Augmenting Counterexample-Guided Abstraction Refinement with Proof Templates

Technical Report CSRG-581

Thomas E. Hart[1], Kelvin Ku[2], Marsha Chechik
Deptartment of Computer Science
University of Toronto
{tomhart,kelvin,chechik}@cs.toronto.edu

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

David Lie
Department of Electrical and Computer Engineering
University of Toronto
lie@eecg.cs.toronto.edu

July 16, 2008

[1]Supported by an NSERC Canada Graduate Scholarship and MITACS
[2]Supported by MITACS

Abstract

Buffer overflow vulnerabilities are the result of programing errors that allow out-of-bounds writes to arrays. Verifying the safety of array writes is thus vital to ensuring program security. However, existing software model checkers based on abstraction-refinement perform poorly at this task, resulting in analyses which often depend on array size.

We observe that many of these analyses can be made efficient by providing *proof templates*, which specify a modular proof strategy with predicates and assumptions to use and then discharge. Our proof templates, which are associated with common programming idioms, guide the model checker towards proofs that are independent of array size.

We have integrated this technique into our software model checker, PTYASM, and have evaluated our approach on a set of testcases derived from the Verisec suite, demonstrating that our technique enables verification of the safety of array accesses independently of array size.

1 Introduction

Software model checking based on predicate abstraction and counterexample-guided abstraction refinement (CEGAR) has been shown to be effective for checking correctness of highly non-trivial programs [2] and is now part of commercial tools such as SDV [25]. The paradigm is very powerful since the abstraction – a set of predicates – is improved dynamically, based on identification of infeasible counterexamples. The process continues until the abstraction is sufficiently precise to prove the property of interest. In practice, the power of CEGAR software model checkers (henceforth referred to as SMCs) is limited by their ability to choose predicates well. Perfectly selecting predicates is impossible due to the undecidability of software verification, so this process always relies on heuristics.

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

Figure 1: An array bounds checking example based on code from Sendmail.

We aim to apply SMCs to verifying the absence of buffer overflows, which are a major threat to the security of C programs. Buffer overflows occur when it is possible to write to out-of-bounds array indices, thus allowing attackers to overwrite program control data. Current runtime defenses either offer incomplete protection [31], or add high performance overhead [29], whereas the ability to verify the absence of buffer overflows statically, if made practical, offers complete protection without any runtime overhead.

Figure 1 shows a string processing routine, simplified from a function in Sendmail which was patched after a buffer overflow was found in it[1]. The loop traverses the null-terminated array $src[]$ using i, and selectively copies characters into the array $dest[]$ using j. Current SMCs do poorly on verifying this example and others like it, finding abstractions with a number of predicates dependent on the size of the array $src[]$.

The goal of this paper is to improve the common-case performance of SMCs for verifying the absence of buffer overflows. Specifically, we aim to create abstractions which are independent of the sizes of the arrays being checked. Our solution is to define *proof templates* designed to work when a program uses common idioms to traverse an array, and attempt to guide an SMC towards these proofs automatically. The templates are modular, separating the proof of what must be true before a loop is entered from the analysis of the loop's body. We have implemented an algorithm to heuristically map ⟨*loop, variable*⟩ pairs in a program to proof templates. When the algorithm detects that a template may apply, it guides our SMC towards the template proof by supplying it with a set of predicates and assumptions. If the SMC is able to prove the original property using these predicates and assumptions, it then proceeds to discharge the assumptions (prove them true). If any stage of the analysis fails, the SMC backtracks to an earlier stage, making the overall process *sound*, despite the unsoundness of our algorithm for suggesting proof templates.

The contributions of this paper are as follows:

1. We describe proof templates corresponding to common array traversal idioms and an algorithm to heuristically identify when these templates may apply.

2. We describe an implementation of this technique, PTYASM, which is an enhanced version of the YASM SMC [16].

3. We evaluate our technique on a set of testcases derived from the Verisec suite [24], comparing our implementation with other state-of-the-art SMCs.

[1] From CVE-2003-0681 [12].

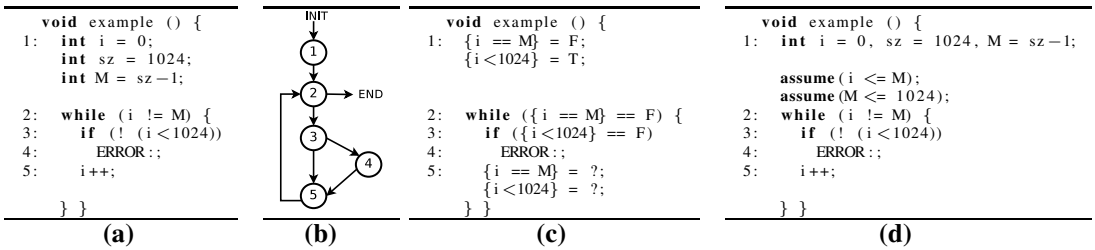

Figure 2: (a) An example program; (b) its control-flow graph; (c) its predicate abstraction; (d) assumptions added to its internal representation.

The rest of this paper is organized as follows. After giving the necessary background on SMCs and describing the architecture of our solution in Section 2, we introduce our proof templates for array traversals in Section 3 and report on the evaluation of our technique's effectiveness in Section 4. We conclude the paper with a comparison with related work (Section 5), and an outline of future research directions (Section 6).

2 Overview

In this section, we describe the CEGAR algorithm used by existing SMCs (Algorithm 1), and our extension, CEGAR+PT, which incorporates the use of proof templates (Algorithm 2).

2.1 CEGAR Software Model Checking

Algorithm 1 CEGAR Software Model Checking

```
1:  procedure CEGAR(P, ψ)                          ▷ Program, Property
2:      E ← ∅                                      ▷ Predicates
3:      loop
4:          M ← ABSTRACT (P, ψ, E)                 ▷ M = model
5:          τ ← MODELCHECK (M, ψ)
6:          if τ = ϵ then                          ▷ No path to ERROR
7:              return SAFE
8:          else
9:              if SPURIOUS(τ, P) then             ▷ Refine abstraction
10:                 E ← E ∪ REFINE (τ)
11:             else
12:                 return UNSAFE
```

CEGAR (Algorithm 1) gives a high-level description of the process used by SMCs. The algorithm checks a property ψ on a program P, where ψ is the reachability of a given line of P, labelled "ERROR". CEGAR can be used to check assertions by transforming a statement **assert**(p) into a conditional, such as the one on lines 3–4 of Figure 2(a), and checking whether the assertion's failure branch is unreachable. In that case, the assertion is considered to be *safe*. To verify the absence of buffer overflows using an SMC, one needs to instrument a program with assertions at each array access (which can be done automatically by an external tool [29]), and individually verify the safety of each assertion.

The ABSTRACT phase creates a finite model $\mathcal{M}$ which conservatively approximates P, by modelling the data state at each location in P's control flow graph using a set E of atomic predicates [3], such as "$x \leq y$" and "$x + 1 \leq A[z]$". Since the initial set of predicates is empty (line 2 of CEGAR), the initial abstraction is P's control flow graph. The MODELCHECK phase uses efficient search algorithms [9] to find a path to ERROR in $\mathcal{M}$. This phase returns a counterexample τ, or the special value ϵ if ERROR is unreachable in $\mathcal{M}$. If there is no counterexample, ψ holds on P, and the program is declared to be SAFE. Otherwise, if τ represents a possible execution of P, the program is declared to be UNSAFE. Finally, if τ is spurious, the set of predicates is augmented by a refinement phase which attempts to rule out τ by building

a more precise model of P. The process continues iteratively until the either of the definite answers (SAFE or UNSAFE) are obtained, or the analysis procedure runs out of resources.

We illustrate CEGAR on the example program in Figure 2(a), which implements a simple array traversal. Model-checking the control-flow graph of this program (shown in Figure 2(b)) yields the counterexample path $\tau_1 = \langle 1, 2, 3, 4 \rangle$, on which ERROR is reached on the first loop iteration. Since τ_1 is spurious, the REFINE phase searches for some set of predicates which, if added to E, would eliminate τ_1 from the abstract model. Predicate generation schemes are heuristic, and differ between tools [4, 7, 18], but typically return only the predicates needed to eliminate the *one* spurious path found. Thus, a typical implementation may return the predicates $i = M$ and $i < 1024$. Abstracting the program using these predicates yields the abstraction in Figure 2(c). Note that these predicates are not expressive enough to represent the effect of the statement `i++` on line 5 of Figure 2(a); thus, their values are unknown at the end of the first iteration of the loop beginning on line 2 (denoted by "?" on line 5 of Figure 2(c)). The next run of MODELCHECK thus finds the path $\tau_2 = \langle 1, 2, 3, 5, 2, 3, 4 \rangle$: with τ_1 eliminated, the SMC now suggests that ERROR may be reached in *two* iterations of the loop, rather than one. REFINE eliminates τ_2 by returning the predicates $i + 1 = M$ and $i + 1 < 1024$. CEGAR continues eliminating paths containing increasing numbers of iterations through the loop, finally proving ERROR unreachable after 1024 iterations. This phenomenon, termed *loop unrolling*, results in an analysis which depends on the size of the array a loop traverses. For large arrays, this leads to state explosion, making the analysis infeasible.

2.2 CEGAR with Proof Templates

Iteratively removing paths of increasing length from loops like the one in Figure 2(a) is an inefficient and unnatural way to prove safety. The example can be proved more naturally by showing inductively that $i \leq M$ always holds at line 2:

1. Initially, $i \leq M$ holds trivially at line 2.
2. If $i \leq M$ at line 2 and the loop is entered, then $i \leq M \wedge i \neq M$, so $i < M$ at line 3, and $i \leq M$ after the `i++` on line 5.
3. Thus, $i \leq M$ at line 2 and $i < M$ at line 3.
4. Combining the fact that $i < M$ at line 3 with the fact that $M \leq 1024$, we have that ERROR is unreachable.

CEGAR+PT (Algorithm 2) attempts to guide an SMC towards proofs like the one above by introducing *proof templates*, which provide outlines of correctness proofs. The call to BUILDDB on line 2 builds a database which maps ⟨*loop, variable*⟩ pairs in the program to proof templates, by examining the structure of the loop. We describe BUILDDB and our proof templates in Section 3, concentrating for now on how the templates are used. When a loop is being unrolled, USETEMPLATE (line 20) queries the database built by BUILDDB to see if a template may be useful. However, BUILDDB may suggest inappropriate templates. The calls to BACKTRACK on lines 17 and 27 ensure that proof templates which do not prove helpful are eventually abandoned.

We illustrate CEGAR+PT on the example program in Figure 2(a), but stress that CEGAR+PT also works on more complicated programs like the one in Figure 1. BUILDDB records that i appears to be bounded by M in the loop on lines 2–5 of the program in Figure 2(a), and guesses that this bound can be used to prove the safety of the assertion. CEGAR+PT then runs as CEGAR, until it realizes that the loop is being unrolled[2]. Then, it invokes USETEMPLATE, which identifies and applies a corresponding proof template. USETEMPLATE adds predicates to E based on the structure of the loop; however, this alone is insufficient, as some of these predicates must be true before the loop is entered. Often proving that these predicates are in fact true on loop entry requires the discovery of additional "support" predicates. In our experiments with an earlier system, in which USETEMPLATE only added predicates to E, we found that the SMC would often focus its refinement efforts on the loop body, rather than discovering the support predicates, rendering our proof templates ineffective.

The proof templates must ensure that these support predicates are discovered. Thus, USETEMPLATE also adds explicit assumptions to P, in order to communicate the structure of the proof to the SMC, making

[2]Detecting loop unrolling can be done algorithmically, e.g., [23]; our current implementation uses a heuristic.

Algorithm 2 CEGAR+PT — CEGAR with proof templates.

```
1:  procedure CEGAR+PT(P, ψ)                                   ▷ Program, Property
2:      DB ← BUILDDB (P)                                       ▷ Template occurances
3:      E ← ∅                                                  ▷ Predicates
4:      ψ_0 ← ψ                                                ▷ Initial property
5:      S ← empty stack                                        ▷ Backtracking stack
6:      loop
7:          M ← ABSTRACT (P, ψ, E)                             ▷ M = model
8:          τ ← MODELCHECK (M, ψ)
9:          if τ = ϵ then                                      ▷ No path to ERROR
10:             if NOASSUMPTIONSLEFT(S) then
11:                 return SAFE
12:             else                                           ▷ Discharge assumptions
13:                 (P, E, ψ) ← DISCHARGENEXT(S)
14:         else
15:             if SPURIOUS(τ, P) then                         ▷ Refine abstraction
16:                 if TIMEOUT (S) then                        ▷ Template not helping
17:                     (P, E, ψ) ←BACKTRACK (S)
18:                 else
19:                     if ∃ℓ · UNROLLING(ℓ) ∧ HAVETEMPLATE(ℓ, DB, ψ, S) then
20:                         (P, E, S) ←USETEMPLATE(L, DB, ψ, S)
21:                     else
22:                         E ← E ∪ REFINE (τ)
23:             else
24:                 if ψ = ψ_0 then
25:                     return UNSAFE
26:                 else                                       ▷ An assumption did not hold
27:                     (P, E, ψ) ←BACKTRACK (S)
```

the proof template much more effective. The assumptions also modularize the analysis, which can make it much more efficient. An **assume(**p**)** statement at a line l of P tells the SMC to assume that p holds at l.

Since BUILDDB may suggest proof templates whose assumptions do not hold, CEGAR+PT must discharge all assumptions used (line 13), and backtrack if any assumption does not hold (line 27). This is facilitated by a backtracking stack S. Whenever USETEMPLATE supplies a template on line 20, it adds a stack frame to S. The stack frame contains (1) the current iteration of the loop beginning on line 6 of CEGAR+PT (to enable the TIMEOUT check on line 16), (2) the current values of P, E, and ψ (so that the calls to BACKTRACK on lines 17 and 27 can restore the state before the template was applied), and (3) the assumptions associated with the template (so that the SMC can discharge them, as indicated on line 13 of the algorithm). S also keeps track of the number of times a template has been applied to $\langle \ell, i \rangle$, to make sure that each candidate template can be applied in turn. We implement TIMEOUT by keeping a count of the current iteration of the loop beginning on line 6 of CEGAR+PT, and backtracking if the difference between the current count and the saved count on the template's stack frame exceeds a preset threshold, currently set at 20 iterations.

In the example program in Figure 2(a), USETEMPLATE adds to E the predicates $i \leq M$, $M \leq i$, and $M \leq 1024$ and the assumptions shown in Figure 2(d). These predicates and assumptions are part of the definition of the template, and are instantiated using the parameters i, M, and 1024, which come from the program. With these, the SMC can prove ERROR unreachable using the inductive argument described at the beginning of this section. In particular, $i < M$ can be represented as $\neg(M \leq i)$, so the SMC can precisely model the effect of incrementing i within the loop, and can use the assumption that $i \leq M$ initially to infer that i is bounded on each iteration of the loop. CEGAR+PT can then prove the assertion safe by using the assumption that $M \leq 1024$, and then discharging all assumptions made.

Line 19 of the algorithm checks whether a template can be used for the pair $\langle \ell, i \rangle$. In our implementation, templates can be used iff (1) $\psi = \psi_0$, (2) there is a proof template for $\langle \ell, i \rangle$, and (3) no template for $\langle \ell, i \rangle$ is already in use. The first condition is needed since, in this paper, we focus only on using templates for

Original Problem	Solve Under Assumptions	Discharge Assumptions

P: {...}
 	
while (i <= M) {
  Q: {...}
  assert (i <= N);
  R: {...}
}
	P: {...}
	assume (M+c <= N);
	while (i <= M) {
	  Q: {...}
	  assert ((M+c <= N) && (i <= M+c));
	  R: {...}
	}
		P: {...}
		assert (M+c <= N);
		while (i <= M) {
		  Q: {...}
		
		  R: {...}
		}

Predicates: $i \leq M, M \leq i, i \leq M + c, M + c \leq N$.

Figure 3: Structure of single-variable explicit template.

a single loop, leaving the extension to multiple dependent loops as future work. The results reported in Section 4 lead us to believe that proof templates are a promising foundation for enhancing SMCs.

3 Proof Templates for Array Traversals

In this section, we describe four proof templates for array traversals. Each template decomposes a proof of safety into two phases: proving the original property under additional assumptions, and discharging these assumptions. We also describe how we guide an SMC towards these proofs.

Each template is parameterized by a set of expressions, which come from the text of the program. For simplicity, we present each template using loops with a single *loop condition* at the head of the loop, in which obtaining most template parameters is trivial. We describe how our implementation handles more general loops, and obtains template parameters for them, in Section 3.5. Furthermore, while we describe our templates using explicit array indexing rather than pointers, our templates can be extended to handle pointers represented as $\langle$*base,offset*$\rangle$ pairs.

The heuristics for choosing among our four templates are based on combinations of two conditions: whether the iterator (defined below) in the loop condition is the same as the iterator in the assertion being checked, and whether the loop condition is an arithmetic comparison on an iterator or a test on an array cell. We have found that this information is often sufficient to choose the correct template.

3.1 Preliminaries

We define our templates using the notion of a *loop iterator*. Informally, an iterator is a variable whose value in one loop iteration influences its value in the next. We make this definition precise using data dependence and dominators [27].

A statement s is a *definition* of a variable v (or s *defines* v) if s contains an assignment to v. If s reads the value of v, we say that s *uses* v. For any loop ℓ, constants and variables which are used but not defined in ℓ are called *loop constants*.

Let s_1 and s_2 be statements. We say that s_1 is *dependent* on s_2, written $s_1 \ \delta \ s_2$, if there exists a variable v such that s_1 uses v, s_2 defines it, and the definition reaches s_1. We write $s_1 \ \delta^\star \ s_2$ if there exists a set of statements $s'_1, \ldots s'_n$, such that $s_1 \ \delta \ s'_1 \ \delta \ \cdots \ \delta s'_n \ \delta \ s_2$.

Given a loop ℓ, a variable i is an *iterator of* ℓ iff there exists a statement s such that

- s is a definition of i within ℓ;
- $s \ \delta^\star \ s$; and
- s is not dominated by any other definition s' in ℓ such that s' assigns a loop constant to i.

While iterators are similar to loop counters or induction variables [27], they are more general, since they need not change by a fixed amount on every loop iteration.

Finally, we describe our templates using Hoare triples [19]. In this notation, $\{p\}\ S\ \{q\}$ means that p and q are logical expressions, S is a program fragment, and that if p holds initially, then q holds after S executes.

3.2 Single-Variable Explicit Template

We use the single-variable explicit template when a loop iterator i appears in a bounds check within a loop, and the loop condition is a comparison between i and some loop constant M, e.g., as in the program in Figure 2(a). We call the template "explicit" because an iterator is being explicitly compared to a bound in the loop conditional (as opposed to a check involving an array cell).

Figure 3 shows how the template is communicated to an SMC. We describe the template assuming that the loop condition is $i \leq M$, but the details are similar if the loop condition is $i < M$ or $i \neq M$. The symbols P, Q, and R in Figure 3 denote regions of the program. The template's parameters are i, M, N, and c; roughly, c represents (a guess at) the maximum amount by which i can be increased in Q. The loop condition $i \leq M$ is true at the start of each iteration, and our goal is to prove the safety of the **assert**$(i \leq N)$ within the loop. The template breaks the proof of safety down as follows:

1. $\{true\}\ P\ \{M + c \leq N\}$,
2. $\{M + c \leq N\}\ Q; R\ \{M + c \leq N\}$, and
3. $\{i \leq M \wedge M + c \leq N\}\ Q\ \{i \leq M + c \wedge M + c \leq N\}$.

The first two points ensure that $M + c \leq N$ is true at the beginning of each iteration of the loop. The third point ensures that if $M + c \leq N$ and the loop condition hold at the beginning of the loop, then the assertion is safe. Note the template's modularity: the first point requires no analysis of the loop, while the latter two points focus on the analysis of the loop's body.

The second column of Figure 3 shows the effect of the changes that USETEMPLATE makes to the program's internal representation in order to guide the SMC towards this proof. It adds an **assume**$(M + c \leq N)$ before the loop, and supplies the SMC with the predicates $i \leq M$, $M + c \leq N$, and $i \leq M + c$, since these predicates appear in the proof template. It also supplies the SMC with the predicate $M \leq i$, which we have often found to be useful: together with $i \leq M$, it allows us to describe any comparison $(<, \neq, \ldots)$ between i and M. If the SMC is able to prove the safety of the original assertion, lines 13–14 of CEGAR+PT tell the SMC to discharge the assumption $\{true\}\ P\ \{M + c \leq N\}$, effectively by changing the **assume**$(M + c \leq N)$ statement into an assertion, as shown in the last column of Figure 3. Finally, in cases where the loop condition is $i \neq M$, such as in our example in Figure 2(a), USETEMPLATE also inserts an **assume**$(i \leq M)$ statement before the loop, and the SMC must additionally prove that $\{i < M\}\ Q; R\ \{i \leq M\}$ holds, so that i is bounded on each iteration.

3.3 Two-Variable Explicit Template

Loops over arrays often involve two iterators — for example, iterators i and j may index into two different buffers, as in the example in Figure 1; this is especially common when copying data using pointers. We use the two-variable explicit template when the loop condition is a test on one iterator (the *leader*), but the bounds check is on another. The main idea is to try to prove that the change in the second iterator on any iteration of the loop is bounded by the change in the leader.

Figure 4 shows the structure of the template. The parameters are i, j, M, N, and c. Assume that the loop condition is $i \leq M$, and that the bounds check is **assert**$(j \leq N)$; the details are similar if the loop condition is $i \neq M$ or $i < M$. We introduce the variables i_s and j_s to denote the values of i and j before the loop is entered; these variables allow us to represent the notion of change in i and j. Let P' be the composition of statements $(P; i_s = i; j_s = j)$, and let $\Phi = (M + c - i_s \leq N - j_s)$ and $\Psi = (j - j_s \leq i - i_s)$. Then, the template breaks the proof of safety down as follows:

1. $\{\Psi\}\ Q; R\ \{\Psi\}$,
2. $\{true\}\ P'\ \{\Phi\}$,
3. $\{\Phi\}\ Q; R\ \{\Phi\}$, and
4. $\{i \leq M \wedge \Phi \wedge \Psi\}\ Q\ \{i \leq M + c \wedge \Phi \wedge \Psi\}$.

The first step guarantees that $j - j_s \leq i - i_s$ on every loop iteration; this holds trivially on the first iteration. The next two steps do the same for $M + c - i_s \leq N - j_s$. The intuitive meaning of this expression is that the maximum amount which may be copied into the buffer accessed by j cannot exceed the room remaining in

Original Problem	Solve Under Assumptions	Discharge Assumptions

P: {...}
 	
while (i <= M) {
  Q: {...}
  assert (j <= N);
 	
  R: {...}
}
	P: {...}
	i_s = i;
	j_s = j;
	assume (M + c − i_s + j_s <= N);
	while (i <= M) {
	  Q: {...}
	  assert ((i <= M + c) &&
	  (j − j_s <= i − i_s) &&
	  (M + c − i_s + j_s <= N));
	  R: {...}
	}
		P: {...}
		i_s = i;
		j_s = j;
		assert (M + c − i_s + j_s <= N);
		while (i <= M) {
		  Q: {...}
		
		  R: {...}
		}

Predicates: $i \leq M, M \leq i, i \leq M + c, j - j_s \leq i - i_s, M + c - i_s + j_s \leq N.$

Figure 4: Structure of two-variable explicit template.

Original Problem	Solve Under Assumptions	Discharge Assumptions

P: {...}
 	
while (A[i] != '\0') {
  Q: {...}
  assert (i <= N);
  R: {...}
}
	P: {...}
	assume (strlen(A) <= N);
	assume (i <= strlen(A));
	while (A[i] != '\0') {
	  Q: {...}
	  assert ((strlen(A) <= N) && (i <= strlen(A));
	  R: {...}
	}
		P: {...}
		assert (strlen(A) <= N);
		assert (i <= strlen(A));
		while (A[i] != '\0') {
		  Q: {...}
		
		  R: {...}
		}

Predicates: $i \leq \mathit{strlen}(A), \mathit{strlen}(A) \leq i, \mathit{strlen}(A) \leq N, A[i] = \text{`\textbackslash 0'}.$

Figure 5: Structure of single-variable string template.

it at loop entry. The last step says that if, at the beginning of every iteration, $(i \leq M) \wedge (M + c + j_s - i_s \leq N) \wedge (j - j_s \leq i - i_s)$, then the assertion is safe, by the following argument:

$$\begin{array}{lll} j & \leq i + j_s - i_s & \text{rewrite } j - j_s \leq i - i_s \\ & \leq i + (N - (M + c)) & \text{since } M + c - i_s \leq N - j_s \\ & = N + (i - (M + c)) & \\ & \leq N & \text{since } i \leq M + c \end{array}$$

As with the single-variable case, this template is modular.

We have presented the template assuming that the leader increases on each loop iteration. The template is similar if the leader decreases instead, e.g., if the leader is a count of the amount of space left in an array. In this case, the expressions $j - j_s \leq i - i_s$ and $M + c - i_s \leq N - j_s$ are replaced by $j - j_s \leq i_s - i$ and $i_s - M - c \leq N - j_s$, respectively.

3.4 Handling Strings

We define strings to be null-terminated character arrays. Thus, when a loop traverses a string A using an iterator i, it will likely have a loop condition $A[i] \neq$ '\0' (or one that implies it, such as $A[i] =$ ' '), rather than an explicit arithmetic comparison on i. For such loops, we use the proof template shown in Figure 5. For conciseness, let $sl = \mathit{strlen}(A)$, where

$$\mathit{strlen}(A) \stackrel{\text{def}}{=} \min\{x \geq 0 \mid A[x] = \text{`\textbackslash 0'}\}$$

The template's parameters are i, A, and N. The template breaks the proof of safety down into five steps:

1. $\{true\}\ P\ \{sl \leq N\}$,
2. $\{sl \leq N\}\ Q; R\ \{sl \leq N\}$,
3. $\{true\}\ P\ \{i \leq sl\}$,
4. $\{i < sl\}\ Q; R\ \{i \leq sl\}$, and
5. $\{i < sl \wedge sl \leq N\}\ Q\ \{i \leq sl \wedge sl \leq N\}$.

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);`
	`}`

Figure 6: Program instrumentation for string reads and writes; NONDET means non–deterministic assignment.

Original Problem	Solve Under Assumptions	Discharge Assumptions

`P: {...}`
 	
`while (A[i] != '\0') {`
`  Q: {...}`
`  assert (j <= N);`
 	
`  R: {...}`
`}`
	`P: {...}`
	`i_s = i;`
	`j_s = j;`
	`assume (strlen(A) - i_s + j_s <= N);`
	`while (A[i] != '\0') {`
	`  Q: {...}`
	`  assert ((i <= strlen(A)) &&`
	`  (j - j_s <= i - i_s) &&`
	`  (strlen(A) - i_s + j_s <= N));`
	`  R: {...}`
	`}`
		`P: {...}`
		`i_s = i;`
		`j_s = j;`
		`assert (strlen(A) - i_s + j_s <= N);`
		`while (A[i] != '\0') {`
		`  Q: {...}`
		
		`  R: {...}`
		`}`

Predicates: $i \leq strlen(A)$, $strlen(A) \leq i$, $j - j_s \leq i - i_s$, $strlen(A) - i_s + j_s \leq N$, $A[i] =$ '\0'.

Figure 7: Structure of two-variable string template.

The first two ensure that $sl \leq N$ at the start of every loop iteration. The second two ensure that $i < sl$ at the start of every loop iteration; they take advantage of the fact that if $i \leq sl$ holds before the loop is entered, then $i < sl$ on loop entry, since $A[i] \neq$ '\0' $\wedge$ $A[sl] =$ '\0' implies that $i \neq sl$. The last step ensures that if $i < sl$ and $sl \leq N$ at the beginning of each loop iteration, the assertion is safe.

Our treatment of strings extends naturally to two-variable traversals; see Figure 7 for details.

In order to represent predicates containing the *strlen* function as atomic predicates, we conservatively approximate *strlen* using program instrumentation. Specifically, we associate with each array A a variable $A\_nullpos$, and add the instrumentation shown in Figure 6 to make $A\_nullpos$ a safe approximation of *strlen*(A). NONDET stands for non-deterministic assignment. From the definition of *strlen*, $A\_nullpos \geq 0$ and $A[A\_nullpos] =$ '\0', so SMCs can assume them as invariants throughout the program.

3.5 Applying Templates in Practice

Earlier in this section, we described a heuristic for choosing among our proof templates for structured loops: if the loop condition is an arithmetic test on an iterator, we choose an explicit template; otherwise, if the loop condition is a test on an array cell, we choose a string template. If the iterator in the loop condition is the same as the iterator in the assertion being checked, we choose a single-variable template; otherwise, we choose a two-variable template. We now generalize this test to more general loops, and describe how we obtain template parameters in this context. The generalized algorithm comprises BUILDDB, which is implemented as an extension to CIL [28].

Exit branches. More general loops may have *exit branches* at their heads *or* within their bodies. An exit branch is a branch statement upon which a loop exit is control-dependent [15], e.g., the branches on line 10 of Figure 1, and lines 5, 8, and 9 of the program in Figure 8. Our implementation chooses templates by examining exit branches. We have found that our proof templates work equally well for common less structured loops such as those in Figures 1 and 8. Since a loop may have multiple exit branches (e.g., `strncpy`), our implementation can suggest multiple proof templates for the same $\langle$*loop,iterator*$\rangle$ pair.

Deriving parameters. So far, we have assumed that iterators appear directly in exit branch conditions and assertions (making it trivial to obtain template parameters), and that the assertions appear within the loop's body. We relax both conditions by searching for expressions *derived from* iterators, where an expres-

```
1   void example () {
2     char buf[BUF_SZ], c; int len=BUF_SZ-1, i=0, tmp;
3     while (1) {
4       c = NONDET;
5       if (i == len) return;
6       if (c == '\\') {
7         i++;
8         if (i == len) return; }
9       else if (c == '.') break;
10      i++; }
11    tmp = i+1;
12    assert (tmp < BUF_SZ); }
```

Figure 8: A loop with multiple exit branches, based on code from Apache.

sion is derived from an iterator if it is semantically equivalent to a linear expression over the iterator and loop constants. These are found using *reaching definitions* [27]. Since we restrict derived expressions to be linear, we can invert them in order to obtain template parameters from exit branches and assertions.

We can use proof templates to check an assertion outside of a loop, provided that the loop dominates the assertion. In this case, we restrict the loop constants in derived expressions to those which are not defined on any path from the loop to the dominated assertion. In Figure 8, the assertion on line 12 is dominated by the loop on lines 3–10, and *tmp* is derived from i.

For strings, we search for expressions whose values are derived from array cells whose indices are, in turn, derived from iterators; for example, in the program in Figure 1, *ch* is derived from $src[i]$.

Derived expressions allow us to obtain the template parameters i, j, M, N, and A. We compute a candidate value for the remaining parameter, c, through a simple symbolic execution [21] which follows CFG predecessors from an assertion to the nearest exit branch used in the template, computing the sum of all updates to an iterator.

4 Evaluation

In this section, we compare the performance of an SMC augmented with proof templates (PTYASM) against three other SMCs: YASM [16] (without proof templates), BLAST [17, 18], and SATABS [10]. We ran the tools on testcases derived from the Verisec suite [24], a benchmark designed to compare verification techniques. The suite consists of 149 small C programs (containing at most 538 LOC) derived from 22 buffer overflow vulnerabilities reported in CVE [12], a public database of security vulnerabilities. Each testcase in the suite includes not only a vulnerable version but also its safe counterpart in which the buffer overflows have been fixed by applying the official patch. Buffer size declarations are parameterized by a macro which can be set by the experimenter.

We selected 26 patched testcases from 18 vulnerabilities by excluding those cases which either did not contain buffer-dependent loops or which contained loop structures already represented in the set. Since our current implementation of proof templates only supports programs with a single loop and a single assertion, we constructed, by hand, single-loop testcases isolating each array bounds assertion in each of the selected programs. Overall, 59 testcases were constructed for the evaluation.

The SMCs used in the experiments are briefly described below. YASM [16] generates multi-valued abstractions and uses weakest preconditions for refinement. BLAST [17, 18] uses a "lazy" abstraction strategy and Craig interpolation for refinement. SatAbs [10] uses a SAT-solver for predicate abstraction and refinement. We ran the tools with the recommended options: YASM with `--refiner cbj-i-s`; BLAST, release 2.4, with `-craig -predH 7`; and SATABS, release 1.8, with `--iterations 0`, which disables the CEGAR iteration limit, and `--no-bounds-check`, which disables the built-in bounds-checking, as it can interfere with our manually-inserted bounds check assertions.

The test platform was a dual quad-core Intel E5355 2.66GHz system with 16GB RAM running Ubuntu 6.06.2. To pass each testcase, a tool had to prove the safety of the array bounds assertion within the timeout period, set at 600s; the timeout is necessary because SMCs may run for an unbounded amount of time rather than returning incorrect results. The buffer size was chosen to be 1024, in order to be realistically large and to prevent the tools from successfully passing a testcase by loop unrolling. If a tool timed out, crashed, or incorrectly reported that an assertion was violated, we counted this as a failure.

Class	#Cases	PTYASM	YASM	BLAST	SATABS
None	13	9	9	9	9
1ex	25	25	8	10	13
1str	11	8	0	0	0
2ex	7	4	0	0	0
2str	3	3	0	0	0
Total	59	49	17	19	22

Table 1: Evaluation results: number of testcases solved at buffer size 1024 with a timeout of 600s.

```
1  while (A[i] != 0) {
2    skipping = (j >= M);
3    if (!skipping) {
4      assert (j < N);
5      j++; }
6    i++; }
```

(a)

```
1  while (A[i] != 0) {
2
3    if (A[i] == '?') {
4      assert (i < N);
5      A[i] = 0; }
6    i++; }
```

(b)

Figure 9: Programs where (a) exit branches do not yield the correct template and (b) the current template is insufficiently general.

The results of the experiments are summarized in Table 1. Initially, we manually classified the testcases into classes associated with templates, where a program belongs to a template's class if (1) the template, or a minor generalization of it, can be used to prove safety, and (2) the template can be inferred by examining exit branches. Classes are indicated in the first column of the table: "None" denotes no template, and "1ex", "1str", "2ex", and "2str" denote the single-variable explicit, single-variable string, two-variable explicit, and two-variable string templates, respectively. Each row of the table shows the results for one class of testcases. For example, the second row shows that there were 25 "1ex" testcases in total, that PTYASM passed 25 of these, and that YASM, BLAST, and SATABS passed 8, 10, and 13 of the "1ex" cases, respectively. Overall, PTYASM achieved a success rate of 83%, a significant improvement over the previous best of 37% (SATABS). In the 46 cases outside of the "None" class, PTYASM achieved a success rate of 87% whereas the previous best was 28% (SATABS). The complete experimental data and test materials are available online at `http://www.cs.toronto.edu/~kelvin/ase08`.

Overall, PTYASM failed to check ten testcases within the timeout period. In one of these testcases, the correct template was supplied to PTYASM, but it was unable to converge within the timeout period; this is one of the failing "1str" testcases. The remaining nine failures fall into two groups. In both groups, PTYASM backtracked to the standard refinement strategy (loop unrolling) and eventually timed out. The first group consists of the four failing testcases in the "None" class, for which BUILDDB did not suggest a correct template, either because no appropriate template exists, or because a correct template exists, but cannot be inferred from exit branches. Figure 9(a) illustrates a loop structure which occurs in a testcase derived from Sendmail CVE-2002-1337 for which BUILDDB suggests the two-variable string template, but the exit branch condition $A[i] \neq 0$ does not guarantee safety if the precondition $strlen(A) - i_s \leq N - j_s$ fails to hold. Instead, the use of the flag variable *skipping* prevents the iterator j from increasing past M. The correctness argument actually depends on the precondition $M \leq N$, and the single-variable explicit proof template would be sufficient to allow the model-checker to solve this testcase without unrolling. It is straightforward to extend BUILDDB to recognize this loop structure: (1) apply *expression propagation*, replacing the use of *skipping* on line 3 with its unique reaching definition from line 2, namely the expression $j \geq M$; and (2) in addition to examining exit branches, suggest proof templates based on bounds checks on iterators, such as the check that $\neg(j >= M)$ on line 3 of Figure 9(a), when these checks dominate assertions.

The second group consists of five testcases in which the exit branches indicate the proper correctness argument, but our current template is insufficiently general. These account for two failing "1str" testcases, which come from a loop in Apache (from CVE-2006-3747), and three failing "2ex" testcases, which come from a loop in Sendmail (from CVE-1999-0206). Figure 9(b) shows a simplified version of the loop from Apache. Here, BUILDDB suggests the single-variable string template, and the exit branch condition $A[i] \neq 0$ guarantees safety since $i < strlen_s(A) \leq N$ is a loop invariant, where $strlen_s(A)$ is the initial string length of A. However, the loop body may write a null character into A, thereby decreasing $strlen(A)$ and

violating the proof step $\{i < strlen(A)\}\ Q; R\ \{i \leq strlen(A)\}$. To handle such loops, the string templates can be extended with instrumentation and predicates to track $strlen_s(A)$.

Overall, the results show that proof templates are an effective strategy for verifying bounds checks on large buffers. In many cases where a template applied, using proof templates enabled YASM+PT to converge whereas the existing strategies failed.

5 Related Work

To our knowledge, we are the first to propose augmenting SMCs with proof templates designed to capture common programming patterns, and to explicitly use assumptions to enable the analysis of loops for which predicates must be true on entry. However, others have proposed extensions to CEGAR to help it analyze programs with loops.

Beyer et al. [6] integrate invariant synthesis [5] with CEGAR for analyzing loops, but do not address strings or provide a mechanism to ensure that needed predicates are true on loop entry. The user must specify an *invariant template*, which requires intuition about the structure of the loop and the property being checked. It may be possible to combine our approach with theirs, by using our algorithm to suggest proof templates, casting them as invariant templates, and using their invariant synthesis to customize them.

ACSAR [30] uses *transfinite refinement* to better abstract loops, and has been applied to the verification of small loops. ACSAR is able to determine that updates of two variables are paired, as in our two-variable cases, but does not analyze strings or address the problem of predicates which must be true on loop entry.

Jhala and McMillan's split prover [20] constrains the predicates which REFINE may return in order to ensure that CEGAR eventually converges if a proof of safety exists within difference logic (i.e., predicates of the form $x - y \leq c$, where c is a constant). The split prover can prevent loop unrolling in some circumstances, but since it has no knowledge of programming patterns, it does not attempt to guide an SMC towards an efficient proof. Furthermore, this approach may drastically increase the overall number of predicates if the proof of safety requires a predicate with a large numeric constant as a difference bound.

Chaki and Hissam [8] use an SMC to verify the absence of buffer overflows in functions from the C standard library (i.e., `strcpy`, etc.) by introducing stubs for these functions, but make no attempt to analyze custom loops. Other projects have augmented SMCs to better prove the *presence* of buffer overflows, either using looping counterexamples [23, 35], or concrete execution [22]; these efforts are orthogonal to ours, since they seek to prove the presence of errors, rather than their absence.

The EUREKA SMC changes the CEGAR paradigm by using linear programs in the model checking phase instead of boolean programs. This enables some problems involving array traversal to be solved efficiently, but makes the SMC's model checking phase incomplete [1].

Several tools [14,33,34] use abstract interpretation [11] to verify the safety of array bounds checks. The key difference between these tools and SMCs is that they rely on fixed abstractions which are powerful, but cannot be refined at analysis time and therefore require careful construction and customization to the software being checked. Since the abstract domains are fixed, these tools can either keep track of too much information, leading to long analysis times [36], or of too little information, leading to false alarms. We note that many tools based on abstract interpretation explicitly keep track of string length [14]; this approach inspired our string instrumentation.

Other tools use Hoare-style deductive verification to verify the absence of buffer overflows. These tools typically require an inference procedure to provide loop invariants [26]. Denney and Fischer [13] have applied a pattern-based approach which is similar in spirit to ours to deductive verification, but not for the problem of array bounds checking, and their scope is restricted to automatically-generated programs.

6 Conclusions and Future Work

In this paper, we have described an architecture for augmenting SMCs with proof templates, a set of proof templates designed to prove the safety of common array traversals, and an algorithm to heuristically choose among these templates. Our experiments show that proof templates enable SMCs to prove the safety of array bounds checks much more effectively. We note two areas for future work.

Extending to multiple loops. We have addressed the abstraction of a single loop using proof templates, but plan to generalize this framework to analyze inter-dependent loops; in particular, we would like to use

proof templates while discharging assumptions. The main difficulty of this problem is that the assumptions do not have the same form as the original assertions, and may require bounding multiple variables.

Extending to other problem domains. We are interested in generalizing our methodology to problem domains other than array bounds verification by taking advantage of other analyses of program structure with which we can associate templates. Existing techniques for automatically finding instances of design patterns [32] lead us to believe that such generalizations hold promise.

References

[1] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction Refinement of Linear Programs with Arrays. In *Proc. TACAS'07*.

[2] 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*, pages 73–85, 2006.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. "Automatic Predicate Abstraction of C Programs". In *Proc. PLDI'01*, pages 203–213, New York, NY, USA, 2001.

[4] T. Ball and S. Rajamani. "Generating Abstract Explanations of Spurious Counterexamples in C Programs". Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. "Invariant Synthesis for Combined Theories". In *Proc. VMCAI'07*, LNCS 4349, pages 378–394, 2007.

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

[7] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. "Predicate Abstraction with Minimum Predicates". In *Proc. CHARME'03*, pages 19–34, 2003.

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

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

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

[11] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In *Proc. POPL'78*, pages 84–96, 1978.

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

[13] E. Denney and B. Fischer. Annotation Inference for Safety Certification of Automatically Generated Code (Extended Abstract). In *Proc.ASE'06*.

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

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization". *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

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

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

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. "Abstractions from Proofs". In *Proc. POPL'04*, pages 232–244, 2004.

[19] C. Hoare. "An Axiomatic Basis for Computer Programming". *Comm. ACM*, 12(10):576–580, 1969.

[20] R. Jhala and K. L. McMillan. "A Practical and Complete Approach to Predicate Refinement". In *Proc. TACAS'06*, pages 459–473, 2006.

[21] J. C. King. "Symbolic Execution and Program Testing". *Comm. ACM*, 19(7):385–394, 1976.

[22] D. Kroening, A. Groce, and E. Clarke. "Counterexample Guided Abstraction Refinement via Program Execution". In *Proc. ICFEM'04*, pages 224–238, 2004.

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

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

[25] Microsoft. Static Driver Verifier, 2004. `http://www.microsoft.com/whdc/devtools/tools/SDV.mspx`.

[26] Y. Moy. "Sufficient Preconditions for Modular Assertion Checking". In *Proc. VMCAI'08*, LNCS, pages 188–202, 2008.

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

[28] 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, pages 213–228, 2002.

[29] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. "CCured: Type-Safe Retrofitting of Legacy Software". *ACM TOPLAS*, 27(3):477–526, 2005.

[30] M. N. Seghir and A. Podelski. "ACSAR: Software Model Checking with Transfinite Refinement". In *Proc. SPIN'07*, pages 274–278, 2007.

[31] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. "On the Effectiveness of Address-space Randomization". In *Proc. CCS'04*, pages 298–307, 2004.

[32] N. Shi and R. A. Olsson. "Reverse Engineering of Design Patterns from Java Source Code". In *Proc. ASE'06*, pages 123–134, 2006.

[33] The MathWorks. PolySpace Products for Embedded Software Verification. `http://www.mathworks.com/products/polyspace`.

[34] A. Venet and G. Brat. "Precise and Efficient Static Array Bound Checking for Large Embedded C Programs". In *Proc. PLDI'04*, pages 231–242, 2004.

[35] C. Wang, A. Gupta, and F. Ivancic. "Induction in CEGAR for Detecting Counterexamples". *Proc. FMCAD'07*, pages 77–84, 2007.

[36] M. Zitser, R. Lippmann, and T. Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". *SIGSOFT SEN*, 29(6):97–106, 2004.