PTYASM: Software Model Checking with Proof Templates

Thomas E. Hart*, Kelvin Ku*, Arie Gurfinkel†, Marsha Chechik*, David Lie‡
*Department of Computer Science, University of Toronto, {tomhart,kelvin,chechik}@cs.utoronto.ca
†Software Engineering Institute, Carnegie Mellon University, arie@sei.cmu.edu
‡Department of Electrical and Computer Engineering, University of Toronto, lie@eecg.utoronto.ca

***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 used PTYASM to verify 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.**

I. INTRODUCTION

Software model checkers (SMCs) based on predicate abstraction and refinement are powerful and commercially successful verification tools [1]. SMCs 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. Unfortunately, due to the undecidability of software verification in general, SMCs can perform poorly even when the analyzed program uses common and well-understood idioms. In our companion paper [2], 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 proof.

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 [3]. 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.

II. 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 pre-defined 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.

```
1  void example1 () {
2    char src[1024];
3    char dest[1024];
4    char ch; int i=0, j=0;
5
6    src[1023] = '\0';
7    if (src[i] == '*') i++;
8
9    while (1) {
10     ch = src[i];
11     if (ch == '\0'
12         || ch == ',')
13       break;
14     assert (j < 1024);
15     dest[j] = ch;
16     j++;
17     i++;  } }
```

(a)

```
1  void example2 () {
2    char buf[1024], c;
3    int len=1023, i=0;
4    int tmp;
5
6    while (1) {
7      c = NONDET;
8      if (i == len) return;
9      if (c == '\\') {
10       i++;
11       if (i == len)
12         return; }
13     else if (c == '.')
14       break;
15     i++; }
16   tmp = i+1;
17   assert (tmp < 1024); }
```

(b)

Fig. 1. Bounds checking examples based on (a) Sendmail and (b) Apache.

PTYASM has two components: a *loop scanner* and an SMC. The loop scanner searches a C file for loops in which proof templates may apply and derives parameters for these templates, and also instruments the program. It is implemented as a CIL [4] extension consisting of 2 KLOC of OCaml code. The loop scanner's output is passed to the SMC, which is a version of YASM [5] that has been enhanced with knowledge of proof templates. YASM is based on multi-valued model checking, 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. PTYASM currently supports arrays, but not pointers or procedures. We illustrate the operation of PTYASM using the example programs in Figure 1. The concepts are explored more fully in our companion paper [2].

Loop scanner. The loop scanner suggests proof templates and their parameters. It uses standard compiler techniques such as use-def chains and dominators [6] to identify loop *iterators* and expressions *derived from* them. An iterator, roughly, is a variable whose value in one loop iteration is dependent on its value in a previous iteration. An expression is derived from an iterator if it is semantically equivalent to a linear expression over the iterator and loop constants.

The loop scanner chooses which template(s) to use by examining *exit branches* and assertions. The hypothesis is that exit branches bound loop iterators. Assertions can be within

```
⟨function='example1'⟩⟨loop='VERISEC_example1_line9_0'⟩⟨var='i'⟩⟨numvars='1'⟩⟨type='str'⟩⟨array='src'⟩
⟨function='example1'⟩⟨loop='VERISEC_example1_line9_0'⟩⟨var='j'⟩⟨numvars='2'⟩⟨type='str'⟩⟨leader='i'⟩⟨array='src'⟩
```

Fig. 2. Output of loop scanner on example program in Figure 1(a).

Program Statement	Instrumentation

`A[i] = e`	`if ((e == 0) && (i >= 0)) {`
	`  A_nullpos = NONDET;`
	`  assume (A_nullpos <= i);`
	`} else if (i == A_nullpos) {`
	`  A_nullpos = NONDET;`
	`  assume (A_nullpos > i);`
	`}`

`if (A[i] == e)`	`if (e == 0) {`
	`  assume (A_nullpos <= i);`
	`}`

Fig. 3. Program instrumentation for string reads and writes; NONDET represents non–deterministic assignment.

the body of a loop (as in line 14 of Figure 1(a)), or dominated by a loop (as in line 17 of Figure 1(b)), in which case derived expressions can only use those loop constants which are not defined between the loop and the assertion. In our example programs, the exit branches are lines 11–12 of Figure 1(a), and lines 8, 11, and 13 of Figure 1(b). The iterators in Figure 1(a) are i and j, and, since the exit branch on lines 11–12 contains a test on *ch*, which is derived from *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 2 shows the output of the loop scanner, which records these two guesses. Similarly, in Figure 1(b), the assertion on line 17 is on *tmp*, which is derived from the iterator i, and i is bounded by *len* in the exit branches on lines 8 and 11.

The loop scanner adds instrumentation to the analyzed file to facilitate the templates. We use a *length abstraction* [7] to enable the use of *strlen* in predicates: each array A has an associated variable $A\_nullpos$ which represents an upper approximation of *strlen*(A), and is updated by the the instrumentation shown in Figure 3. By the definition of *strlen*, SMCs can assume $A\_nullpos \geq 0$ and $A[A\_nullpos] =$ '\0' as invariants throughout the program. The loop scanner also adds instrumentation to record the values of iterators at the start of each loop, in order to facilitate two-variable templates.

SMC. YASM takes both the instrumented C file and the output of the loop scanner as inputs. YASM does not depend on the correctness of the loop scanner: if the loop scanner suggests a template which does not help YASM prove that a bounds check cannot fail, it 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 9–17 of Figure 1(a), it queries the output of the loop scanner for possible proof templates, finding the template suggested on line 2 of Figure 2. 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. YASM works similarly on the example in Figure 1(b).

III. Evaluation and Discussion

We have tested PTYASM on a set of 59 testcases derived from the Verisec suite [3], 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) verified 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 [2].

PTYASM demonstrates a novel way to add programmer knowledge to SMCs, safely incorporating heuristic analyses without compromising soundness. Outside the context of SMCs, Denney and Fischer [8] used a pattern-based approach to guide Hoare-style certification, but their scope is restricted to programs automatically generated by AUTOBAYES and AUTOFILTER. Beyer et al. augment an SMC with template-based path invariants [9] to better handle loops, but 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*, 2006, pp. 73–85.

[2] T. E. Hart, K. Ku, M. Chechik, D. Lie, and A. Gurfinkel, "Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates," in *Proc. ASE'08*, 2008.

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

[4] 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*, ser. LNCS 2304, 2002, pp. 213–228.

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

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

[7] N. Dor, M. Rodeh, and S. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," in *Proc. PLDI '03*, 2003, pp. 155–167.

[8] E. Denney and B. Fischer, "Annotation Inference for Safety Certification of Automatically Generated Code (Extended Abstract)," in *Proc. ASE '06*, pp. 265–268.

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