Abductive Inference and its Applications in Program Analysis, Verification, and Synthesis

Isil Dillig
University of Texas, Austin

* Joint work with Ken McMillan, Tom Dillig, Boyang Li, Alex Aiken, Mooly Sagiv
Abduction: Inference of missing hypotheses
What is Abduction?

- **Abduction**: Inference of missing hypotheses

Given known facts $\Gamma$ and desired outcome $\phi$, abductive inference finds “simple” explanatory hypothesis $\psi$ such that

$$\Gamma \land \psi \models \phi \quad \text{and} \quad \text{SAT}(\Gamma \land \psi)$$
Abduction: Inference of missing hypotheses

Given known facts $\Gamma$ and desired outcome $\phi$, abductive inference finds “simple” explanatory hypothesis $\psi$ such that

$$\Gamma \land \psi \models \phi \quad \text{and} \quad \text{SAT}(\Gamma \land \psi)$$

i.e., given invalid formula $\Gamma \Rightarrow \phi$, find a “simple” formula $\psi$ such that $\Gamma \land \psi \Rightarrow \phi$ is valid and $\psi$ does not contradict $\Gamma$. 

What is Abduction?
Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”:
\((R \Rightarrow W \land C) \land (W \Rightarrow S)\)

Conclusion: “It is cloudy and slippery”, i.e., \(C \land S\)

Conclusion doesn't follow from premises; use abduction to find missing hypothesis
Possible solution: \(R\), i.e., “It is rainy”
Simple Example

- Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”:
  \((R \Rightarrow W \land C) \land (W \Rightarrow S)\)

- Conclusion: “It is cloudy and slippery”, i.e., \(C \land S\)
Simple Example

- Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”: 
  
  \((R \Rightarrow W \land C) \land (W \Rightarrow S)\)

- Conclusion: “It is cloudy and slippery”, i.e., \(C \land S\)

- Conclusion doesn’t follow from premises; use abduction to find missing hypothesis
Simple Example

- Premises: “If it rains, then it is wet and cloudy”, “If it is wet, then it is slippery”: 
  \((R \Rightarrow W \land C) \land (W \Rightarrow S)\)

- Conclusion: “It is cloudy and slippery”, i.e., \(C \land S\)

- Conclusion doesn’t follow from premises; use abduction to find missing hypothesis

- Possible solution: \(R\), i.e., “It is rainy”
Suppose we know $x \geq -2$
Arithmetic Example

Suppose we know $x \geq -2$

Want to prove: $x + y > 10$
Suppose we know $x \geq -2$

Want to prove: $x + y > 10$

Abductive explanation: $y > 12$
1. Properties of desired solutions
Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction
Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction
5. Use of abduction in program synthesis
Outline of Talk

1. Properties of desired solutions
2. Algorithm for performing abduction in LRA/LIA
3. Loop invariant generation using abduction
4. Compositional verification using abduction
5. Use of abduction in program synthesis
6. Conclusion and future directions
In general, the abduction problem $\Gamma \land ? \models \phi$ has infinitely many solutions.
In general, the abduction problem $\Gamma \land ? \models \phi$ has infinitely many solutions.

**Trivial solution:** $\phi$, but not useful because it does not take into account what we know.
In general, the abduction problem $\Gamma \land \phi \models \phi$ has infinitely many solutions.

- **Trivial solution:** $\phi$, but not useful because does not take into account what we know.

- So, what kind of solutions do you want to compute?
Which Abductive Explanations Are Good?

Guiding Principle:
Occam’s Razor
Guiding Principle: Occam’s Razor

- If there are multiple competing hypotheses, select the one that makes fewest assumptions
Which Abductive Explanations Are Good?

**Guiding Principle:**

**Occam’s Razor**

- If there are multiple competing hypotheses, select the one that makes fewest assumptions.

- **Generality:** If explanation $A$ is logically weaker than explanation $B$, always prefer $A$.
Which Abductive Explanations Are Good?

Guiding Principle: Occam’s Razor

- If there are multiple competing hypotheses, select the one that makes fewest assumptions

  - Generality: If explanation $A$ is logically weaker than explanation $B$, always prefer $A$

  - Succinctness: Minimize number of variables
Solutions We Will Compute

Want to compute logically weakest solutions with fewest variables
Want to compute logically weakest solutions with fewest variables

First talk about how to compute solutions with fewest variables
Want to compute logically weakest solutions with fewest variables

- First talk about how to compute solutions with fewest variables
- Then talk about how to obtain most general solution containing these variables
To find solutions with fewest variables, we use minimum satisfying assignments of formulas.
To find solutions with fewest variables, we use **minimum satisfying assignments** of formulas

**Minimum satisfying assignment (MSA):**

- assigns values to a subset of variables in formula
- sufficient to make formula true
- Among all other partial satisfying assignments, contains fewest variables
Consider the following formula in linear integer arithmetic:

\[ x + y + w > 0 \lor x + y + z + w < 5 \]
Example

Consider the following formula in linear integer arithmetic:

\[ x + y + w > 0 \lor x + y + z + w < 5 \]

Minimum satisfying assignment:  \[ z = 0 \]
Consider the following formula in linear integer arithmetic:

\[ x + y + w > 0 \lor x + y + z + w < 5 \]

Minimum satisfying assignment: \( z = 0 \)

Note: Algorithm for computing MSAs given in our CAV’12 paper, “Minimum Satisfying Assignments for SMT”
Given facts $\Gamma$ and conclusion $\phi$, MSA $\sigma$ of $\Gamma \Rightarrow \phi$ consistent with $\Gamma$ is a solution to abduction problem:

$$\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \land \Gamma \models \phi$$
Given facts $\Gamma$ and conclusion $\phi$, MSA $\sigma$ of $\Gamma \Rightarrow \phi$ consistent with $\Gamma$ is a solution to abduction problem:

$$\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \land \Gamma \models \phi$$

Furthermore, it uses a fewest number of variables
Why Are MSAs Useful for Abduction?

Given facts \( \Gamma \) and conclusion \( \phi \), MSA \( \sigma \) of \( \Gamma \Rightarrow \phi \) consistent with \( \Gamma \) is a solution to abduction problem:

\[
\sigma \models \Gamma \Rightarrow \phi \quad \text{hence} \quad \sigma \land \Gamma \models \phi
\]

Furthermore, it uses a fewest number of variables

But it is not the most general solution
Finding Most General Solutions

Key idea: Quantifier elimination

- To find most general solution containing variables in the MSA, universally quantify all other variables $\overline{V}$ and apply quantifier elimination to $\forall \overline{V}. \Gamma \Rightarrow \phi$
Finding Most General Solutions

Key idea: Quantifier elimination

- To find most general solution containing variables in the MSA, universally quantify all other variables \( \overline{V} \) and apply quantifier elimination to \( \forall \overline{V}. \Gamma \Rightarrow \phi \)

- This yields most general solution with fewest variables
Abduction Algorithm

- **abduce** yields formula $\psi$ such that

  $$\Gamma \land \psi \models \phi$$

  and $\psi$ is consistent with $\Gamma$ and $\theta$

abduce(\Gamma, \phi, \theta)\{

}\}
Abduction Algorithm

- **abduce** yields formula \( \psi \) such that
  \[
  \Gamma \land \psi \models \phi
  \]
  and \( \psi \) is consistent with \( \Gamma \) and \( \theta \)

- First, compute all variables in MSA of \( \Gamma \Rightarrow \phi \) consistent with \( \Gamma, \theta \)

```latex
\text{abduce}(\Gamma, \phi, \theta)\{
  V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\})
}\}
```
Abduction Algorithm

- **abduce** yields formula $\psi$ such that

  $$\Gamma \land \psi \models \phi$$

  and $\psi$ is consistent with $\Gamma$ and $\theta$

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with $\Gamma, \theta$

- $\forall$-quantify variables not in the MSA and apply quantifier elimination

```latex
\text{abduce}(\Gamma, \phi, \theta)\
\text{ } \\
V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\}) \\
\psi = \text{QE}(\forall V. (\Gamma \Rightarrow \phi))
```

13 / 47
Abduction Algorithm

- **abduce** yields formula $\psi$ such that

$$\Gamma \land \psi \models \phi$$

and $\psi$ is consistent with $\Gamma$ and $\theta$

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with $\Gamma, \theta$

- $\forall$-quantify variables not in the MSA and apply quantifier elimination

- Remove subparts of $\psi$ implied or contradicted by $\Gamma$ (SAS’10)

abduce($\Gamma, \phi, \theta$)

\[
V = \text{msa}(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\})
\]

\[
\psi = \text{QE}(\forall V. (\Gamma \Rightarrow \phi))
\]

\[
\psi' = \text{simplify}(\psi, \Gamma)
\]

}
Abduction Algorithm

- **abduce** yields formula $\psi$ such that

  $$\Gamma \land \psi \models \phi$$

  and $\psi$ is consistent with $\Gamma$ and $\theta$

- First, compute all variables in MSA of $\Gamma \Rightarrow \phi$ consistent with $\Gamma, \theta$

- $\forall$-quantify variables not in the MSA and apply quantifier elimination

- Remove subparts of $\psi$ implied or contradicted by $\Gamma$ (SAS’10)

```plaintext
abduce(\Gamma, \phi, \theta)\
V = msa(\Gamma \Rightarrow \phi, \theta \cup \{\Gamma\})
\psi = QE(\forall V. (\Gamma \Rightarrow \phi))
\psi' = simplify(\psi, \Gamma)
return \psi'
```
Consider abduction problem in LIA defined by conclusion $\phi : x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$.
Consider abduction problem in LIA defined by conclusion $\phi : x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$

Compute MSA for $\Gamma \Rightarrow \phi$:
Consider abduction problem in LIA defined by conclusion $\phi : x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$.

Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$.
Consider abduction problem in LIA defined by conclusion \( \phi : x + y > 10 \) and known facts \( \Gamma : x \geq -2 \land x = y \).

Compute MSA for \( \Gamma \Rightarrow \phi \): \( y = 15 \).

Universally quantify \( x \) and eliminate from \( \Gamma \Rightarrow \phi \):
Consider abduction problem in LIA defined by conclusion
$\phi : x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$

Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$

Universally quantify $x$ and eliminate from $\Gamma \Rightarrow \phi$:
$y < -2 \lor y > 5$
Consider abduction problem in LIA defined by conclusion $\phi : x + y > 10$ and known facts $\Gamma : x \geq -2 \land x = y$

Compute MSA for $\Gamma \Rightarrow \phi$: $y = 15$

Universally quantify $x$ and eliminate from $\Gamma \Rightarrow \phi$: $y < -2 \lor y > 5$

Simplify with respect to assumptions:
Consider abduction problem in LIA defined by conclusion \( \phi : x + y > 10 \) and known facts \( \Gamma : x \geq -2 \land x = y \).

Compute MSA for \( \Gamma \Rightarrow \phi : y = 15 \).

Universally quantify \( x \) and eliminate from \( \Gamma \Rightarrow \phi : y < -2 \lor y > 5 \).

Simplify with respect to assumptions: \( y > 5 \).
Applications

- Abduction
- Interpolation
- CEGAR
- Abstract interpretation

Applications:
- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
- Inference of missing library specifications (APLAS’13)
- Diagnosis of static analysis warnings (PLDI’12)
- Synthesis of missing guards (CAV’14)
Applications

- Loop invariant generation (OOPSLA’13)
Applications

- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
Applications

- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
- Inference of missing library specifications (APLAS’13)
Applications

- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
- Inference of missing library specifications (APLAS’13)
- Diagnosis of static analysis warnings (PLDI’12)
Applications

- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
- Inference of missing library specifications (APLAS’13)
- Diagnosis of static analysis warnings (PLDI’12)
- Synthesis of missing guards (CAV’14)
Applications

- Loop invariant generation (OOPSLA’13)
- Compositional program verification (TACAS’13)
- Inference of missing library specifications (APLAS’13)
- Diagnosis of static analysis warnings (PLDI’12)
- Synthesis of missing guards (CAV’14)
Most challenging aspect of program verification: loop invariant generation.

Inductive loop invariant $Inv$ is implied by $Pre$ and preserved in each iteration assuming only $Inv$.

But $Inv$ is only useful if it is sufficient to prove $Post$. 
Most challenging aspect of program verification: loop invariant generation
Application #1: Loop Invariant Generation

- Most challenging aspect of program verification: **loop invariant generation**

- Inductive loop invariant \( \text{Inv} \) is implied by \( \text{Pre} \) and preserved in each iteration assuming only \( \text{Inv} \)
Most challenging aspect of program verification: loop invariant generation

Inductive loop invariant \( \text{Inv} \) is implied by \( \text{Pre} \) and preserved in each iteration assuming only \( \text{Inv} \)

But \( \text{Inv} \) is only useful if it is sufficient to prove \( \text{Post} \)
Key idea 1: Given loop $L$ and postcondition $Post$, use abduction to speculate candidate invariants.
Key idea 1: Given loop $L$ and postcondition $Post$, use *abduction* to speculate candidate invariants.

```plaintext
assume(P);
while(C) {
    S;
}
assert(Q);
```
Key idea 1: Given loop \( L \) and postcondition \( \text{Post} \), use abduction to speculate candidate invariants.

Use abduction to speculate an invariant \( I \) that implies post-condition \( Q \):

\[
(\neg C \land ?) \Rightarrow Q
\]

```c
assume(P);
while(C) {
    S;
}
assert(Q);
```
Key idea 1: Given loop $L$ and postcondition $Post$, use abduction to speculate candidate invariants

- Use abduction to speculate an invariant $I$ that implies post-condition $Q$:
  \[(\neg C \land \mathcal{R}) \Rightarrow Q\]

- Three possibilities:

```plaintext
assume(P);
while(C) {
    S;
}
assert(Q);
```
Key idea 1: Given loop \( L \) and postcondition \( \text{Post} \), use abduction to speculate candidate invariants.

- Use abduction to speculate an invariant \( I \) that implies post-condition \( Q \):
  \[
  (\neg C \land ?) \Rightarrow Q
  \]

- Three possibilities:
  - \( I \) is inductive: \( I \land C \Rightarrow \text{wp}(S, I), \ P \Rightarrow I \)
High Level Idea

**Key idea 1:** Given loop \( L \) and postcondition \( Post \), use **abduction** to speculate candidate invariants

```plaintext
assume(P);
while(C) {
    S;
}
assert(Q);
```

- Use abduction to speculate an invariant \( I \) that implies post-condition \( Q \):
  
  \[
  (\neg C \land ?) \Rightarrow Q
  \]

- Three possibilities:
  - 😊 \( I \) is inductive: \( I \land C \Rightarrow wp(S, I), \quad P \Rightarrow I \)
  - 😞 \( I \) is an invariant, but not inductive
High Level Idea

**Key idea 1:** Given loop $L$ and postcondition $Post$, use **abduction** to speculate candidate invariants

- Use abduction to speculate an invariant $I$ that implies post-condition $Q$:
  
  $$(\neg C \land ?) \Rightarrow Q$$

- Three possibilities:
  - ☑️ $I$ is inductive: $I \land C \Rightarrow wp(S, I), \ P \Rightarrow I$
  - 😞 $I$ is an invariant, but not inductive
  - 😞 $I$ is not an invariant

```plaintext
assume(P);
while(C) {
  S;
}
assert(Q);
```
If candidate invariant $I$ is not inductive, our algorithm tries to **strengthen** it.
If candidate invariant $I$ is not inductive, our algorithm tries to **strengthen** it.

Use abduction again to find a strengthening $I'$ of $I$:

$$(?? \land (I \land C)) \Rightarrow wp(S, I)$$
If candidate invariant $I$ is not inductive, our algorithm tries to **strengthen** it.

Use abduction again to find a strengthening $I'$ of $I$:

$$(?? \land (I \land C)) \Rightarrow wp(S, I)$$

If $I'$ is an invariant, then so is $I$, i.e.,

$I$ is inductive **relative to** $I'$.
If candidate invariant $I$ is not inductive, our algorithm tries to strengthen it.

Use abduction again to find a strengthening $I'$ of $I$:

$$
(?? \land (I \land C)) \Rightarrow \text{wp}(S, I)
$$

If $I'$ is an invariant, then so is $I$, i.e.,

$I$ is inductive relative to $I'$

Now, check if $I \land I'$ is inductive, if not, keep strengthening using (i) proof goes through, or (ii) get contradiction.
Start by solving abduction problem:

\[ i > n \land ?? \Rightarrow i \geq 1 \]

\{i=1, j = 0, n<5\}

```
while(i <= n) {
    j := j+i;
    i := i+j;
}
{\{i >= 1\}}
```
Simple Example

- Start by solving abduction problem:

\[ i > n \land ?? \Rightarrow i \geq 1 \]

- Algorithm returns solution \( i \geq 1 \)
Start by solving abduction problem:

\[ i > n \land ?? \Rightarrow i \geq 1 \]

Algorithm returns solution \( i \geq 1 \)

Not inductive; so try strengthening:

\[ i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1 \]
Simple Example

Start by solving abduction problem:

\[ i > n \land ?? \Rightarrow i \geq 1 \]

Algorithm returns solution \( i \geq 1 \)

Not inductive; so try strengthening:

\[ i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1 \]

Solution is \( j \geq -1 \), so new candidate invariant:

\[ i \geq 1 \land j \geq -1 \]
Start by solving abduction problem:

\[ i > n \land ?? \Rightarrow i \geq 1 \]

Algorithm returns solution \( i \geq 1 \)

Not inductive; so try strengthening:

\[ i \leq n \land i \geq 1 \land ?? \Rightarrow j + 2i \geq 1 \]

Solution is \( j \geq -1 \), so new candidate invariant:

\[ i \geq 1 \land j \geq -1 \]

This is inductive, so algorithm terminates
Recall: Abduction procedure takes $\Gamma$, $\phi$, and set $\theta$
How to Perform Backtracking

- **Recall:** Abduction procedure takes $\Gamma$, $\phi$, and set $\theta$

- Solution must be consistent with every $\varphi \in \theta$

$\text{abduce}(\Gamma, \phi, \theta)$
How to Perform Backtracking

- **Recall:** Abduction procedure takes $\Gamma$, $\phi$, and set $\theta$

- Solution must be consistent with every $\varphi \in \theta$

- If $I$ is refuted, add $\neg I$ to $\theta$ to obtain different solution
How to Perform Backtracking

- **Recall:** Abduction procedure takes $\Gamma$, $\phi$, and set $\theta$

- Solution must be consistent with every $\varphi \in \theta$

- If $I$ is refuted, add $\neg I$ to $\theta$ to obtain different solution

Algorithm lazily generates abductive explanations
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks
- Compared our results against BLAST, InvGen, and Interproc:

![Bar Chart]

- HOLA
- BLAST
- InvGen
- Interproc

Can verify 13 benchmarks that no other tool can verify, but cannot prove two benchmarks at least one tool can show.

No termination guarantees.
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks

- Compared our results against BLAST, InvGen, and Interproc:

<table>
<thead>
<tr>
<th></th>
<th>HOLA</th>
<th>BLAST</th>
<th>InvGen</th>
<th>Interproc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>53.5</td>
<td>43.5</td>
<td>47.8</td>
<td>37</td>
</tr>
</tbody>
</table>

- Can verify 13 benchmarks that no other tool can verify, but cannot prove two benchmarks at least one tool can show
Some Experimental Results

- Evaluated this technique on 46 loop invariant benchmarks
- Compared our results against BLAST, InvGen, and Interproc:
  - Can verify 13 benchmarks that no other tool can verify, but cannot prove two benchmarks at least one tool can show
  - No termination guarantees
Compositional approaches decompose proof into lemmas. Two key advantages:

1. Scalability: Each lemma concerns small syntactic part $\Rightarrow$ reason about program fragments in isolation.
2. Abstraction: Each lemma can be proven using a different abstraction $\Rightarrow$ combine strengths of different techniques.
Compositional approaches decompose proof into lemmas
Application #2: Compositional Verification

Compositional approaches decompose proof into lemmas

Two key advantages:

- Scalability: Each lemma concerns small syntactic part \( \Rightarrow \) reason about program fragments in isolation
- Abstraction: Each lemma can be proven using a different abstraction \( \Rightarrow \) combine strengths of different techniques
Application #2: Compositional Verification

Compositional approaches decompose proof into lemmas

Two key advantages:

1. **Scalability**: Each lemma concerns small syntactic part → reason about program fragments in isolation
Compositional approaches decompose proof into lemmas

Two key advantages:

1. **Scalability**: Each lemma concerns small syntactic part $\Rightarrow$ reason about program fragments in isolation

2. **Abstraction**: Each lemma can be proven using a different abstraction $\Rightarrow$ combine strengths of different techniques
**Key idea:** Use *abduction* to decompose proof into auxiliary lemmas
Key idea: Use abduction to decompose proof into auxiliary lemmas.

- Lemmas are snippets annotated with assertions and assumptions.
Key idea: Use abduction to decompose proof into auxiliary lemmas

- Lemmas are snippets annotated with assertions and assumptions

- Lemmas are discharged using portfolio of client analyses
**Key idea:** Use *abduction* to decompose proof into auxiliary lemmas

- Lemmas are snippets annotated with assertions and assumptions
- Lemmas are discharged using portfolio of client analyses
- Combine lemmas into overall proof using *circular compositional reasoning*
Overview of Compositional Verification Approach

- **Key idea:** Use **abduction** to decompose proof into auxiliary lemmas

  - Lemmas are snippets annotated with assertions and assumptions

  - Lemmas are discharged using portfolio of client analyses

  - Combine lemmas into overall proof using **circular compositional reasoning**

    - Each lemma can assume correctness of all other lemmas
Consider this code snippet:

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

Want to reason about two fragments in isolation.

Focus on fragment containing assertion.

Cannot verify it yet because need precondition "z is odd".

Want to automatically infer such missing assumptions!
Consider this code snippet:

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;

int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

Want to reason about two fragments in isolation:

**Fragment 1**

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;

int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

**Fragment 2**

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
```

Cannot verify it yet because we need precondition “z is odd”.

Want to automatically infer such missing assumptions!
Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;

int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```
Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion
- Cannot verify it yet because need precondition “z is odd”

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```
Example

- Consider this code snippet
- Want to reason about two fragments in isolation
- Focus on fragment containing assertion
- Cannot verify it yet because need precondition “z is odd”
- Want to automatically infer such missing assumptions!

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;

int x=0, y=0, w=0;
while(*) {
    assert(x==y);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```
Idea: Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)
Example cont.
Program Decoration and VC Generation

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume();
while(*) {
  assert(x==y);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
assume();
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)

- Generate VCs over unknowns $\phi_1$ and $\phi_2$
Example cont.
Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ₁);
while(*) {
  assert(x==y);
  assume(φ₂);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
assume( );
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)

- Generate VCs over unknowns $\phi_1$ and $\phi_2$

- **VC 1:**

  $$(x = 0 \land y = 0 \\
  \land w = 0 \land \phi_1) \Rightarrow x = y$$

Example cont.
Program Decoration and VC Generation

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume( );
while(*) {
  assert(x==y);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
assume( );
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)

- Generate VCs over unknowns $\phi_1$ and $\phi_2$

- VC 1:

\[
(x = 0 \land y = 0 \\
\land w = 0 \land \phi_1) \Rightarrow x = y
\]

- VC 2:

\[
(\phi_2 \land x = y) \Rightarrow \text{wp}(\sigma, x = y)
\]
Idea: Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)

Generate VCs over unknowns $\phi_1$ and $\phi_2$

VC 1: VALID

$$(x = 0 \land y = 0 \land w = 0 \land \phi_1) \Rightarrow x = y$$

VC 2:

$$(\phi_2 \land x = y) \Rightarrow wp(\sigma, x = y)$$
Example cont.
Program Decoration and VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
assume( );
```

- **Idea:** Decorate program with assume statements containing placeholders (e.g., $\phi_1, \phi_2$)

- Generate VCs over unknowns $\phi_1$ and $\phi_2$

- **VC 1:** VALID

\[
(x = 0 \land y = 0 \\
\land w = 0 \land \phi_1) \Rightarrow x = y
\]

- **VC 2:** NOT VALID

\[
(\phi_2 \land x = y) \Rightarrow wp(\sigma, x = y)
\]
Fix VC2 using abduction:

\[ \phi_2 : (w + z) \% 2 = 1 \]
Fix VC2 using abduction:

\[ \phi_2 : (w + z) \% 2 = 1 \]

Now use **circular compositional reasoning**
Example, cont.
Lemma Inference using Abduction

- Fix VC2 using abduction:
  \[ \phi_2 : (w + z) \mod 2 = 1 \]

- Now use circular compositional reasoning

- **Subgoal 1**: Prove \( x = y \) using \( (w + z) \mod 2 = 1 \)

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);
  assume((w+z)%2=1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```
Example, cont.
Lemma Inference using Abduction

- Fix VC2 using abduction:

\[ \phi_2 : (w + z) \% 2 = 1 \]

- Now use **circular compositional reasoning**

  - **Subgoal 1**: Prove \( x = y \) using \((w + z) \% 2 = 1\)

  - **Subgoal 2**: Prove \( \phi_2 \) assuming \( x = y \)

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
    assume(x==y);
    assert((w+z)\%2=1);
    z+=x+y+w;
    y++;
    x+=z\%2;
    w+=2;
}
```
Fix VC2 using abduction:

\[ \phi_2 : (w + z) \% 2 = 1 \]

Now use circular compositional reasoning

Subgoal 1: Prove \( x = y \) using \( (w + z) \% 2 = 1 \)

Subgoal 2: Prove \( \phi_2 \) assuming \( x = y \)

Subgoal 1 is immediately discharged; focus on subgoal 2

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assume(x==y);
  assert((w+z)%2=1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```
Example, cont.

```
int x=0, y=0, w=0;
while(*) {
    assume(x==y);
    assert((w+z)%2==1);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

- Invoke client analyses to discharge proof subgoal
Example, cont.

Invoke client analyses to discharge proof subgoal

No client can prove it because initial value of $z$ unconstrained
Go back to lemma inference and annotate program with unknown precondition.
Example, cont.

```c
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(ϕ₁);
while(*) {
    assume(x==y);
    assert((w+z)%2==1);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```

- Go back to lemma inference and annotate program with unknown precondition
- Generate VC and solve for unknown $ϕ₁$:

$$ϕ₁ : z \equiv 1 \mod 2 = 1$$
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(\(\phi_1\));
while(*) {
    assume(x==y);
    assert(((w+z)%2==1);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}

- Go back to lemma inference and annotate program with unknown precondition
- Generate VC and solve for unknown \(\phi_1\): 
  \[ \phi_1 : z\%2 = 1 \]
- Now, \(\phi_1\) becomes a lemma (assertion) to be proven
Now, annotate first fragment with assertion and invoke clients
Now, annotate first fragment with assertion and invoke clients

Can be shown by any client analysis that can establish

\[ i = 3j + 1 \]
Example, cont.

Now, add this as assumption to second fragment

```c
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
    assume(x==y);
    assert((w+z)%2==1);
    z+=x+y+w;
    y++;
    x+=z%2;
    w+=2;
}
```
Now, add this as assumption to second fragment

Again, invoke client analyses to verify second fragment
Now, add this as assumption to second fragment

Again, invoke client analyses to verify second fragment – can be proven using linear congruences
Now, add this as assumption to second fragment

Again, invoke client analyses to verify second fragment – can be proven using linear congruences

**We have now proven the original assertion!**
Approach involves two key ingredients: **assertion elimination** and **assertion introduction**
Approach involves two key ingredients: **assertion elimination** and **assertion introduction**

Assertions introduced using abductive inference
Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**

- Assertions introduced using abductive inference

- Assertions eliminated using client analyses and circular compositional reasoning
Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**

- Assertions introduced using abductive inference

- Assertions eliminated using client analyses and circular compositional reasoning

- Similar to SMT solver – core part performs VC gen + abduction
Essence of Technique

- Approach involves two key ingredients: **assertion elimination** and **assertion introduction**

- Assertions introduced using abductive inference

- Assertions eliminated using client analyses and circular compositional reasoning

- Similar to SMT solver – core part performs VC gen + abduction

- Client analyses similar to theory solvers
Experiments

- Used this technique to verify safety properties in C programs
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (s)</th>
<th># queries</th>
<th>Avg # vars in query</th>
<th>Avg LOC in query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wizardpen Linux Driver</td>
<td>1242</td>
<td>3.8</td>
<td>5</td>
<td>1.5</td>
<td>29</td>
</tr>
<tr>
<td>OpenSSH clientloop</td>
<td>1987</td>
<td>2.8</td>
<td>3</td>
<td>2.3</td>
<td>5</td>
</tr>
<tr>
<td>Coreutils su</td>
<td>1057</td>
<td>3.0</td>
<td>5</td>
<td>1.7</td>
<td>6</td>
</tr>
<tr>
<td>GSL Histogram</td>
<td>526</td>
<td>0.6</td>
<td>4</td>
<td>3.6</td>
<td>15</td>
</tr>
<tr>
<td>GSL Matrix</td>
<td>7233</td>
<td>16.9</td>
<td>8</td>
<td>1.8</td>
<td>7</td>
</tr>
</tbody>
</table>
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (s)</th>
<th># queries</th>
<th>Avg # vars in query</th>
<th>Avg LOC in query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wizardpen Linux Driver</td>
<td>1242</td>
<td>3.8</td>
<td>5</td>
<td>1.5</td>
<td>29</td>
</tr>
<tr>
<td>OpenSSH clientloop</td>
<td>1987</td>
<td>2.8</td>
<td>3</td>
<td>2.3</td>
<td>5</td>
</tr>
<tr>
<td>Coreutils su</td>
<td>1057</td>
<td>3.0</td>
<td>5</td>
<td>1.7</td>
<td>6</td>
</tr>
<tr>
<td>GSL Histogram</td>
<td>526</td>
<td>0.6</td>
<td>4</td>
<td>3.6</td>
<td>15</td>
</tr>
<tr>
<td>GSL Matrix</td>
<td>7233</td>
<td>16.9</td>
<td>8</td>
<td>1.8</td>
<td>7</td>
</tr>
</tbody>
</table>

Property can be proven using our technique, but not using individual clients
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (s)</th>
<th># queries</th>
<th>Avg # vars in query</th>
<th>Avg LOC in query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wizardpen Linux Driver</td>
<td>1242</td>
<td>3.8</td>
<td>5</td>
<td>1.5</td>
<td>29</td>
</tr>
<tr>
<td>OpenSSH clientloop</td>
<td>1987</td>
<td>2.8</td>
<td>3</td>
<td>2.3</td>
<td>5</td>
</tr>
<tr>
<td>Coreutils su</td>
<td>1057</td>
<td>3.0</td>
<td>5</td>
<td>1.7</td>
<td>6</td>
</tr>
<tr>
<td>GSL Histogram</td>
<td>526</td>
<td>0.6</td>
<td>4</td>
<td>3.6</td>
<td>15</td>
</tr>
<tr>
<td>GSL Matrix</td>
<td>7233</td>
<td>16.9</td>
<td>8</td>
<td>1.8</td>
<td>7</td>
</tr>
</tbody>
</table>

Verification time reasonable (0.6-16.9s)
Experiments

- Used this technique to verify safety properties in C programs
- Used four different static analysis tools as clients

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Time (s)</th>
<th># queries</th>
<th>Avg # vars in query</th>
<th>Avg LOC in query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wizardpen Linux Driver</td>
<td>1242</td>
<td>3.8</td>
<td>5</td>
<td>1.5</td>
<td>29</td>
</tr>
<tr>
<td>OpenSSH clientloop</td>
<td>1987</td>
<td>2.8</td>
<td>3</td>
<td>2.3</td>
<td>5</td>
</tr>
<tr>
<td>Coreutils su</td>
<td>1057</td>
<td>3.0</td>
<td>5</td>
<td>1.7</td>
<td>6</td>
</tr>
<tr>
<td>GSL Histogram</td>
<td>526</td>
<td>0.6</td>
<td>4</td>
<td>3.6</td>
<td>15</td>
</tr>
<tr>
<td>GSL Matrix</td>
<td>7233</td>
<td>16.9</td>
<td>8</td>
<td>1.8</td>
<td>7</td>
</tr>
</tbody>
</table>

Fragments extracted for queries small in practice
In program sketching, programmer writes a draft program with “holes”
In program sketching, programmer writes a draft program with “holes”

Program synthesizer completes the holes in a way that satisfies specification
In **program sketching**, programmer writes a draft program with “holes”

Program synthesizer completes the holes in a way that satisfies specification

Abduction is useful for synthesizing unknown guards in program sketches
Programmers often write checks to prevent memory safety errors (buffer overruns, null dereferences, ...)

```java
if(C) {R} else { /* handle error */}
```
Programmers often write checks to prevent memory safety errors (buffer overruns, null dereferences, ...)

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

Such checks are tedious to write and error-prone (e.g., off-by-one errors common cause of buffer overflows)
Key Idea: Program synthesis to guarantee memory safety

if(???) {R} else { /* handle error */}
**Key Idea:** Program synthesis to guarantee memory safety

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

Programmer specifies *which* parts of the program should be protected and how to handle error.
Key Idea: Program synthesis to guarantee memory safety

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

1. Programmer specifies which parts of the program should be protected and how to handle error.

2. Technique synthesizes guards that guarantee memory safety.
   - Guards should be as permissive and concise as possible.
   - Key ingredient of synthesis algorithm is abduction.
**Solution Overview**

1. **Constraint Generation:**

   \[ F_1(x_1, \ldots x_{k-1}) \land x_k \]

   \[ \Rightarrow \]

   \[ F_2(x_{k+1}, \ldots x_n) \]

2. **Constraint Solving:**

   \[ x_1 = \phi_1, \ldots, x_n = \phi_n \]
**Constraint Generation:**

- Represent unknown guards using placeholders

\[ F_1(x_1, \ldots, x_{k-1}) \land x_k \Rightarrow F_2(x_{k+1}, \ldots, x_n) \]

\[ x_1 = \phi_1, \ldots, x_n = \phi_n \]
Solution Overview

**Constraint Generation:**
- Represent unknown guards using placeholders
- Perform dual forward and backward analysis to generate constraint for each unknown

\[ F_1(x_1, \ldots, x_{k-1}) \land x_k \Rightarrow F_2(x_{k+1}, \ldots, x_n) \]

\[ x_1 = \phi_1, \ldots, x_n = \phi_n \]
Solution Overview

1. **Constraint Generation:**
   - Represent unknown guards using placeholders
   - Perform dual forward and backward analysis to generate constraint for each unknown

2. **Constraint Solving:**

$$F_1(x_1, \ldots, x_{k-1}) \land x_k \Rightarrow \ F_2(x_{k+1}, \ldots, x_n)$$

$$x_1 = \phi_1, \ldots, x_n = \phi_n$$
Constraint Generation:

- Represent unknown guards using placeholders
- Perform dual forward and backward analysis to generate constraint for each unknown

Constraint Solving:

- An extended abduction algorithm for solving constraint system with multiple unknowns
Generate one constraint per unknown

\[ \phi \]

\{ ... \}

\}

if(??)
{
   ...
}

\[ \psi \]

\{ ...

\}

To guarantee memory safety, find ?? such that \( \phi \land ?? \models \psi \)
In More Detail

- Generate one constraint per unknown
- Compute postcondition $\phi$ of code before unknown

$$\phi$$

```java
{ ...
}
```

$$\psi$$

```java
if(??)
{
{ ...
}
```
In More Detail

- Generate one constraint per unknown
- Compute postcondition $\phi$ of code before unknown
- Compute safety precondition $\psi$ of code nested inside unknown

To guarantee memory safety, find $??$ such that $\phi \land ?? \vdash \psi$

This is almost an abduction problem, but $\phi, \psi$ can have other unknowns

Impose ordering on constraints and reduce to standard abduction
In More Detail

- Generate one constraint per unknown
- Compute postcondition $\phi$ of code before unknown
- Compute safety precondition $\psi$ of code nested inside unknown
- To guarantee memory safety, find ?? such that $\phi \land ?? \models \psi$
Generate one constraint per unknown

Compute postcondition $\phi$ of code before unknown

Compute safety precondition $\psi$ of code nested inside unknown

To guarantee memory safety, find ?? such that $\phi \land ?? \models \psi$

This is almost an abduction problem, but $\phi, \psi$ can have other unknowns
- Generate one constraint per unknown
- Compute postcondition $\phi$ of code before unknown
- Compute safety precondition $\psi$ of code nested inside unknown
- To guarantee memory safety, find $??$ such that $\phi \land ?? \models \psi$
- This is almost an abduction problem, but $\phi, \psi$ can have other unknowns
- Impose ordering on constraints and reduce to standard abduction
Example

- Code snippet from Unix Coreutils with protected memory access

```c
int main(int argc, 
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) { 
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Example

- Code snippet from Unix Coreutils with protected memory access

  **Convention:** For pointer \( p \):

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;  
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Example

- Code snippet from Unix Coreutils with protected memory access
- **Convention**: For pointer $p$:
  - $p^+$ represents distance to end of memory block
  - $p^-$ represents distance from beginning of memory block

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...)
  {
    optind++;
    if(*) {
      argv++;
      argc--;
    }
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Example

- Code snippet from Unix Coreutils with protected memory access

- **Convention:** For pointer $p$
  - $p^+$ represents distance to end of memory block
  - $p^-$ represents distance from beginning of memory block

```c
int main(int argc, char** argv) {
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {
      argv++;
      argc--;
    }
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
**First Step:** Compute what is known at ?? ⇒ *postcondition* $\phi$

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++; argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
First Step: Compute what is known at ?? \(\Rightarrow\) postcondition \(\phi\)

- From language semantics:

\[\text{argv}^+ = \text{argc} \land \text{argv}^- = 0\]

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;  
  optind=0;
  while(...) { 
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
First Step: Compute what is known at ?? ⇒ postcondition $\phi$

- From language semantics:
  
  $$argv^+ = argc \land argv^- = 0$$

- From computing the strongest postcondition:
  
  $$argv^+ = argc \land argv^- \geq 1 \land optind \geq 0$$

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Second Step: Compute what needs to hold at ?? to ensure memory safety
\[ \Rightarrow \text{precondition } \psi \]
Example Cont.

- **Second Step:** Compute what needs to hold at $??$ to ensure memory safety
  $\Rightarrow$ **precondition** $\psi$

- Buffer access:

  $$\mathit{optind} + 1 < \mathit{argv}^+ \land \mathit{optind} + 1 \geq -\mathit{argv}^-$$

```c
int main(int argc,
         char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
            argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Solve abduction problem

\[ \phi \land \ldots \models \psi \quad \text{where} \]

\[ \phi : \quad \begin{align*}
argv^+ & = \text{argc} \land \\
argv^- & \geq 1 \land \text{optind} \geq 0
\end{align*} \]

\[ \psi : \quad \begin{align*}
\text{optind} + 1 & < argv^+ \land \\
\text{optind} + 1 & \geq -argv^-
\end{align*} \]

```c
int main(int argc, char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++; argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Solve abduction problem
\[ \phi \land \psi \models \psi \] where
\[ \phi : \]
\[ argv^+ = argc \land \]
\[ argv^- \geq 1 \land optind \geq 0 \]
\[ \psi : \]
\[ optind + 1 < argv^+ \land \]
\[ optind + 1 \geq -argv^- \]

Solution: \[ argc - optind > 1 \]

```c
int main(int argc, 
    char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;
  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```
Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH
Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH
- Removed conditionals used to prevent memory safety errors
Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH
- Removed conditionals used to prevent memory safety errors
- Used our new technique to synthesize the missing guards
Experiments Cont.

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

Used technique to synthesize 27 unknown guards in real C programs
Experiments Cont.

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

In 21 out of 27 cases, tool inferred same predicate as programmer
Experiments Cont.

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

In 4 cases, syntactically different, but semantically equivalent guards
Experiments Cont.

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

In 2 cases, guards did not match
⇒ bug in original program!
Abduction = logical formulation of “guessing”
Abduction = logical formulation of “guessing”

Lots of uses in automated reasoning about programs, particularly when combined with backtracking search

\[ \Gamma \wedge ? \models \phi \]
Abduction = logical formulation of “guessing”

Lots of uses in automated reasoning about programs, particularly when combined with backtracking search

If you are interested in using abduction, check out:

http://www.cs.utexas.edu/~tdillig/mistral/explain.html
Abduction = logical formulation of “guessing”

Lots of uses in automated reasoning about programs, particularly when combined with backtracking search

If you are interested in using abduction, check out:

http://www.cs.utexas.edu/~tdillig/mistral/explain.html

Easy to use: `expl = conclusion.abduce(premises);`
Limitations and Future Work

⚠️ Abduction algorithm uses quantifier elimination ⇒ limited scalability and requires to theories that admit QE

Future work:
- Alternative algorithms that don’t use QE
- Multi-abduction algorithm to simultaneously infer multiple unknowns
Limitations and Future Work

⚠️ Abduction algorithm uses quantifier elimination ⇒ limited scalability and requires to theories that admit QE

- **Future work**: Alternative algorithms that don’t use QE
Limitations and Future Work

⚠️ Abduction algorithm uses quantifier elimination ⇒ limited scalability and requires theories that admit QE

- **Future work**: Alternative algorithms that don’t use QE

⚠️ Abduction requires single unknown in LHS, but sometimes there are multiple unknowns
Limitations and Future Work

⚠ Abduction algorithm uses quantifier elimination $\Rightarrow$ limited scalability and requires to theories that admit QE

- **Future work:** Alternative algorithms that don’t use QE

⚠ Abduction requires single unknown in LHS, but sometimes there are multiple unknowns

- **On-going work:** Multi-abduction algorithm to simultaneously infer multiple unknowns
Questions?

Thank you!