++;
14     i++;  } }
```

Figure 2. An array bounds checking example.

2. Implementation

PtYasm associates proof templates with loops in a program, and uses these templates to ensure that these loops keep array indices in-bounds. Our system uses four predefined templates, resulting from the combination of two conditions: whether an array index is kept in bounds via arithmetic comparisons or via tests on an array cell, and whether the test is on the same index which appears in the bounds check, or on some other variable. Each template is parameterized, and includes a set of *assumptions* which must be true before loop entry, and a method to prove that a bounds check cannot fail given that these assumptions hold.

Figure 1 shows the architecture of PtYasm, which has two components. The first component is the *loop scanner*, which scans a C file for loops in which proof templates may apply and derives parameters for these templates, and also instruments the program. The loop scanner consists of 2 KLOC of OCaml code, and is implemented as an exten-

```
⟨function='foo'⟩⟨loop='VERISEC_foo_line10_0'⟩⟨var='i'⟩⟨numvars='1'⟩⟨type='str'⟩⟨array='src'⟩
⟨function='foo'⟩⟨loop='VERISEC_foo_line10_0'⟩⟨var='j'⟩⟨numvars='2'⟩⟨type='str'⟩⟨leader='i'⟩⟨array='src'⟩
```

Figure 3. Output of loop scanner on example program in Figure 2.

sion to CIL [7], a tool for analyzing and transforming C code. The second component is a *software model checker* augmented with knowledge of proof templates. Our software model checker is an enhanced version of YASM [3], an SMC based on multi-valued model checking. YASM is written in Java, and uses the CUDD BDD library and the CVC Lite theorem prover. We changed or added approximately 2.8 KLOC in YASM in order to support proof templates. Our implementation is publicly available at `http://www.cs.toronto.edu/~tomhart/ptyasm`. Implementing PTYASM took approximately six months, including several rewrites.

We illustrate the operation of PTYASM using the example program in Figure 2, which copies characters from the string *src* into *dest*, and contains a bounds check on j on line 11. The loop scanner identifies loop *iterators* (roughly, variables whose value in one iteration is dependent on their value in a previous iteration) using standard compiler techniques such as use-def chains and dominators [6], and attempts to guess how an iterator may be bounded by examining the comparisons in loop exit branches. In this case, the iterators are i and j, and, since the exit branch on line 10 contains a test on *src*$[i]$, the loop scanner guesses that i is kept in bounds by *src*'s sentinel null character. Since no loop exit branch tests j, the loop scanner guesses that j is kept in bounds by following i. Figure 3 shows the output of the loop scanner, which records these two guesses.

The loop scanner also instruments the C file — for example, by adding metadata to keep track of the lengths of strings. YASM takes both this instrumented C file and the output of the loop scanner as inputs. YASM does not trust the output of the loop scanner, which may suggest inapplicable proof templates. If the loop scanner suggests an inapplicable template, YASM will be unable to prove that the bounds check cannot fail, and will backtrack, trying any other suggested templates. The loop scanner can thus aggressively use heuristic analyses without compromising the soundness of the overall analysis.

When YASM detects that it is unrolling the loop on lines 8–14 of Figure 2, it queries the output of the loop scanner for possible proof templates, finding the template suggested on line 2 of Figure 3. As directed by the template, YASM then adds a set of predicates to its abstraction of the program, and inserts a set of **assume** statements before the loop. These predicates and assumptions guide YASM towards the template proof. YASM then verifies the safety of the bounds checking assertion, and then discharges each assumption used. The details of the template appear in the companion paper [4].

3. Evaluation and Discussion

We have tested PTYASM on a set of 59 testcases derived from the Verisec suite [5], and compared its performance with that of YASM (without proof templates), BLAST, and SATABS. Within a timeout period of 10 minutes, PTYASM was able to verify 49 testcases, YASM (without proof templates) verifies 17, BLAST 19, and SATABS 22. Because it uses proof templates, PTYASM is able to verify many more of our testcases than the other SMCs. The complete details of our experiments appear in the companion paper [4], and the complete experimental data and test materials are available online at `http://www.cs.toronto.edu/~kelvin/ase08`.

PTYASM demonstrates a novel way to add programmer knowledge to SMCs, safely incorporating heuristic analyses without compromising soundness. The closest work to ours is that of Beyer et al. on path invariants [2], which tries to generate an abstraction of a loop given an *invariant template* provided by the user. They do not address how to conjecture these templates, consider strings, or separate the analysis of a loop's body from the assumptions about the paths leading to the loop. We suspect that our techniques and theirs may be usefully combined.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough Static Analysis of Device Drivers". In *Proc. EuroSys'06*, pp. 73–85, 2006.

[2] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. "Path Invariants". In *Proc. PLDI'07*, pp. 300–309, 2007.

[3] A. Gurfinkel, O. Wei, and M. Chechik. "YASM: A Software Model-Checker for Verification and Refutation". In *Proc. CAV'06*, LNCS 4144, pp. 170–174, 2006.

[4] T. E. Hart, K. Ku, M. Chechik, D. Lie, and A. Gurfinkel. Augmenting counterexample-guided abstraction refinement with proof templates. Submitted to ASE'08.

[5] K. Ku, T. E. Hart, M. Chechik, and D. Lie. "A Buffer Overflow Benchmark for Software Model Checkers". In *Proc. ASE'07*, pp. 389–392, 2007.

[6] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[7] G. Necula, S. McPeak, S. Rahul, and W. Weimer. "CIL: Intermediate Language and Tools for Analsysis and Transformation of C Programs". In *Proc. CC'02*, LNCS 2304, pp. 213–228, 2002.

Appendix

A. Availability and Maturity

We have made PtYasm available at `http://www.cs.toronto.edu/~tomhart/ptyasm/`. The website includes installation instructions. PtYasm requires Java, OCaml, CIL, ANTLR, and CVCL.

PtYasm is a very early research prototype. Although Yasm itself can correctly handle pointers and procedures, our extensions to support proof templates do not. These limitations are mostly due to the instrumentation added by the loop scanner, and can be remedied.

B. Presentation

We plan to present PtYasm by demonstrating its operation on a small set of testcases, showing the different stages of the analysis. We have four example testcases which are simplified versions of some testcases from our evaluation in the companion paper, for which PtYasm's analysis time is in the 10–90 second range. These examples are included in the PtYasm distribution on our webpage, under the `examples/` directory.

Figure 4 shows one of our examples; it is based on code in libgd. The goal is to make sure that *next* stays within bounds of the array *in*[], which is a string. Depending on which branches within the loop are taken, *next* can be incremented a variable number of times during a single loop iteration.

Running the loop scanner on the example program shown in Figure 4 produces the output shown in Figure 5. We are interested in the first line, which says that *next* is kept in bounds by the null terminator of the string *in*[]. The other line conjectures an argument as to how i is bounded in the loop, but since i is never used to index into a buffer, this line is ignored by Yasm.

Figure 6 shows a fragment of the instrumented version of the program in Figure 4, which we would show during the presentation. This excerpt shows an instance of the instrumentation the loop scanner adds to track string length. The variable *in_nullpos* represents a conservative approximation of the string length of *in*[]. In general, the loop scanner creates a variable *A_nullpos* for each array A in the program. Lines 5–12 of the listing in Figure 6 are an **assume(**p**)** statement, implemented as `if (!p) while(1) {}`. Thus, the effect of the instrumentation is, after a write of a null character to *in*[1024], to assume that *strlen*(*in*) $\leq$ 1024. This instrumentation is accompanied by the invariants that $in[in_nullpos]$ = '\0' and $in_nullpos \geq 0$, which Yasm assumes true at every program point.

Using the instrumented C file and the proof templates suggested by the loop scanner, Yasm is able to efficiently verify the safety of the assertion in this program. When Yasm detects loop unrolling, it queries the template database, finding the suggested template on line 1 of Figure 5. The information on this line parameterizes a template stored in Yasm, telling it to use the predicates $in[next]$ = '\0', $next \leq in_nullpos$, and $in_nullpos \leq next$ to abstract the loop, and to assume that before the loop is entered $in \leq in_nullpos$ and $in_nullpos \leq 1024$. Yasm adds these assumptions by adding **assume** statements to the program before the loop. The parameter 1024 comes from the bounds check assertion on line 11 of the original program.

These predicates lead to an efficient abstraction of the loop, since they let us prove that $next \leq in_nullpos$ is a loop invariant. Each increment of $next$ is guarded by a check on whether $in[next]$ = '\0', after which Yasm can conclude that $next \neq in_nullpos$, since $in[next] \neq$ '\0' $= in[in_nullpos]$. Hence, if $next \leq in_nullpos$ before the branch, and the branch in which $in[next] \neq$ '\0' is taken, we know that $next < in_nullpos$ (represented as $\neg(in_nullpos \leq next)$), and incrementing $next$ preserves the invariant that $next \leq in_nullpos$, since $\{next <$

```
1   int main ()
2   {
3     int next, encoding, i, ch, len;
4     char in [1024];
5     in [1023] = '\0';
6
7     encoding = NONDET();
8     if (encoding > 2 || encoding < 0) return 0;
9
10    for (next = 0, i=0; in[next] != '\0'; i++) {
11      assert (next < 1024);
12      ch = in[next];
13
14      if (ch == '\r' || ch == '\n'){
15        next++;
16        continue;
17      }
18
19      switch (encoding) {
20      case gdFTEX_Unicode:
21        len = 1;
22        next += len;
23        break;
24      case gdFTEX_Shift_JIS:
25        {
26          unsigned char c;
27          c = (unsigned char) in[next];
28          if (0xA1 <= c && c <= 0xFE) {
29            next++;
30          }
31        }
32        break;
33      case gdFTEX_Big5:
34        ch = (in[next]);
35        next++;
36        break;
37      }
38    }
39
40    return 0;
41  }
```

Figure 4. An array bounds checking example based on code from libgd.

```
⟨function='main'⟩⟨loop='VERISEC_gd_line40_0'⟩⟨var='next'⟩⟨numvars='1'⟩⟨type='str'⟩⟨array='in'⟩
⟨function='main'⟩⟨loop='VERISEC_gd_line40_0'⟩⟨var='i'⟩⟨numvars='2'⟩⟨type='str'⟩⟨leader='next'⟩⟨array='in'⟩
```

Figure 5. Output of loop scanner when run on example in Figure 4.

```
1   {
2   in[1024] = (char)0;
3   {
4   in_nullpos = "NONDET";
5   if (! (in_nullpos <= 1024)) {
6     VERISEC_ASSUMPTION_1: /* CIL Label */
7     while (1) {
8
9     }
10  } else {
11
12  }
13  }
14  }
```

Figure 6. Excerpt from instrumented version of program in Figure 4, showing string instrumentation which the loop scanner adds automatically. A second preprocessing phase replaces the string "NONDET" with the identifier YASM_NONDET**.**

$in_nullpos\}$ $next := next + 1$ $\{next \leq in_nullpos\}$.

After proving the safety of the original assertion, YASM discharges the assumptions used by turning the **assume** statements into assertions, and restarting the model checking process to prove that these assertions cannot fail. Figure 7 shows the final output of PTYASM on this example, including the predicates needed to prove each assertion. The total analysis time is just over six seconds, making this program a good example for a live demonstration.

The version of YASM without proof templates cannot verify this example efficiently, as it gets stuck unrolling the loop. In our experiments, BLAST and SATABS suffered from the same problem. Our presentation would include a demonstration of YASM's failure to verify this example without proof templates — the user can see that YASM is getting stuck loop unrolling, because it keeps generating predicates of the form $in[next]$ = '\0', $in[next + 1]$ = '\0', $in[next+2]$ = '\0', ... and $next < 1024, next+1 < 1024, next + 2 < 1024, \ldots$.

PtYasm : gd.c

File Edit View Terminal Tabs Help

```
Done in 6.293s



FINAL RESULT
        PROPERTY: EF (pc = ERROR)
        RESULT: false

Predicates to prove EF (pc = VERISEC_gd_line40_0_PRECONDITION2): 2
[(main::next <= main::in_nullpos), (main::in_nullpos <= 1024)]

Predicates to prove EF (pc = ERROR): 9
[(main::next < 1025), ((main::next + 1) < 1025), (main::in[(main::next + 1)] = 0), (main::ch =
 13), (main::in[main::next] = 13), (main::next <= main::in_nullpos), (main::in_nullpos <= main
::next), (main::in_nullpos <= 1024), (main::in[main::next] = 0)]

Predicates to prove EF (pc = VERISEC_gd_line40_0_PRECONDITION1): 2
[(main::next <= main::in_nullpos), (main::in_nullpos <= 1024)]

Total number of predicates: 13 (9 unique)
$
```

Figure 7. Final output of PtYasm on example in Figure 4. `in_nullpos` **is a metadata variable which keeps track of** $strlen(in)$**.**