Optimal Guard Synthesis for Memory Safety

Thomas Dillig[1], Isil Dillig[1], and Swarat Chaudhuri[2]

[1] UT Austin
[2] Rice University

Abstract. This paper presents a new synthesis-based approach for writing low-level memory-safe code. Given a partial program with missing guards, our algorithm synthesizes concrete predicates to plug in for the missing guards such that all buffer accesses in the program are memory safe. Furthermore, guards synthesized by our technique are the simplest and weakest among guards that guarantee memory safety, relative to the inferred loop invariants. Our approach is fully automatic and does not require any hints from the user. We have implemented our algorithm in a prototype synthesis tool for C programs, and we show that the proposed approach is able to successfully synthesize guards that closely match hand-written programmer code in a set of real-world C programs.

1 Introduction

Memory safety errors are a perennial source of crashes and vulnerabilities in programs written in unsafe languages, and even expert programmers often write erroneous code that accesses out-of-bounds buffers or invalid memory. Over the past few decades, there has been much research on helping programmers write memory safe code. Broadly speaking, existing approaches fall into two categories:

Dynamic instrumentation. Many approaches, such as those employed in memory managed languages like Java and C#, add run-time checks to guarantee the safety of each memory access. While such approaches prevent memory corruption and associated security vulnerabilities, they do not prevent run-time failures and often add significant performance overhead.

Static verification. Much recent research has focused on statically guaranteeing memory safety of programs written in unsafe languages [1–5]. While these techniques can uncover all potential memory safety errors, the errors identified by the verifier may be hard to understand, debug, and fix.

In this paper, we propose a new approach based on *program synthesis* to the design of memory-safe low-level code. Concretely, suppose that a programmer wishes to write a region of code `R` implementing a given functionality, but `R` can access out-of-bounds memory under certain assumptions about program inputs or previously taken branches. In our approach, the programmer embeds `R` within the scope of an unknown *guard* predicate whose sole purpose is to ensure the memory safety of `R`. This is done using a syntax of the form:

```
if(??) {R}  else { /* handle error */ }
```

where the unknown guard is indicated by `??`. Our approach uses a new *guard synthesis* algorithm to compute a predicate `P` over program variables such that, when `??` is replaced by `P`, all memory accesses within `R` are provably memory-safe.

Unlike dynamic approaches, our method does not require run-time instrumentation to track allocation sizes or pointer offsets, thereby avoiding the associated performance overhead. Instead, we statically infer a single guard that guarantees the safety of *all* memory accesses within a code block. Furthermore, our approach goes beyond static verification: It not only guarantees memory safety, but also helps the programmer write safe-by-construction code. The programmer is only asked to tell us *which* code snippets must be protected by a guard, rather than the tedious, low-level details of *how* to protect them.

Our synthesis algorithm is based on the principle of *logical abduction*. Abduction is the problem of finding missing hypotheses in a logical inference task. In more detail, suppose we have a premise P and a desired conclusion C for an inference (P and C will be typically generated as constraints from a program) such that $P \not\models C$. Given P and C, abduction infers a simplest and most general explanation E such that $P \wedge E \models C$ and $P \wedge E \not\models \text{false}$.

Previous work has shown how to use abduction for program verification, by framing unknown invariants as missing hypotheses in a logical inference problem [6, 7, 5]. While adapting abduction to synthesis is a nontrivial technical challenge, the end result is an algorithm with several appealing properties:

Optimality of synthesis. Our algorithm gives a guarantee of *optimal synthesis* — i.e., the synthesized guards are optimal according to a quantitative criterion among all guards that guarantee memory safety. Optimality has been argued to be an important criterion in program synthesis. For instance, Alur et al. [8] argue that "Ideally, [in synthesis] we would like to associate a cost with each [synthesized] expression, and consider the problem of optimal synthesis which requires the synthesis tool to return the expression with the least cost among the correct ones. A natural cost metric is the size of the expression." However, few existing approaches to software synthesis take on such an optimality goal.

The notion of costs used in this paper is two-dimensional: one dimension quantifies expression complexity (we use the number of variables as a proxy for complexity), and the other quantifies generality (weaker guards have lower costs). The guards we synthesize are *Pareto-optimal* with respect to this notion of costs — i.e., there is no solution that is weaker as well as less complex.

Automation. Unlike most recent approaches to program synthesis [9–11], our algorithm can synthesize expressions without the aid of user-specified structural hints. In particular, the programmer does not need to provide expression templates with unknown coefficients to be inferred.

Practicality. Our algorithm incorporates precise reasoning about array bounds and low-level pointer arithmetic, which are necessary ingredients for synthesizing guards to guarantee memory safety. Furthermore, as shown in our experimental evaluation, the proposed synthesis algorithm can successfully synthesize guards required for memory safety in real C applications and produces guards that closely match hand-written code.

```
1.  int main(int argc, char** argv) {
2.    char *command = NULL;
3.    if (argc <= 1)  {
4.        error (0, 0, _("too few arguments"));
5.        usage (EXIT_FAIL);
6.    }
7.    argv++; argc--;
8.    while ((optc = getopt(argc, argv, ...)) != -1) {
9.      switch(optc) {
10.        case 'c':
11.          command =  optarg; break;
12.        ...
13.      }
14.    }
15.   if (??)  usage (EXIT_CANCELED);
16.    timeout = parse (argv[optind++]);
17.    files = argv + optind;
18.    if (!target_dir) {
19.      if (! (mkdir_and_install ? install_in_parents(files[0], files[1])
20                                 : install_in_file(files[0], files[1])))
21.        ...
22.    }
23.  }
```

Fig. 1: Motivating example

2 Motivating example and overview

We now present an overview of our approach using a motivating example. Consider the code snippet shown in Figure 1, which is based on the Unix coreutils. This program parses command line arguments with the help of a clib function called `getopt`. Specifically, lines 8-14 process the optional command line arguments while the code after line 16 performs the program's required functionality. Here, variable `optind` used at lines 16-17 is initialized by `getopt` to be the index of the next element to be processed in `argv`. Looking at lines 16-23, the programmer expects the user to pass some required arguments and accesses them at lines 16, 19, and 20. However, since the user may have forgotten to pass the required arguments, the programmer must explicitly check whether the memory accesses at lines 16,19,20 are safe in order to prevent potentially disastrous buffer overflow or underflow errors. If her assumptions are not met, the programmer wishes to terminate the program by calling the exit function called `usage`. However, coming up with the correct condition under which to terminate the program is tricky even on the small code snippet shown here: The programmer has performed pointer arithmetic on `argv` at line 7, and the variable `files` is an alias of `argv` at offset `optind` which has previously been modified at line 16.

Using our technique, the programmer can use the `??` predicate at line 15 to indicate the unknown check required for ensuring memory safety of the remainder of the program. Our technique then automatically synthesizes the guard `(argc - optind) > 2` as a sufficient condition for the safety of all buffer accesses in lines 16-23. Since the check inferred by our technique is correct-by-construction, the remainder of the program is guaranteed to be memory safe.

Algorithm overview. Our algorithm proceeds in two phases, consisting of constraint generation and solving. During constraint generation, we represent the unknown guards using placeholder formulas χ and then generate verification conditions over these unknown χ's. The constraint solving phase, which employs an iterative abduction-based algorithm, infers a concrete predicate for each χ that makes all generated VCs valid. In addition to guaranteeing Pareto-optimality, this approach does not require the user to specify templates describing the shape of the unknown guards. Furthermore, since the abduced solutions imply the validity of the VCs, we do not need to externally validate the correctness of the synthesized program using a separate verifier or model checker.

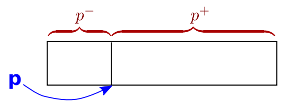

Fig. 2: Auxiliary variables

The constraint generation phase consists of two key ingredients: First, to reason about out-of-bounds memory accesses, we introduce ghost variables that track allocation sizes and pointer offsets. Specifically, for each pointer `p`, a variable p^- indicates the offset of `p` in the block of memory it points to, and p^+ tracks the size of `p` relative to p^-. This is shown in Figure 2. These ghost variables enable reasoning about pointer arithmetic in a precise way and allow us to generate symbolic verification conditions for memory safety.

The second key ingredient of constraint generation is a dual forwards and backwards static analysis that simultaneously computes strongest postconditions and weakest preconditions. For each unknown guard to be synthesized, the forwards analysis computes a formula ϕ representing facts that are known at this program point, while the backwards analysis provides a weakest precondition ψ for the safety of the code protected by this unknown guard. Now, given a statement `S` involving an unknown guard and the two formulas ϕ and ψ , our technique generates the VC $(\phi \wedge \chi(\boldsymbol{v})) \rightarrow \psi$ where χ is a predicate representing the unknown guard and $\boldsymbol{v}$ represents all program variables in scope at this program point. Here, formulas ϕ and ψ may also contain other unknowns.

In the constraint solving phase, we use an iterative, worklist-based algorithm that employs abduction to solve for the unknown χ predicates. Given a set of constraints $\mathcal{C}$ of the form $(F_1(\chi_1, \ldots \chi_{i-1}) \wedge \chi_i) \rightarrow F_2(\chi_{i+1}, \ldots \chi_n)$ where $F(\boldsymbol{\chi})$ denotes a formula over unknowns $\boldsymbol{\chi}$, we show how to infer a solution for each χ_i such that all constraints in $\mathcal{C}$ become valid. Our algorithm guarantees the Pareto-optimality of the solution relative to the inferred loop invariants. That is, assuming a fixed set of loop invariants, if we pick any unknown guard and try to improve it according to our cost metric, then the resulting set of guards is no longer a solution to our synthesis problem.

Example redux. We now go back to the code example from Figure 1 to illustrate our approach. Initially, we assume that `argv` points to the beginning of an allocated block of size `argc`; hence, our analysis starts with the fact:

$$\text{argv}^+ = \text{argc} \wedge \text{argv}^- = 0 \qquad (1)$$

Next, we perform forward reasoning to compute the strongest postcondition of (1) right before line 20. Here, the forward analysis yields the condition:

$$\phi : \ \text{argc} > 0 \wedge \text{argv}^+ = \text{argc} \wedge \text{argv}^- = 1 \wedge \text{optind} \geq 0 \tag{2}$$

The first part of the conjunct (argc > 0) comes from the condition at line 3: Since `usage` is an exit function, we know argc > 1 at line 6, which implies argc > 0 after line 7. The second part ($\text{argv}^+ = \text{argc}$) states that the size of `argv` is still `argc`; this is because `argc` is decremented while `argv` is incremented at line 7. According to the third conjunct ($\text{argv}^- = 1$), `argv` points to the second element in the original argument array due to the pointer increment at line 7. Finally, the last conjunct (optind ≥ 0) is a postcondition established by the call to `getopt`.

Next, we focus on the backwards analysis. To guarantee the safety of the buffer access `files[1]` at line 19, we need $1 < \text{files}^+$ and $1 \geq -\text{files}^-$ to ensure there are no buffer overflows and underflows respectively. Using similar reasoning for the accesses `files[0]` and `argv[optind++]`, our analysis generates the following necessary condition for the safety of the code after line 15:

$$\psi : \begin{array}{c} \text{optind} < \text{argv}^+ \wedge \text{optind} \geq -\text{argv}^- \ \wedge \\ \text{target\_dir} = 0 \rightarrow (1 < \text{argv}^+ - \text{optind} - 1) \ \wedge \\ \text{target\_dir} = 0 \rightarrow 0 \geq -\text{argv}^- - \text{optind} - 1) \end{array} \tag{3}$$

Observe that files^- and files^+ do not appear in this formula because the backwards analysis relates the size and offset of `files` to those of `argv` when computing the weakest precondition of `files = argv + optind` at line 17. Now, to synthesize the unknown guard at line 15, we generate the following constraint:

$$(\phi \wedge \chi(\boldsymbol{v})) \rightarrow \psi \tag{4}$$

where ϕ and ψ come from Equations 2 and 3, χ is the unknown guard, and $\boldsymbol{v}$ represents program variables in scope at line 15. Note that, since $\text{argv}^-, \text{argv}^+$ etc. are ghost variables, they are not allowed to appear in our solution for χ.

Now, inferring a formula to plug in for χ that makes Equation 4 valid is a logical abduction problem. By using abduction to solve for χ, we obtain the solution `argc - optind > 2`. Observe that there are other guards that also guarantee memory safety in this example, such as:

(S1) $\text{argc} > \text{optind} \wedge (\text{target\_dir} = 0 \rightarrow \text{argc} - \text{optind} > 2)$, or
(S2) $\text{argc} = 4 \wedge \text{optind} = 1$

However, both of these solutions are undesirable because (S1) is overly complicated, while (S2) is not sufficiently general.

3 Language and Preliminaries

We describe our techniques using the small imperative language given in Figure 3. Here, a program takes inputs $\boldsymbol{v}$ and consists of one or more statements. We

$$\begin{array}{lll}
\text{Program P} & := & \lambda \boldsymbol{v}.\ S \\
\text{Guard } G & := & ??_i \mid C \\
\text{Statement S} & := & \text{skip} \mid v := E \mid S_1; S_2 \mid [p] := \text{alloc}(E) \mid [p_1] := [p_2] \oplus E \\
 & & \mid \text{access}([p], E) \mid \text{if}(G) \text{ then } S_1 \text{ else } S_2; \\
 & & \mid \text{while}(C) \text{ do } S \mid \text{while}(C \wedge ??_i) \text{ do } S \\
\text{Conditional } C & := & E_1 \text{ comp } E_2 \ (\text{comp} \in \{<, >, =\}) \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C \\
\text{Expression } E & := & \text{int} \mid v \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \cdot E_2
\end{array}$$

Fig. 3: Language used for the formal development

syntactically differentiate between scalar variables v and pointers $[p]$, which are always written inside brackets. Statements include skip, scalar assignments ($v := E$), sequencing, memory allocations ($[p] = \text{alloc}(E)$), and pointer arithmetic ($[p_1] = [p_2] \oplus E$) which makes p_1 point to offset E in the memory block pointed to by $[p_2]$. The statement $\text{access}([p], E)$ accesses the E'th offset of $[p]$. Since our main concern is to guarantee the safety of memory accesses, we use the access statement to model both array reads and writes. In particular, $\text{access}([p], E)$ fails if E is not a valid offset in the memory region pointed to by E. We say that an access is *safe* if it can never fail in any execution; otherwise, it is *unsafe*. A program P is *memory-safe* if all accesses in P are safe.

In this language, unknown predicates $??_i$ occur either as tests in if statements or as continuation conditions of while loops. We say a guard $\mathtt{G}_1$ is an *ancestor* of guard $\mathtt{G}_2$ if $\mathtt{G}_2$ is nested inside $\mathtt{G}_1$; conversely, we say $\mathtt{G}_2$ is a *descendant* of $\mathtt{G}_1$. We call a program *complete* if it does not contain any unknown guards. Given a program P and a mapping σ from unknown guards to concrete predicates, we write $P[\sigma]$ to denote the program obtained by substituting each $??_i$ with $\sigma(??_i)$.

Definition 1 *Mapping σ is a solution to the guard synthesis problem defined by program P iff (i) $P[\sigma]$ is a complete and memory-safe program, and (ii) $\forall v \in \text{dom}(\sigma).\ \sigma(v) \not\Rightarrow \text{false}$.*

According to the second condition, a valid solution cannot instantiate any unknown predicate with false. Hence, the synthesis problem is unsolvable if we cannot guarantee memory safety without creating dead code.

Definition 2 *Given solutions σ and σ' to the synthesis problem, we say that σ* refines *σ', written $\sigma' \preceq \sigma$, if for some unknown $\chi \in \text{dom}(\sigma)$, we have either (i) $\sigma'(\chi) \Rightarrow \sigma(\chi)$ and $\sigma(\chi) \not\Rightarrow \sigma'(\chi)$, or (ii) $|\text{vars}(\sigma(\chi))| < |\text{vars}(\sigma'(\chi))|$.*

In other words, solution σ refines σ' if it improves some guard either in terms of generality or simplicity.

Definition 3 *Solution σ is a* Pareto-optimal solution *to the synthesis problem if for all other solutions σ', we have $\sigma' \preceq \sigma$.*

Intuitively, this means that, if we take solution σ and try to improve any guard in σ according to our cost metric, then the resulting mapping is no longer a solution. In the rest of the paper, we use the word "optimality" to mean Pareto-optimality in the above sense.

$$(1)\quad \frac{}{\phi, \psi \vdash \text{skip} : \phi, \psi, \emptyset}$$

$$(2)\quad \frac{\phi' = \exists v'.(v = E[v'/v] \wedge \phi[v'/v])}{\phi, \psi \vdash v := E : \phi', \psi[E/v], \emptyset}$$

$$(3)\quad \frac{\phi, \psi_1 \vdash S_1 : \phi_1, \psi_2, \mathcal{C}_1 \qquad \phi_1, \psi \vdash S_2 : \phi_2, \psi_1, \mathcal{C}_2}{\phi, \psi \vdash S_1; S_2 : \phi_2, \psi_2, \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$(4)\quad \frac{\phi, \psi \vdash p^- := 0; p^+ := E : \phi', \psi', \emptyset}{\phi, \psi \vdash [p] := \text{alloc}(E) : \phi', \psi', \emptyset}$$

$$(5)\quad \frac{\begin{array}{c}\phi, \psi \vdash (p_1^- := p_2^- + E) : \phi_1, \psi_1, \emptyset \\ \phi_1, \psi_1 \vdash (p_1^+ := p_2^+ - E) : \phi_2, \psi_2, \emptyset\end{array}}{\phi, \psi \vdash [p_1] := [p_2] \oplus E : \phi_2, \psi_2, \emptyset}$$

$$(6)\quad \frac{\varphi_{\text{safe}} = (E \geq -p^- \wedge E < p^+)}{\phi, \psi \vdash \text{access}([p], E) : \phi \wedge \varphi_{\text{safe}}, \psi \wedge \varphi_{\text{safe}}, \emptyset}$$

$$(7a)\quad \frac{\begin{array}{c}\phi \wedge C, \psi \ \vdash S_1 : \phi_1, \psi_1, \mathcal{C}_1 \quad \phi \wedge \neg C, \psi \vdash S_2 : \phi_2, \psi_2, \mathcal{C}_2 \\ \psi' = (C \rightarrow \psi_1) \vee (\neg C \rightarrow \psi_2)\end{array}}{\phi, \psi \vdash \text{if}(C) \text{ then } S_1 \text{ else } S_2 : \phi_1 \vee \phi_2, \psi', \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$(7b)\quad \frac{\begin{array}{c}\phi \wedge \chi_i(\boldsymbol{v}), \text{true} \vdash S_1 : \_, \varphi, \mathcal{C}_1 \\ \text{VC} = (\phi \wedge \chi_i(\boldsymbol{v}) \rightarrow \varphi) \\ \phi \wedge \chi_i(\boldsymbol{v}), \psi \vdash \widetilde{S_1} : \phi_1, \psi_1, \_ \\ \phi \wedge \neg\chi_i(\boldsymbol{v}), \psi \vdash S_2 : \phi_2, \psi_2, \mathcal{C}_2 \\ \psi' = (\chi_i(\boldsymbol{v}) \rightarrow \psi_1) \wedge (\neg\chi_i(\boldsymbol{v}) \rightarrow \psi_2)\end{array}}{\phi, \psi \vdash \text{if}(??_i) \text{ then } S_1 \text{ else } S_2 : \phi_1 \vee \phi_2, \psi', \text{VC} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$(8a)\quad \frac{I \wedge C, I \vdash S : \_, I', \mathcal{C} \qquad I \wedge C \Rightarrow I'}{\phi, \psi \vdash \text{while}(C) \text{ do } S : I \wedge \neg C, I, \mathcal{C}}$$

$$(8b)\quad \frac{\begin{array}{c}\_, I \vdash \widetilde{S} : \_, I', \_ \quad I \wedge C \Rightarrow I' \\ I \wedge C \wedge \chi_i(\boldsymbol{v}), \text{true} \vdash S : \_, \psi, \mathcal{C} \\ \text{VC} = (I \wedge C \wedge \chi_i(\boldsymbol{v}) \rightarrow \psi)\end{array}}{\phi, \psi \vdash \text{while}(C \wedge ??_i) \text{ do } S : I \wedge \neg(C \wedge \chi_i(\boldsymbol{v})), I, \mathcal{C} \cup \text{VC}}$$

$$(9)\quad \frac{\text{true}, \text{true} \vdash S : \phi, \psi, \mathcal{C}}{\vdash \lambda\boldsymbol{v}.S : \phi, \psi, \psi \cup \mathcal{C}}$$

Fig. 4: Inference Rules for Constraint Generation

4 Constraint Generation

The constraint generation phase is shown in Figure 4 as inference rules of the form $\phi, \psi \vdash S : \phi', \psi', \mathcal{C}$ where S is a statement, ϕ, ψ, ϕ', ψ' are formulas, and $\mathcal{C}$ is a set of constraints. The meaning of this judgment is that, if all constraints in $\mathcal{C}$ are valid, then $\{\phi\}S\{\phi'\}$ and $\{\psi'\}S\{\psi\}$ are valid Hoare triples. We call the computation of postcondition ϕ' from ϕ the *forward analysis* and the computation of precondition ψ' from ψ the *backward analysis*. The constraints $\mathcal{C}$ track assumptions about unknown predicates that must hold to ensure memory safety.

Since some of the rules in Figure 4 describe standard pre- and post-condition computation, we only explain some of these rules. Rule (4) for memory allocation $[p] = \text{alloc}(E)$ uses ghost variables p^- and p^+. Since $[p]$ points to the beginning of a memory block of size E, the allocation has the effect of assigning p^- to 0 and p^+ to E. Hence, ϕ' and ψ' are obtained by computing the strongest postcondition of ϕ and weakest precondition of ψ with respect to the statement $p^- := 0; p^+ := E$.

Rule (5) for pointer arithmetic computes the effect of this statement on p_1^- and p_1^+. Since $[p_2]$ points to offset p_2^- in memory block M, $[p_1]$ points to offset $p_2^- + E$ within M. Hence, we obtain ϕ_1 as $\text{sp}(p_1^- := p_2^- + E, \phi)$ and ψ_1 as $\text{wp}(p_1^- := p_2^- + E, \psi)$. Similarly, $\phi_2 = \text{sp}(p_1^+ := p_2^+ - E, \phi_1)$ and $\psi_2 = \text{wp}(p_1^+ := p_2^+ - E, \psi_1)$.

Rule (6) describes memory accesses. To guarantee that ψ holds after the memory access, expression E must evaluate to a valid offset in the memory block pointed to by $[p]$. Using ghost variables p^- and p^+, we can express this as $\varphi_{\text{safe}} \equiv\ E < p^+ \land E \geq -p^-$ Hence, the weakest precondition of ψ with respect to the access statement is $\psi \land \varphi_{\text{safe}}$.

Constraint generation for conditionals with unknown guards is given in Rule (7b). The first line of this rule computes a weakest sufficient condition for ensuring memory safety of statement S_1. Here, we compute the precondition of S_1 with respect to true rather than ψ because the unknown guard is only required to guarantee the safety of S_1 rather than the remainder of the entire program. Thus, the formula φ computed here represents the weakest sufficient condition for ensuring memory safety of S_1. When we analyze S_1, observe that the forward analysis propagates $\phi \land \chi_i(\boldsymbol{v})$ as the precondition of S_1; hence, statement preconditions computed by the forward analysis may refer to unknown predicates χ_i. The constraints $\mathcal{C}_1$ obtained in this rule describe the restrictions that must be satisfied by the unknown guards nested inside S_1.

The second line in rule (7b) generates a constraint (VC) on the unknown predicate χ_i. Specifically, VC stipulates that the conjunction of the unknown guard χ_i and the precondition ϕ should be strong enough to imply the safety of memory accesses within S_1. Note that the generated VC may contain multiple unknown predicates since both ϕ and φ may refer to other χ_j's.

The third line in rule (7b) uses the notation $\widetilde{S}$, which denotes statement S with each $\text{access}([p], E)$ statement within S replaced by skip. Here, we analyze statement S_1 a second time but with two important differences from our previous analysis. First, since we consider $\widetilde{S_1}$, we ignore any memory accesses within S_1. Second, we compute the weakest precondition of $\widetilde{S_1}$ with respect to ψ rather than true because we need to ensure the safety of memory accesses that come after S_1. However, we ignore all memory accesses within S_1 because the synthesized guard already ensures the safety of these accesses. Also, observe that this rule discards constraints generated when analyzing $\widetilde{S_1}$, which is sound because any constraints generated while analyzing $\widetilde{S_1}$ are trivially valid.

Another important point to note about rule (7b) is that the backwards analysis propagates the constraint $(\chi_i(\boldsymbol{v}) \rightarrow \psi_1) \land (\neg\chi_i(\boldsymbol{v}) \rightarrow \psi_2)$ as the weakest precondition for the if statement. Hence, statement postconditions computed by the backwards analysis may also refer to unknown predicates.

Example 1. Consider the following code example:

```
1. $[p] := \mathrm{alloc}(n);$   $[q] := \mathrm{alloc}(n);$   $[p] := [p] \oplus 1;$
2. $\mathrm{if}(??_1)$  then
3.     $n := n + 1;$    $\mathrm{access}([p], 3);$
4.     $\mathrm{if}(??_2)$  then  $\mathrm{access}([q], n - 2)$  else skip
5. else  skip
```

In the forward direction, our analysis computes the following precondition ϕ for the if statement at line 2: $\phi : p^- = 1 \wedge p^+ = n - 1 \wedge q^- = 0 \wedge q^+ = n$. Before the assignment at line 3, the forwards analysis computes the precondition $\chi_1(n) \wedge p^- = 1 \wedge p^+ = n - 1 \wedge q^- = 0 \wedge q^+ = n$ where χ_1 denotes unknown guard $??_1$ and n is the only scalar variable in scope at this point. For the if statement at line 4, we have the following precondition:

$$\begin{aligned}\exists n'.\ (\chi_1(n')\ \wedge n = n' + 1\ \wedge p^- = 1 \wedge\ p^+ = n' - 1 \\ \wedge\ q^- = 0 \wedge q^+ = n' \wedge p^+ > 3 \wedge p^- \geq -3)\end{aligned}$$

Note that, since there is an assignment to n at line 3, the variable n inside the unknown predicate $\chi_1(n)$ gets substituted by n'.

Now, in the backwards direction, the precondition for the then branch of the second if statement is $(n - 2) < q^+ \wedge (n - 2) \geq -q^-$. Hence, when analyzing the if statement at line 4, we generate the following VC:

$$\mathrm{VC}_2 : \begin{array}{l}(\chi_2(n) \wedge \exists n'.\ (\chi_1(n')\ \wedge n = n' + 1\ \wedge p^- = 1 \wedge\ p^+ = n' - 1 \wedge\ q^- = 0 \\ \wedge\ q^+ = n'\ \wedge\ p^+ > 3 \wedge p^- \geq -3)) \rightarrow ((n - 2) < q^+ \wedge (n - 2) \geq -q^-)\end{array}$$

where χ_2 represents $??_2$ and the right-hand-side of the implication is the safety precondition for the then branch. For the if statement at line 2, the backwards analysis computes the precondition for the then branch as $\varphi : 3 < p^+ \wedge 3 \geq -p^-$. Using φ and formula ϕ obtained through the forward analysis, we generate the following VC for the if statement at line 2:

$$\mathrm{VC}_1 : (\chi_1(n) \wedge p^- = 1 \wedge p^+ = n - 1 \wedge q^- = 0 \wedge q^+ = n)\ \rightarrow (3 < p^+ \wedge 3 \geq -p^-)$$

Hence, our algorithm generates the constraints $\mathrm{VC}_1 \cup \mathrm{VC}_2$.

Continuing with the inference rules in Figure 4, rules (8b) and (8a) describe the analysis of while loops with and without unknown safety guards respectively. Rule (8a) gives the standard Hoare rule for while loops, asserting that I is an inductive loop invariant. Since the automatic inference of loop invariants is an orthogonal problem, this paper assumes that loop invariants are provided by an oracle, and our implementation uses standard abstract interpretation based techniques for loop invariant generation.

In rule (8b), our goal is to infer an additional guard as part of the loop continuation condition such that all memory accesses within the loop body are safe. As in rule (8a), the first line of rule (8b) asserts that I is inductive. However, an important difference is that we check the inductiveness of I with respect to $\widetilde{S}$ rather than S because I is not required to be strong enough to prove the safety of memory accesses inside the loop body. In fact, if I was strong enough to prove memory safety, this would mean the additional unknown guard is unnecessary.

The last two lines of rule (8b) compute the safety precondition ψ for the loop body (i.e., $\psi = \mathrm{wp}(\mathrm{true}, S)$) and generate the constraint VC : $I \wedge C \wedge \chi_i(\boldsymbol{v}) \rightarrow \psi$. In other words, together with continuation condition C and known loop invariant I, unknown predicate χ_i should imply the memory safety of the loop body.

Example 2. Consider the following code snippet:

```
1. $[p] = \mathrm{alloc}(n);\ i = 0;$
2. while$(\mathrm{true} \wedge ??_1)$ do
3.     $\mathrm{access}([p], 1);\ [p] = [p] \oplus 1;\ i = i + 1;$
```

Assume we have the loop invariant $I:\ p^- + p^+ = n\ \wedge\ i \geq 0\ \wedge i = p^-$. The safety precondition ψ for the loop body is $1 < p^+ \wedge 1 \geq -p^-$. Hence, rule (8b) from Figure 4 generates the following verification condition:

$$(\chi_1(i, n)\ \wedge\ p^- + p^+ = n\ \wedge\ i \geq 0\ \wedge\ i = p^-) \rightarrow (1 < p^+ \wedge 1 \geq -p^-)$$

The last rule in Figure 4 generates constraints for the entire program. Since we add the program's weakest precondition to $\mathcal{C}$, *the synthesis problem has a solution only if all accesses that are not protected by unknown guards are safe.*

5 Constraint Solving

The rules described in Section 4 generate constraints of the form:

$$\mathcal{C}_i:\quad (F_1(\chi_1, \ldots \chi_{i-1}) \wedge \chi_i(\boldsymbol{v})) \rightarrow F_2(\chi_{i+1}, \ldots \chi_n) \tag{5}$$

where F_1 and F_2 are arbitrary formulas containing program variables and unknowns. In each constraint, there is exactly one key unknown χ_i that does not appear inside boolean connectives or quantifiers. Hence, we refer to $\mathcal{C}_i$ as the constraint associated with χ_i (or the χ_i-constraint). Also, the only unknowns appearing on the right hand side of an implication (i.e., inside F_2) in a χ_i-constraint represent unknown guards that are syntactically nested inside χ_i. Hence, we refer to $\chi_{i+1}, \ldots \chi_n$ as the *descendants* of $\mathcal{C}_i$, denoted $\textsc{Descends}(\mathcal{C}_i)$. In contrast, the unknowns that appear inside F_1 are either ancestors of χ_i or appear in the code before χ_i. We say that $\mathcal{C}_i$ *sequentially depends* on χ_j if χ_j appears inside F_1 and is not an ancestor of χ_i. We write $\textsc{SeqDep}(\mathcal{C}_i)$ to denote the set of χ_j-constraints such that $\mathcal{C}_i$ is sequentially dependent on χ_j.

Example 3. Consider the following code snippet:

```
1. $[a] := \mathrm{alloc}(x);$
2. if$(??_1)$ then access$([a], 1)$ else skip
3. if$(??_2)$ then
4.     if$(??_3)$ then access$([a], x - 3); [b] := \mathrm{alloc}(4);$ else $[b] := \mathrm{alloc}(2);$
5.     access$([b], 3);$
6. else skip
```

Let χ_1, χ_2, χ_3 denote the unknowns $??_1$, $??_2$, and $??_3$, and let $\mathcal{C}_i$ denote each χ_i constraint. Here, we have:

$\mathcal{C}_1 : (a^- = 0 \wedge a^+ = x \wedge \chi_1(x)) \rightarrow (1 < a^+ \wedge 1 \geq -a^-)$
$\mathcal{C}_2 : (a^- = 0 \wedge a^+ = x \wedge \chi_1(x) \wedge \chi_2(x)) \rightarrow ((\chi_3(x) \rightarrow 3 < 4) \wedge (\neg\chi_3(x) \rightarrow 3 < 2))$
$\mathcal{C}_3 : (a^- = 0 \wedge a^+ = x \wedge \chi_1(x) \wedge \chi_2(x) \wedge \chi_3(x)) \rightarrow (x - 3 < a^+ \wedge x - 3 \geq -a^-)$

Therefore, $\textsc{Descends}(\mathcal{C}_1) = \emptyset$, $\textsc{Descends}(\mathcal{C}_2) = \{\mathcal{C}_3\}$, and $\textsc{Descends}(\mathcal{C}_3) = \emptyset$. Also, $\textsc{SeqDeps}(\mathcal{C}_1) = \emptyset$, $\textsc{SeqDeps}(\mathcal{C}_2) = \{\mathcal{C}_1\}$, $\textsc{SeqDeps}(\mathcal{C}_3) = \{\mathcal{C}_1\}$.

Our constraint solving algorithm is given in Figure 5. A key underlying insight is that we only solve a constraint $\mathcal{C}_i$ when all sequential dependencies of $\mathcal{C}_i$ are resolved. The intuition is that if χ_i sequentially depends on χ_j, χ_j will appear in a χ_i-constraint, but not the other way around. Hence, by fixing the solution for χ_j before processing χ_i, we cannot break the optimality of the solution for χ_j.

The Solve algorithm shown in Figure 5 takes as input constraints $\mathcal{C}$ and returns a mapping S from each unknown χ to a concrete predicate or $\emptyset$ if no solution exists. Initially, we add all constraints $\mathcal{C}$ to worklist W and initialize Resolved to $\emptyset$. In each iteration of the Solve loop, we dequeue constraints Δ that have their sequential dependencies resolved (line (3)) and substitute any resolved unknowns in Δ with the solution given by S, yielding a new set Δ' (line 4). Hence, a χ_i-constraint in Δ' does not contain any unknowns that χ_i sequentially depends on. Now, to solve Δ', we first obtain a sound, but not necessarily optimal, solution using the function SolveInit. In particular, although the solutions returned by SolveInit may be stronger than necessary, we iteratively weaken this initial solution using Weaken until we obtain an optimal solution.

The procedure SolveInit processes constraints $\mathcal{C}$ top-down, starting with the outermost guard χ_i. In each iteration, we pick an unsolved constraint $\mathcal{C}_i$ that has only one unknown on the left-hand side of the implication (line 13). However, since we don't yet have a solution for the unknowns $\mathcal{V}$ on the right-hand side, we strengthen $\mathcal{C}_i$ to Φ by universally quantifying $\mathcal{V}$ (line 16).[3] Observe that the universal quantification of $\mathcal{V}$ has the same effect as treating any unknown guard inside χ_i as a non-deterministic choice. The resulting constraint Φ is of the form $(\chi_i(\boldsymbol{v}) \wedge \phi_1) \rightarrow \phi_2$ where ϕ_1 and ϕ_2 do not contain any unknowns; hence, we can solve for unknown χ_i using standard logical abduction. In the algorithm of Figure 5, this is done by calling an abduction procedure called Abduce (line 17). We do not describe the Abduce procedure in this paper and refer the interested reader to [12] for a description of an abduction algorithm which computes a logically weakest solution containing a fewest number of variables. Now, given solution γ for χ_i, we add it to our solution set S and eliminate unknown χ_i from other constraints in $\mathcal{C}$ using the Substitute procedure.

Because of the universal quantification of the unknowns on the right-hand side, the solution S_0 returned by SolveInit may be stronger than necessary. Hence, procedure Weaken iteratively weakens the initial solution until we obtain an optimal solution. In contrast to SolveInit, Weaken processes the constraints bottom-up, starting with the innermost guard first. Specifically, the solution computed by SolveInit for the innermost guard χ_i cannot be further weakened, as it is not possible to obtain a weaker solution for χ_i by plugging in weaker solutions for the unknowns appearing on the left-hand side of a χ_i-constraint. Hence, we add all constraints with no descendants in Δ to Resolved and update S with the corresponding solutions given by S_0 (lines 21-23).

[3] Recall that $\forall \chi_j . \Phi \equiv \Phi[\text{true}/\chi_j] \wedge \Phi[\text{false}/\chi_j]$.

```
procedure  $\textsc{Solve}(\mathcal{C})$:
        input: set of constraints $\mathcal{C}$
        output: mapping $S$ from each $\chi_i$ to $\gamma$ or $\emptyset$ if no solution exists

(1)     $\textsc{Resolved} := \emptyset$;   $S := \emptyset$;   $W := \mathcal{C}$
(2)     while($W \neq \emptyset$ )
(3)         $\Delta := \{\mathcal{C}_i \mid \mathcal{C}_i \in W \wedge \textsc{SeqDep}(\mathcal{C}_i) \subseteq \textsc{Resolved}\}$
(4)         $W := W - G$
(5)         $\Delta' := \textsc{Substitute}(\Delta, S)$
(6)         $S_0 := \textsc{SolveInit}(\Delta')$
(7)         $S := \textsc{Weaken}(\Delta', S_0, S, \textsc{Resolved})$
(8)         if($S = \emptyset$) return $\emptyset$
(9)         $\textsc{Resolved} := \textsc{Resolved} \uplus \Delta$
(10)    return $S$;

procedure  $\textsc{SolveInit}(\mathcal{C})$:
(11)    $S := \emptyset$
(12)    while($\mathcal{C} \neq \emptyset$)
(13)        let $\mathcal{C}_i \in \mathcal{C}$ with one unknown $\chi$ on LHS
(14)        $\mathcal{C} := \mathcal{C} - \mathcal{C}_i$
(15)        $\mathcal{V}$ := unknowns of $\mathcal{C}_i$ on RHS
(16)        $\Phi := \forall \mathcal{V}.\mathcal{C}_i$
(17)        $\gamma := \textsc{Abduce}(\Phi)$
(18)        $S := S \uplus [\chi \mapsto \gamma]$
(19)        $\mathcal{C} := \textsc{Substitute}(\mathcal{C}, S)$
(20)    return $S$

procedure  $\textsc{Weaken}(\Delta, S, S_0, \textsc{Resolved})$
(21)    $\textsc{Done} := \{\Delta_i \mid \Delta_i \in \Delta \wedge \textsc{Descends}(\Delta_i) = \emptyset\}$
(22)    $\textsc{Resolved} := \textsc{Resolved} \uplus \textsc{Done}$
(23)    $S := S \uplus \{[\chi_i \mapsto \gamma_i] \mid \mathcal{C}_i \in \textsc{Done} \wedge S_0(\chi_i) = \gamma_i\}$
(24)    $\Delta := \Delta - \textsc{Done}$;   $S_0 := S_0 - \textsc{Done}$
(25)    while($\Delta \neq \emptyset$)
(26)        let $\textsc{Cur} \in \Delta$ s/t $\textsc{Descends}(\textsc{Cur}) \subseteq \textsc{Resolved}$
(27)        let $\chi = \textsc{Unknown}(\textsc{Cur})$
(28)        $\Delta := \Delta - \textsc{Cur}$;   $S_0 := S_0 - \chi$
(29)        $\theta := \textsc{Cur} \uplus \{\mathcal{C}_i \mid \mathcal{C}_i \in \textsc{Resolved} \wedge \chi \in \textsc{Unknowns}(\mathcal{C}_i)\}$
(30)        $\theta' := \textsc{Substitute}(S \uplus S_0)$
(31)        $\gamma := \textsc{Abduce}(\theta')$
(32)        if $\textsc{Unsat}(\gamma)$ return $\emptyset$
(33)        $S := S \uplus [\chi \mapsto \gamma]$
(34)        $\textsc{Resolved} := \textsc{Resolved} \uplus \textsc{Cur}$
(35)    return $S$
```

Fig. 5: The constraint solving algorithm

The while loop in lines 25-34 iterates over the constraints bottom-up and, in each iteration, picks a χ-constraint Cur all of whose descendants have been resolved (line 26). Now, the solution given by S_0 for χ could be stronger than necessary; thus, we would like to weaken it using the new optimal solutions for χ's descendants. However, since χ will appear on the left-hand side of a constraint $\mathcal{C}_i$ associated with a descendant χ_i of χ, we need to take care not to invalidate the solution for χ_i as we weaken χ. Hence, at line 29, we collect in set θ all resolved constraints in which χ appears. The idea here is that, when we abduce a new solution for χ, we will simultaneously solve all constraints in θ so that we preserve existing solutions for χ's descendants. Now, to solve for χ using abduction, we first eliminate all unknowns in θ except for χ. For this purpose, we substitute all resolved unknowns in θ with their solution given by S (line 30). However, observe that there may also be unknowns in θ that have not yet been resolved; these unknowns correspond to ancestors of χ, which only appear (unnegated) on the outerlevel conjunction of the left-hand side of the constraints in θ. Hence, to eliminate the unresolved unknowns, we will use the initial solution given by S_0 (also line 30). Observe that we can do this without undermining optimality because we will later only further weaken the solutions given by S_0, which cannot cause us to further weaken the solution for χ.

After performing the substitution at line 30, the set of constraints θ' only contains one unknown χ, which we can now solve using standard abduction (line 31).[4] If the resulting solution γ is false, this means our synthesis problem does not have a solution. Otherwise, we add the mapping $\chi \mapsto \gamma$ to our solution set and continue until all constraints in θ have been processed.

Let us call a solution S to the synthesis problem *optimal relative to a set of loop invariants* if, among the set of solutions that any algorithm can generate using these loop invariants, S is optimal. We have:

Theorem 1. *Consider program P such that $\vdash P : \phi, \psi, \mathcal{C}$ according to Figure 4, and let S be the result of* Solve*($\mathcal{C}$). If $S \neq \emptyset$, then $P[S]$ is memory safe. Furthermore, S is an optimal solution to the synthesis problem defined by P relative to the loop invariants used during constraint generation.*

Example 4. Consider the constraints VC_1 and VC_2 from Example 1. Here, neither VC_1 nor VC_2 have sequential dependencies, and since VC_1 contains only one unknown, we first solve for χ_1 in SolveInit, for which abduction yields the solution $n > 4$. Next, we plug this solution into VC_2 (renaming n to n'), which yields the following constraint in Prenex Normal Form:

$$\forall n'.(n' > 4 \ \wedge \ \chi_2(n) \ \wedge \ n = n' + 1 \ \wedge p^- = 1 \wedge \ p^+ = n' - 1 \wedge q^- = 0 \wedge q^+ = n' \\ \wedge p^+ > 3 \wedge p^- \geq -3) \rightarrow (n-2) < q^+ \wedge (n-2) \geq -q^-)$$

Since VC_2 has only one unknown left, we solve it using Abduce and obtain $\chi_2(n) = \text{true}$. Next, we attempt to weaken the solution for χ_1 in Weaken, but since χ_2 does not appear on the right hand side of VC_1, we cannot further weaken the solution for χ_1. Hence, we obtain the solution $[\chi_1 \mapsto n > 4, \chi_2 \mapsto \text{true}]$.

Example 5. Consider constraints $\mathcal{C}_1$,$\mathcal{C}_2$, and $\mathcal{C}_3$ from Example 3. Since $\mathcal{C}_1$ does not have sequential dependencies, we first solve $\mathcal{C}_1$ and obtain the solution $\chi_1 =$

[4] Observe that we can simulatenously solve all constraints in θ using abduction because a pair of constraints of the form $\chi \wedge \phi_1 \ \Rightarrow \phi_2$ and $\chi \wedge \psi_1 \Rightarrow \psi_2$ (where the ϕ's and ψ's are unknown free) can be rewritten as $(\chi \wedge \phi_1 \wedge \psi_1) \Rightarrow (\phi_2 \wedge \psi_2)$, which corresponds to a standard abduction problem.

Program	Lines	# holes	Time (s)	Memory	Synthesis successful?	Bug?
Coreutils hostname	160	1	0.15	10 MB	Yes	No
Coreutils tee	223	1	0.84	10 MB	Yes	Yes
Coreutils runcon	265	2	0.81	12 MB	Yes	No
Coreutils chroot	279	2	0.53	23 MB	Yes	No
Coreutils remove	710	2	1.38	66MB	Yes	No
Coreutils nl	758	3	2.07	80 MB	Yes	No
SSH - sshconnect	810	3	1.43	81 MB	Yes	No
Coreutils mv	929	4	2.03	42 MB	Yes	No
SSH - do_authentication	1,904	4	3.92	86 MB	Yes	Yes
SSH - ssh_session	2,260	5	4.35	81 MB	Yes	No

Fig. 6: Experimental benchmarks and results

$(x > 1)$. In the next iteration, both $\mathcal{C}_2$ and $\mathcal{C}_3$ have their sequential dependencies resolved; hence we plug in $x > 1$ for χ_1 in $\mathcal{C}_2$ and $\mathcal{C}_3$. In SolveInit, we first solve $\mathcal{C}_2$ since it now contains only one unknown (χ_2) on the left hand side. When we universally quantify χ_3 on the right hand side, Abduce yields the solution $\chi_2 =$ false. In the next iteration of SolveInit, we obtain the solution *true* for χ_3. Observe that our initial solution for χ_2 is stronger than necessary; hence we will weaken it. In the procedure Weaken, we simultaneously solve constraints $\mathcal{C}_2$ and $\mathcal{C}_3$, using existing solutions for χ_1 and χ_3. Abduce now yields $\chi_2 = x > 2$. Hence, the final solution is $[\chi_1 = x > 1,\ \chi_2 = x > 2,\ \chi_3 = \text{true}]$.

Example 6. For the constraint generated in Example 2, the Solve procedure computes the solution $\chi_1(i, n) \ = \ i < n - 1$.

6 Implementation and Evaluation

We implemented a prototype tool for synthesizing safety guards for C programs. Our tool is based on the SAIL infrastructure [13] and uses the Mistral SMT solver [12] for solving abduction problems in linear integer arithmetic.

We evaluated our tool on ten benchmark programs written in C. As shown in Figure 6, all of our benchmarks are taken either from the Unix coreutils, which implements basic command line utilities for Unix [14], or OpenSSH, which provides encrypted communication sessions based on the SSH protocol [15]. For each benchmark program, we manually removed 1-5 safety guards from the source code and then used our algorithm to infer these missing guards. [5] In total, we used our tool to synthesize 27 different safety guards.

The results of our experimental evaluation are shown in Figure 6. Our tool was able to successfully synthesize all of the missing guards present in these benchmarks. For 23 of these 27 missing guards, our tool inferred the *exact same predicate* that the programmer had originally written, and for 4 out of the 27 missing guards, it inferred a syntactically different but semantically equivalent

[5] The URL `http://www.cs.utexas.edu/~tdillig/cav14-benchmarks.tar.gz` contains all benchmarks, where each missing guard is indicated with `_SYN`.

condition (e.g., our tool synthesized the guard $x \neq 0$ when the programmer had originally written $x > 0$ but x is already known to be non-negative). In two applications (Coreutils tee and SSH do_authentication), the guards synthesized by our tool did not match the guards in the original program. However, upon further inspection, we found that both of these programs were in fact buggy. For example, in Coreutils tee, the program could indeed access the `argv` array out-of-bounds. We believe that the existence of such bugs in even extremely well-tested applications is evidence that writing memory safe code is hard and that many programmers can benefit from our guard synthesis technique.

As shown in Figure 6, the running time of our algorithm ranges from 0.15 seconds to 4.35 seconds with an average memory consumption of 49 MB. We believe these results suggest that our approach can be integrated into the development process, helping programmers write safe-by-construction low level code.

7 Related work

Program Synthesis. The last few years have seen a flurry of activity in constraint-based software synthesis [9, 10, 16, 17]. As the first abduction-based approach to synthesis, our work is algorithmically very different from prior methods in this area. A concrete benefit is that, unlike prior constraint-based approaches to synthesis [10, 18, 11, 19], our method does not require a template for the expressions being synthesized. A second benefit is that we can show the synthesized expressions to be optimal relative to loop invariants.

There are a few approaches to synthesis that consider optimality as an objective [20–23]. However, in these papers, optimality is defined with respect to an explicit quantitative aspect of program executions, for example execution time. In contrast, in the current work, the cost metric is on the guards that we synthesize; we want to infer guards that are as simple and as general as possible.

Program Analysis for Memory Safety. Memory safety is a core concern in low-level programming, and there is a huge literature on program analysis techniques to guarantee memory safety [24–31, 3, 32, 33]. While many of these techniques can statically detect memory safety errors, they do not help the programmer write safe-by-construction code. Furthermore, unlike dynamic approaches to memory safety [32, 33, 27, 28], our technique guarantees the absence of runtime failures and does not require additional runtime book-keeping.

Abduction-based Verification. Many memory safety verifiers based on separation logic use abductive (or bi-abductive) reasoning for performing modular heap reasoning [5, 34, 35]. In these approaches, abductive reasoning is used to infer missing preconditions of procedures. A more algorithmic form of abduction for first-order theories is considered in [12]. The abduction algorithm described in [12] computes a maximally simple and general solution and is used as a key building block in the constraint solving phase of our synthesis algorithm. This form of SMT-based abduction has also been used for loop invariant generation [6, 7] and for error explanation and diagnosis [36]. The contribution of the present paper is to show how abduction can be used in program synthesis.

References

1. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: Computer Aided Verification, Springer (2011) 178–183
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with blast. In: FASE. Springer (2005) 2–18
3. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2007) 19–33
4. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. ESOP (2010) 246–266
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL (2009) 289–300
6. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: OOPSLA, ACM (2013) 443–456
7. Li, B., Dillig, I., Dillig, T., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: TACAS, Springer (2013) 370–384
8. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. (2013)
9. Solar-Lezama, A.: The sketching approach to program synthesis. In: APLAS. (2009) 4–13
10. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT **15**(5-6) (2013) 497–518
11. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.: Path-based inductive synthesis for program inversion. In: PLDI. (2011) 492–503
12. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: Computer Aided Verification, Springer (2013) 684–689
13. Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language. Stanford University Technical Report
14. http://www.gnu.org/software/coreutils/: Unix coreutils
15. http://www.openssh.com/: Openssh 5.3p1
16. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL. (2010) 327–338
17. Seshia, S.: Sciduction: combining induction, deduction, and structure for verification and synthesis. In: DAC. (2012) 356–365
18. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. In: POPL. (2010) 313–326
19. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL. (2014)
20. Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: CAV. (2009) 140–156
21. Jha, S., Seshia, S., Tiwari, A.: Synthesis of optimal switching logic for hybrid systems. In: Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on. (2011) 107–116
22. Chaudhuri, S., Solar-Lezama, A.: Smooth interpretation. In: PLDI. (2010)
23. S. Chaudhuri, M. Clochard, A.S.L.: Bridging boolean and quantitative synthesis using smoothed proof search. In: POPL. (2014)
24. Younan, Y., Joosen, W., Piessens, F.: Runtime countermeasures for code injection attacks against c and c++ programs. ACM Comput. Surv. **44**(3) (2012) 17

25. Dhurjati, D., Kowshik, S., Adve, V., Lattner, C.: Memory safety without runtime checks or garbage collection. In: LCTES'03. 69–80
26. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Compiler Construction. (2004) 5–23
27. Condit, J., Harren, M., Anderson, Z., Gay, D., Necula, G.: Dependent types for low-level programming. In: ESOP. (2007) 520–535
28. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: Checking memory safety with blast. In: FASE. Springer (2005) 2–18
29. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., OHearn, P.: Scalable shape analysis for systems code. In: Computer Aided Verification, Springer (2008) 385–398
30. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: Computer Aided Verification, Springer (2011) 178–183
31. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: In Network and Distributed System Security Symposium. (2000) 3–17
32. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. POPL (2002) 128–139
33. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: USENIX Annual Technical Conference, General Track. (2002) 275–288
34. Distefano, D., Filipović, I.: Memory leaks detection in java by bi-abductive inference. In: FASE. (2010) 278–292
35. Botinčan, M., Distefano, D., Dodds, M., Grigore, R., Parkinson, M.J.: corestar: The core of jstar. In: In Boogie, Citeseer (2011)
36. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, ACM 181–192