
# Optimal Guard Synthesis for Memory Safety

**Tom Dillig**, Isil Dillig

THE UNIVERSITY OF TEXAS AT AUSTIN

Swarat Chaudhuri

# Memory Safety Errors

- Memory safety errors cause many program errors

# Memory Safety Errors

- Memory safety errors cause many program errors
- In C and C++ perennial source of security vulnerabilities

# Memory Safety Errors

- Memory safety errors cause many program errors
- In C and C++ perennial source of security vulnerabilities
- In Java, C# program crashes due to exceptions

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors
  - \- Does not help programmer write safe-by-construction code

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors
  - \- Does not help programmer write safe-by-construction code
- Dynamic approaches to memory safety

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors
  - \- Does not help programmer write safe-by-construction code
- Dynamic approaches to memory safety
  - \+ Widely used in managed languages

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors
  - \- Does not help programmer write safe-by-construction code
- Dynamic approaches to memory safety
  - \+ Widely used in managed languages
  - \- But only transforms vulnerability into program crash

# Preventing Memory Safety Errors

- Many static techniques proposed to detect memory errors
  - \+ Can guarantee absence of errors
  - \- Can be hard to understand and fix detected errors
  - \- Does not help programmer write safe-by-construction code
- Dynamic approaches to memory safety
  - \+ Widely used in managed languages
  - \- But only transforms vulnerability into program crash
  - \- Run-time overhead

# Key Idea: Use Program Synthesis

**Key Idea: Program synthesis to guarantee memory safety**

# Key Idea: Use Program Synthesis

**Key Idea: Program synthesis to guarantee memory safety**

1. Programmer specifies which parts of the program should be guarded

Example:

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

# Key Idea: Use Program Synthesis

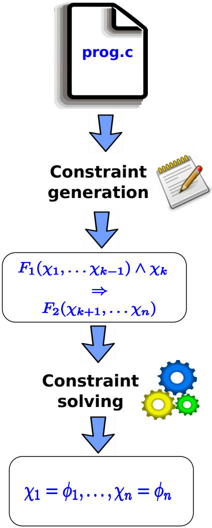
**Key Idea: Program synthesis to guarantee memory safety**

1. Programmer specifies which parts of the program should be guarded
2. Our technique synthesizes correct and optimal guards that guarantee memory safety
   - Optimal means as weak and as simple as possible

Example:

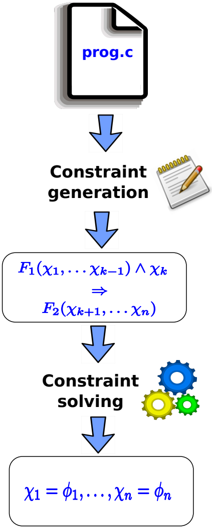
```
if(???)  {R} else { /* handle error */}
```

# Solution Overview



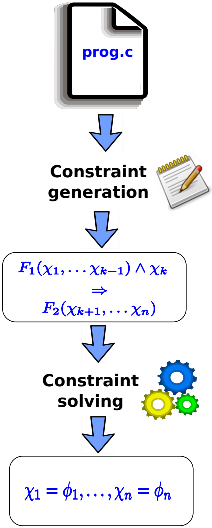

1. Constraint Generation:

# Solution Overview



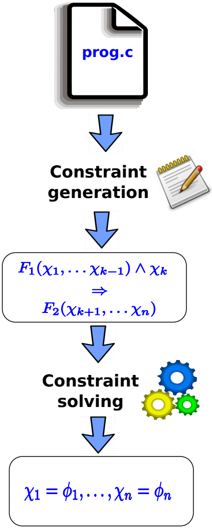

1. Constraint Generation:
    - Represent unknown guards using placeholders

# Solution Overview



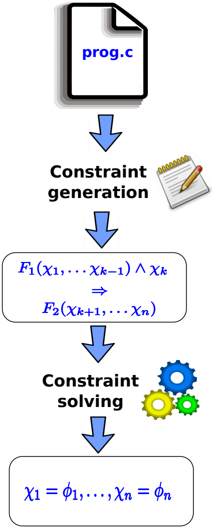

1. Constraint Generation:
    - Represent unknown guards using placeholders
    - Perform dual forward and backward analysis to generate constraint for each unknown

# Solution Overview



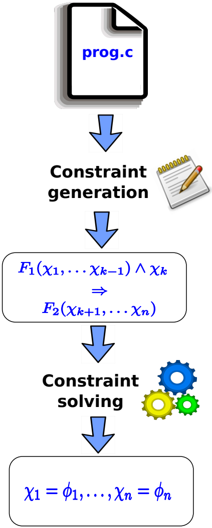

1. Constraint Generation:
   - Represent unknown guards using placeholders
   - Perform dual forward and backward analysis to generate constraint for each unknown
2. Constraint Solving:

# 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:
   - An extended abduction algorithm for solving constraint system with multiple unknowns

# 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:
   - An extended abduction algorithm for solving constraint system with multiple unknowns
   - Guarantees Pareto-optimality

# Constraint Generation Overview

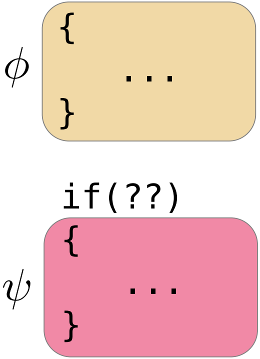
$\phi$

```
{
    ...
}

if(??)
{
    ...
}
```

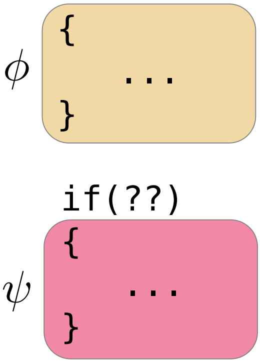
- At synthesis point, compute postcondition $\phi$ of code above ??

# Constraint Generation Overview




- At synthesis point, compute postcondition $\phi$ of code above ??
- Compute precondition $\psi$ that ensures memory safety of code guarded by ??

# Constraint Generation Overview




- At synthesis point, compute postcondition $\phi$ of code above ??
- Compute precondition $\psi$ that ensures memory safety of code guarded by ??
- Condition to guarantee memory safery:

$$\phi \wedge ?? \models \psi$$

# Key Insight: Abduction

**Solution: Abductive inference**

# Key Insight: Abduction

**Solution: Abductive inference**

- Given facts $F$ and desired outcome $O$, find simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \text{SAT}(F \wedge E)$$

# Key Insight: Abduction

**Solution: Abductive inference**

- Given facts $F$ and desired outcome $O$, find simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \text{SAT}(F \wedge E)$$

- $F \equiv$ postcondition $\phi$ before ??

# Key Insight: Abduction

**Solution: Abductive inference**

- Given facts $F$ and desired outcome $O$, find simple explanatory hypothesis $E$ such that

$$F \land E \models O \quad \text{and} \quad \text{SAT}(F \land E)$$

- $F \equiv$ postcondition $\phi$ before ??
- $O \equiv$ memory safety precondition $\psi$

# Key Insight: Abduction

**Solution: Abductive inference**

- Given facts $F$ and desired outcome $O$, find simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \mathrm{SAT}(F \wedge E)$$

- $F \equiv$ postcondition $\phi$ before ??
- $O \equiv$ memory safety precondition $\psi$
- $E \equiv$ Solution for ??

# Constraint Solving

$$F_1(\vec{\chi_1}) \wedge \chi_1 \Rightarrow F_1'(\vec{\chi_1'})$$
$$F_2(\vec{\chi_2}) \wedge \chi_2 \Rightarrow F_2'(\vec{\chi_2'})$$
$$\vdots$$
$$F_n(\vec{\chi_n}) \wedge \chi_n \Rightarrow F_n'(\vec{\chi_n'})$$

- Cannot directly use abduction because constraints have multiple unknowns

# Constraint Solving

$$F_1(\vec{\chi_1}) \wedge \chi_1 \Rightarrow F'_1(\vec{\chi'_1})$$
$$F_2(\vec{\chi_2}) \wedge \chi_2 \Rightarrow F'_2(\vec{\chi'_2})$$
$$\vdots$$
$$F_n(\vec{\chi_n}) \wedge \chi_n \Rightarrow F'_n(\vec{\chi'_n})$$

- Cannot directly use abduction because constraints have multiple unknowns
- New iterative, stratification-based algorithm for solving constraint system

# Constraint Solving

$$F_1(\vec{\chi_1}) \wedge \chi_1 \Rightarrow F_1'(\vec{\chi_1'})$$
$$F_2(\vec{\chi_2}) \wedge \chi_2 \Rightarrow F_2'(\vec{\chi_2'})$$
$$\vdots$$
$$F_n(\vec{\chi_n}) \wedge \chi_n \Rightarrow F_n'(\vec{\chi_n'})$$

- Cannot directly use abduction because constraints have multiple unknowns
- New iterative, stratification-based algorithm for solving constraint system
- Uses abduction as a helper procedure

# Constraint Solving

$$F_1(\vec{\chi_1}) \land \chi_1 \Rightarrow F_1'(\vec{\chi_1'})$$
$$F_2(\vec{\chi_2}) \land \chi_2 \Rightarrow F_2'(\vec{\chi_2'})$$
$$\vdots$$
$$F_n(\vec{\chi_n}) \land \chi_n \Rightarrow F_n'(\vec{\chi_n'})$$

- Cannot directly use abduction because constraints have multiple unknowns
- New iterative, stratification-based algorithm for solving constraint system
- Uses abduction as a helper procedure
- Resulting solution is Pareto-optimal

# Constraint Solving

$$F_1(\vec{\chi_1}) \wedge \chi_1 \Rightarrow F_1'(\vec{\chi_1'})$$
$$F_2(\vec{\chi_2}) \wedge \chi_2 \Rightarrow F_2'(\vec{\chi_2'})$$
$$\vdots$$
$$F_n(\vec{\chi_n}) \wedge \chi_n \Rightarrow F_n'(\vec{\chi_n'})$$

- Cannot directly use abduction because constraints have multiple unknowns
- New iterative, stratification-based algorithm for solving constraint system
- Uses abduction as a helper procedure
- Resulting solution is Pareto-optimal
  - Cannot improve solution for one unknown without making others worse

# Example

- Code snippet from Unix Coreutils with protected memory access

```
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

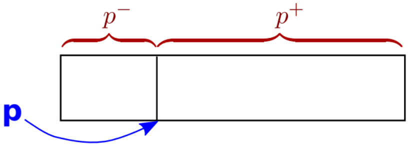
- Code snippet from Unix Coreutils with protected memory access
- Convention: For pointer $p$:

```
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

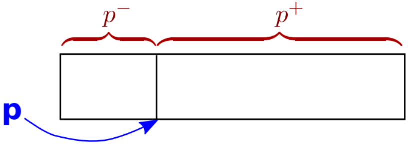



```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  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




```
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 Cont.

- First Step: Compute what is known at ?? $\Rightarrow$ postcondition $\phi$

```
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 Cont.

- First Step: Compute what is known at ?? ⇒ postcondition $\phi$
  - From language semantics:

$$argv^{+} = argc \wedge argv^{-} = 0$$

```
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 Cont.

- First Step: Compute what is known at ?? $\Rightarrow$ postcondition $\phi$
  - From language semantics:

    $$argv^{+} = argc \wedge argv^{-} = 0$$

  - From computing the strongest postcondition:

    $$argv^{+} = argc \wedge$$
    $$argv^{-} \geq 1 \wedge optind \geq 0$$

```
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 Cont.

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

```
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 Cont.

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

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

```
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 Cont.

- Solve abduction problem
  $\phi \wedge ?? \models \psi$ where

$$\phi : \begin{array}{c} argv^{+} = argc \wedge \\ argv^{-} \geq 1 \;\wedge\; optind \geq 0 \end{array}$$

$$\psi : \begin{array}{l} optind + 1 < argv^{+} \wedge \\ optind + 1 \geq -argv^{-} \end{array}$$

```
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 Cont.

- Solve abduction problem
  $\phi \wedge ?? \models \psi$ where

$$\phi : \begin{array}{c} argv^{+} = argc \wedge \\ argv^{-} \geq 1 \wedge optind \geq 0 \end{array}$$

$$\psi : \begin{array}{l} optind + 1 < argv^{+} \wedge \\ optind + 1 \geq -argv^{-} \end{array}$$

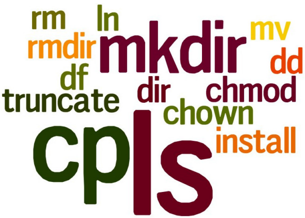
- Solution: $argc - optind > 1$

```
int main(int argc,
   char** argv)
{
  if(argc<=1) return -1;
  argv++; argc--;

  optind=0;
  while(...) {
    optind++;
    if(*) {argv++;
           argc--;}
  }
  if(??) {
    argv[optind+1]=...;
  }
}
```

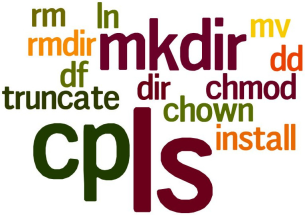
# Experiments

- Evaluated technique on the Unix Coreutils and parts of OpenSSH



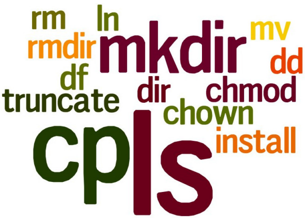
# Experiments

- 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.

| 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 |

# Summary

- New synthesis-based approach for writing memory safe programs

# Summary

- New synthesis-based approach for writing memory safe programs
- Two key ingredients:

# Summary

- New synthesis-based approach for writing memory safe programs
- Two key ingredients:
  - Constraint generation: Generates VCs with placeholders using dual forward and backward reasoning

# Summary

- New synthesis-based approach for writing memory safe programs
- Two key ingredients:
  - Constraint generation: Generates VCs with placeholders using dual forward and backward reasoning
  - Constraint solving: New abduction-based algorithm for finding optimal solutions for placeholders representing unknown guards

# Summary

- New synthesis-based approach for writing memory safe programs
- Two key ingredients:
  - Constraint generation: Generates VCs with placeholders using dual forward and backward reasoning
  - Constraint solving: New abduction-based algorithm for finding optimal solutions for placeholders representing unknown guards
- Experimental validation of our approach

# Questions?