Cartesian Hoare Logic for Verifying *k*-Safety Properties

Marcelo Sousa
University of Oxford

Işıl Dillig
UT Austin

June 15
PLDI 2016

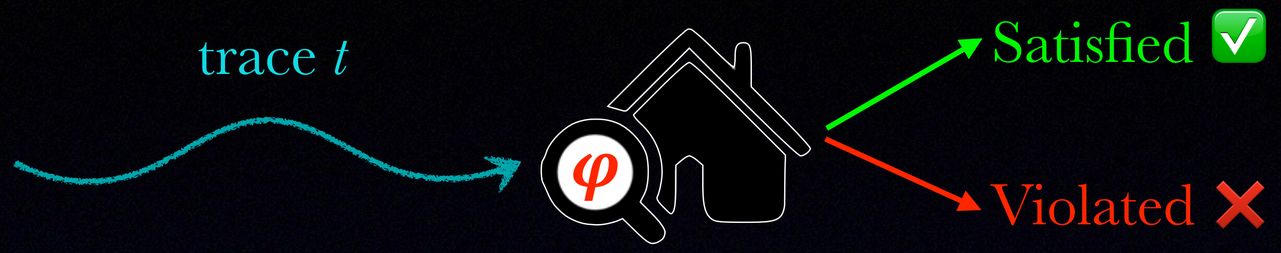
Safety vs k-Safety

Safety properties stipulate that nothing bad happens in one run of the program

Safety vs *k*-Safety

Safety properties stipulate that nothing bad happens in one run of the program

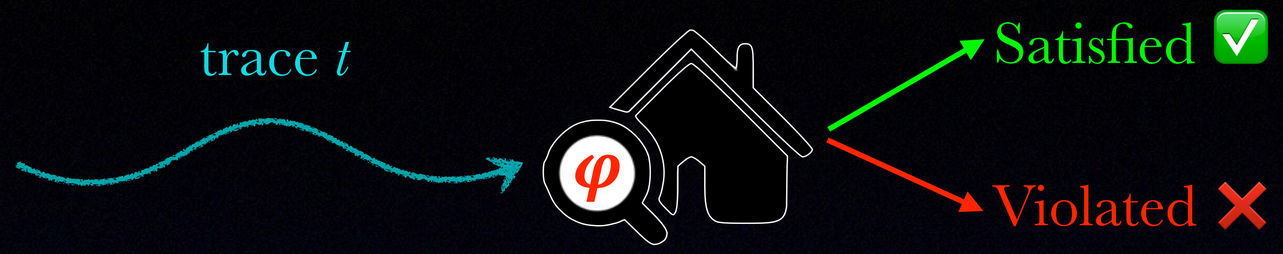
Counterexample:
single trace



Safety vs *k*-Safety

Safety properties stipulate that nothing bad happens in one run of the program

Counterexample:
single trace

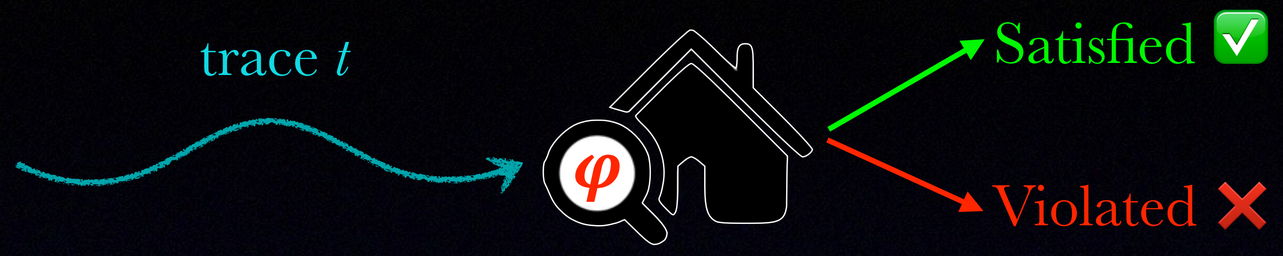


k-safety properties stipulate there is no bad interaction between *k* traces

Safety vs *k*-Safety

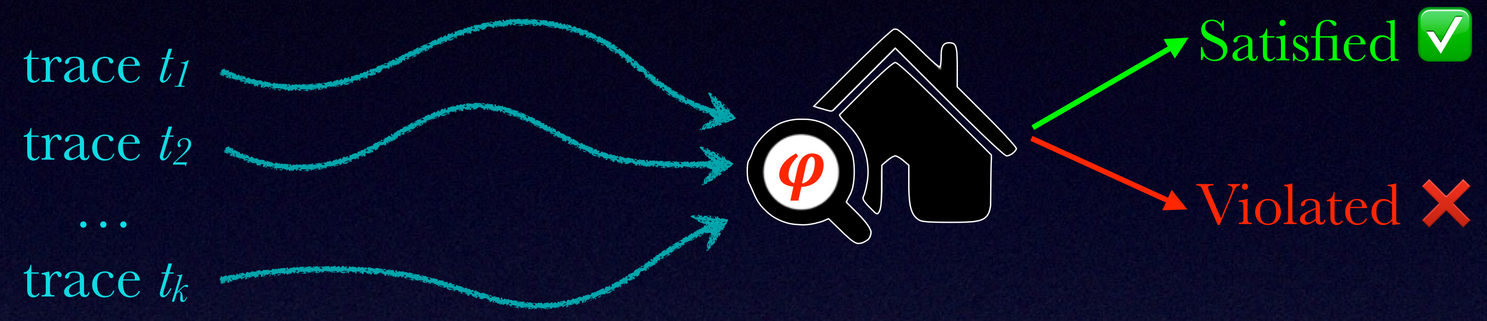
Safety properties stipulate that nothing bad happens in one run of the program

Counterexample:
single trace



k-safety properties stipulate there is no bad interaction between *k* traces

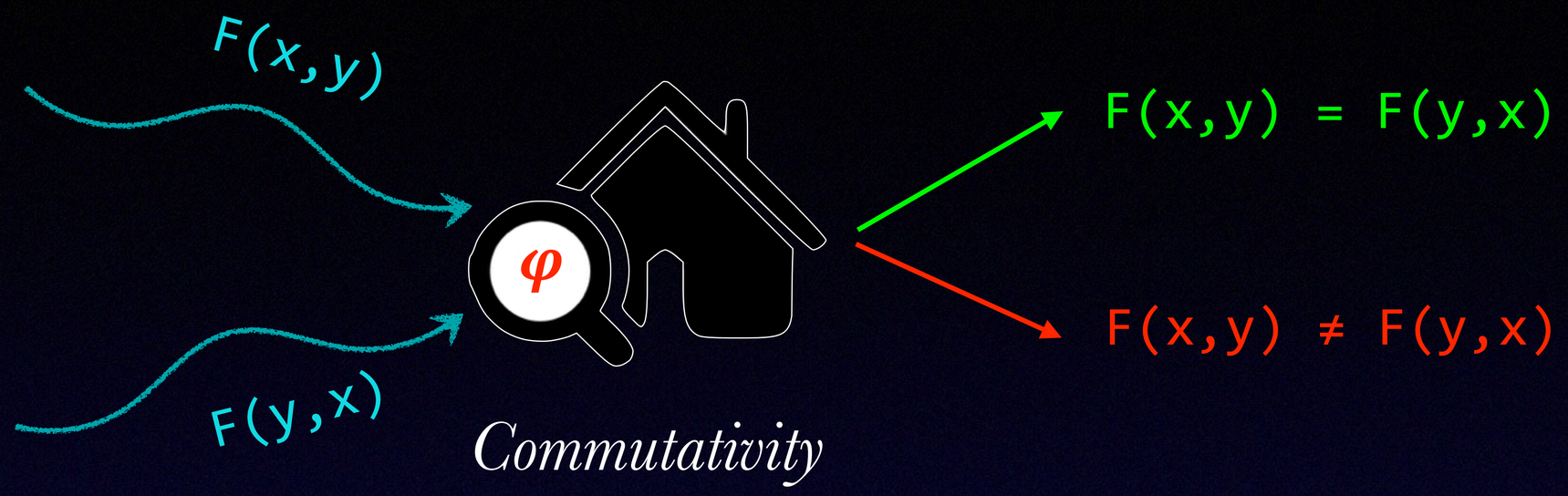
Counterexample:
k traces



Examples of k-safety

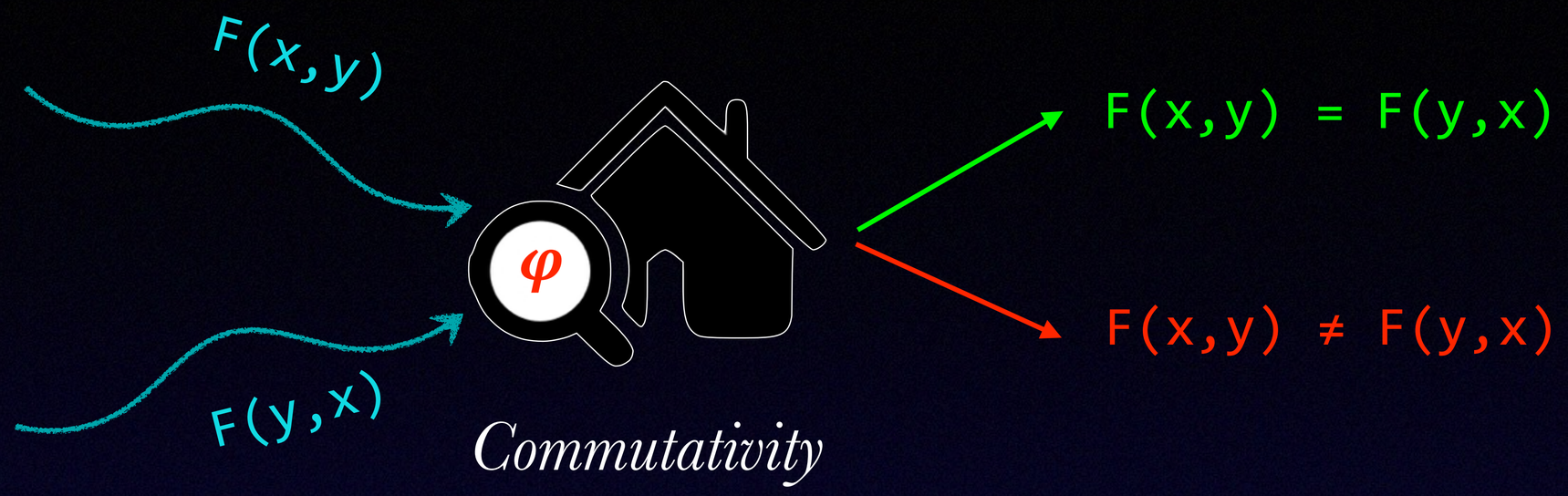
Examples of k-safety

2-safety



Examples of *k*-safety

2-safety



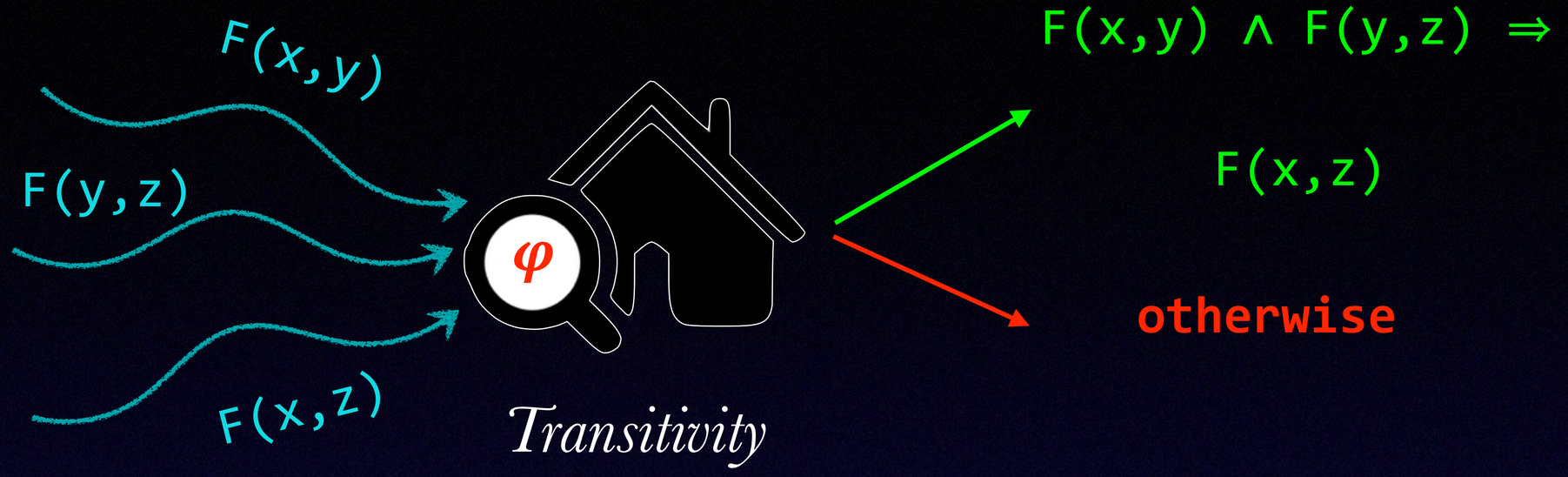
Determinism, Symmetry, Anti-symmetry, Non-interference, …

Examples of k-safety

3-safety

Examples of *k*-safety

3-safety

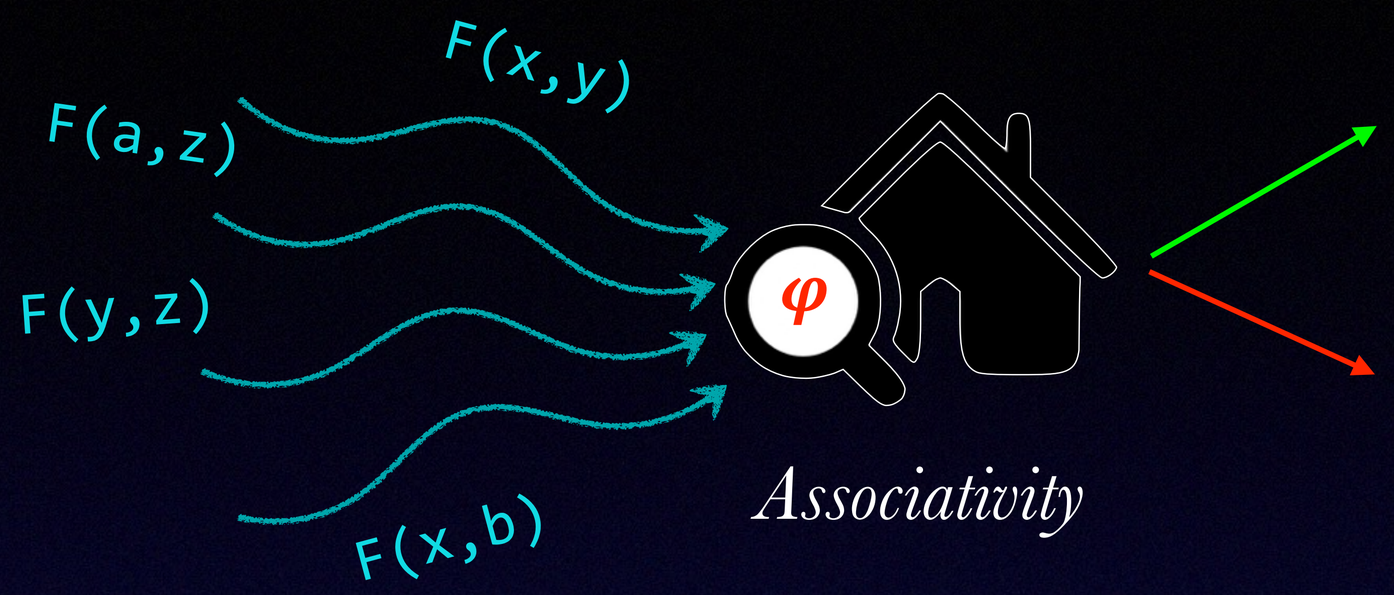


Examples of k-safety

4-safety ?

Examples of *k*-safety

4-safety ?



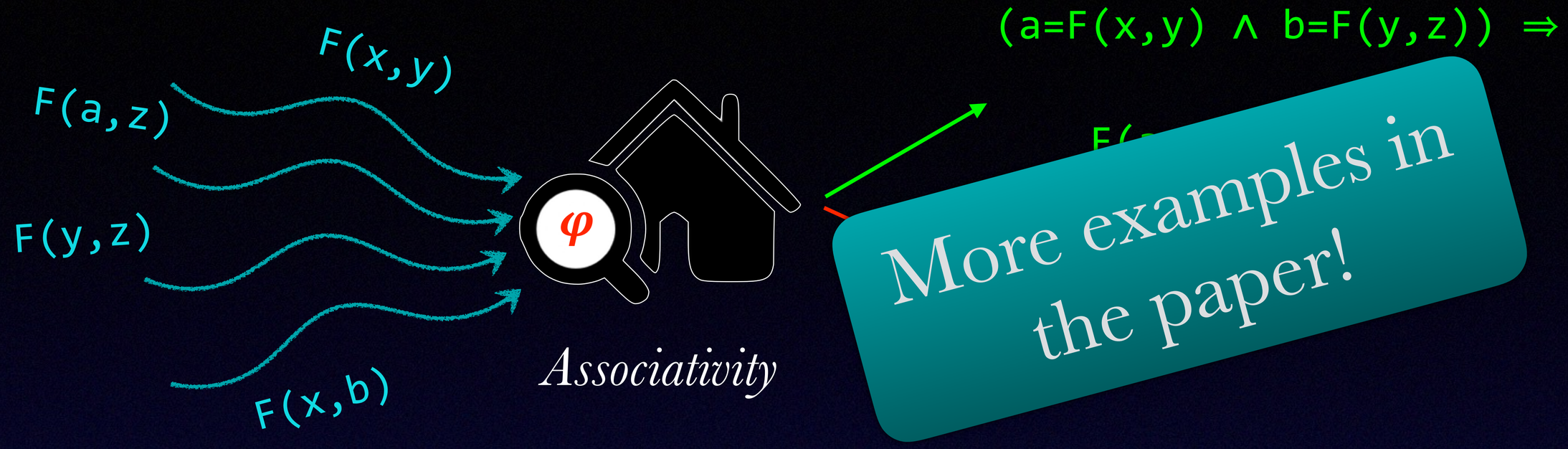
(a=F(x,y) ∧ b=F(y,z)) ⇒

F(a,z) = F(x,b)

otherwise

Examples of *k*-safety

4-safety ?



Goal

Technique for automatically verifying k-safety properties

Goal

Technique for automatically verifying *k*-safety properties

Important problem because *k*-safety is a key part of functional correctness

Goal

Technique for automatically verifying *k*-safety properties

Important problem because *k*-safety is a key part of functional correctness

Example 1. Comparators: Need to obey anti-symmetry, transitivity, totality

Example 2. Reducers: Need to obey commutativity and associativity

Comparator Example

Comparator Example

```
public int compare(String c1, String c2) {
    if (c1.indexOf(’4’) != -1 || c2.indexOf(’4’) != -1) {  // Four of a kind
        if (c1.indexOf(’4’) == c2.indexOf(’4’)) {
            for (int i = 12; i >= 0; i--) {
                if (c1.charAt(i) != ’0’ && c1.charAt(i) != ’4’) {
                    if (c2.charAt(i) != ’0’ && c2.charAt(i) != ’4’) return 0;
                    return 1;  }
                if (c2.charAt(i) != ’0’ && c2.charAt(i) != ’4’) return -1; } }
        return c1.indexOf(’4’) - c2.indexOf(’4’); }
    int tripleCount1 = StringFunctions.countOccurrencesOf(c1, "3");
    int tripleCount2 = StringFunctions.countOccurrencesOf(c2, "3");
    if (tripleCount1 > 1 || (tripleCount1 == 1 && c1.indexOf(’2’) != -1) ||
       tripleCount2 > 1 || (tripleCount2 == 1 && c2.indexOf(’2’) != -1)) { //Full house
        int higherTriple = c1.lastIndexOf(’3’);
        if (higherTriple == c2.lastIndexOf(’3’)) {
            for (int i = 12; i >= 0; i--) {
                if (i == higherTriple) continue;
                if (c1.charAt(i) == ’2’ || c1.charAt(i) == ’3’) {
                    if (c2.charAt(i) == ’2’ || c2.charAt(i) == ’3’) return 0;
                    return 1;  }
                if (c2.charAt(i) == ’2’ || c2.charAt(i) == ’3’) return -1; }  }
        return higherTriple - c2.lastIndexOf(’3’); }
    return 0; }
```

Comparator Example

```
public int compare(String c1, String c2) {
    if (c1.indexOf('4') != -1 || c2.indexOf('4') !=
        if (c1.indexOf('4') == c2.indexOf('4')) {
            for (int i = 12; i >= 0; i--) {
                if (c1.charAt(i) != '0' && c1.charA
                    if (c2.charAt(i) != '0' && c2.
                    return 1;  }
                if (c2.charAt(i) != '0' && c2.ch
        return c1.indexOf('4') - c2.indexOf('4');
    int tripleCount1 = StringFunctions.countOccurr
    int tripleCount2 = StringFunctions.countOccurre
    if (tripleCount1 > 1 || (tripleCount1 == 1 && c
       tripleCount2 > 1 || (tripleCount2 == 1 && c2.
        int higherTriple = c1.lastIndexOf('3');
        if (higherTriple == c2.lastIndexOf('3')) {
            for (int i = 12; i >= 0; i--) {
                if (i == higherTriple) continue;
                if (c1.charAt(i) == '2' || c1.charAt(i) == '3') {
                    if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return 0;
                    return 1;  }
                if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return -1; }  }
        return higherTriple - c2.lastIndexOf('3'); }
    return 0; }
```

Buggy!

Violates transitivity

Non-trivial to see why…

Comparator Example

Many forum posts where programmers ask for static analysis tools for checking *k*-safety

"As many people before, I hit IllegalArgumentException: Comparison method violates its general contract! I need to check the code for all possible occurrences of this. The best way would be a static analysis tool that will go through all the code and detect possible issues."

```
('4') !- 1) {
')) {

charA
c2.

ch
');
ccurr
ccurre
1 && c
&& c2.
house
);
indexOf('3')) {
0; i--) {
if (i == higherTriple) continue;
(c1.charAt(i) == '2' || c1.charAt(i) == '3') {
if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return 0;
return 1;  }
if (c2.charAt(i) == '2' || c2.charAt(i) == '3') return -1; }  }
return higherTriple - c2.lastIndexOf('3'); }
return 0; }
```

Buggy!

Violates transitivity

Non-trivial to see why…

This talk

Technique for verifying k-safety properties for any k

This talk

Technique for verifying k-safety properties for any k

New program logic called
Cartesian Hoare Logic
(CHL)

- Sound
- Relatively Complete
- Easy to Automate

This talk

Technique for verifying k-safety properties for any k

New program logic called
Cartesian Hoare Logic
(CHL)

- Sound
- Relatively Complete
- Easy to Automate

Built a prototype verification tool called **DESCARTES** for verifying k-safety

Brief background

Prior work on verification of 2-safety properties,
particularly in the context of security protocols

Brief background

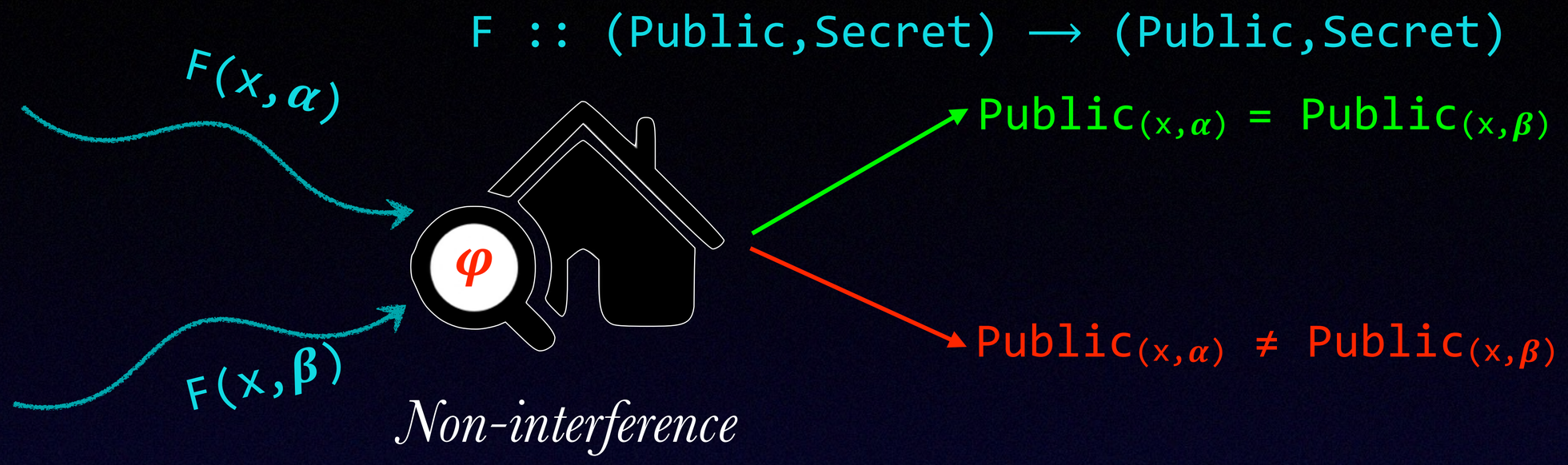
Prior work on verification of 2-safety properties,
particularly in the context of security protocols

```
F :: (Public,Secret) ⟶ (Public,Secret)
```

Non-interference

Brief background

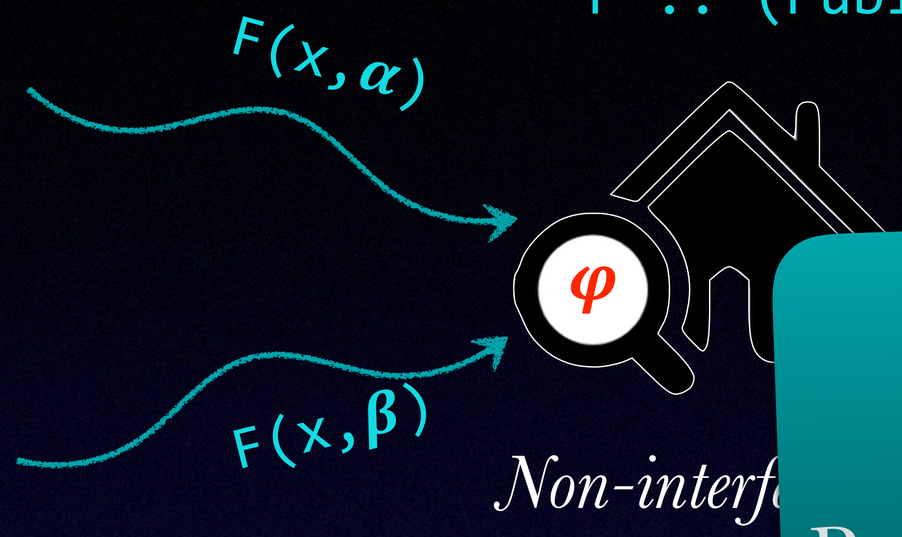
Prior work on verification of 2-safety properties,
particularly in the context of security protocols



Brief background

Prior work on verification of 2-safety properties,
particularly in the context of security protocols

```
F :: (Public,Secret) ⟶ (Public,Secret)
```



$\texttt{Public}_{(x,\alpha)} = \texttt{Public}_{(x,\beta)}$

Method

Reduction of *2*-safety to *1*-safety!

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Given a program

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Given a program and a *2*-safety property:

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Given a program and a *2*-safety property:

- Construct a new program from the original one (simplest method is called *self-composition*)

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

```
void new_func(int x, int y) {
  while (x < 10){
    x++;
  }
  ret1 = x;
  while (y < 10){
    y++;
  }
  ret2 = y;
}
```

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Given a program and a *2*-safety property:

- Construct a new program from the original one (simplest method is called *self-composition*)
- Instrument new program with assertions obtained from the property

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

```
void new_func(int x, int y) {
  assume(x == y);
  while (x < 10){
    x++;
  }
  ret1 = x;
  while (y < 10){
    y++;
  }
  ret2 = y;
  assert(ret1 == ret2);
}
```

Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

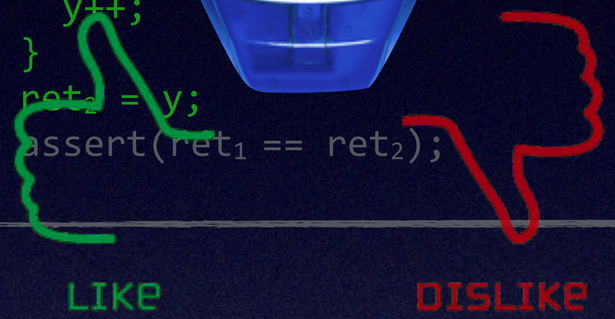
Given a program and a *2*-safety property:

- Construct a new program from the original one (simplest method is called *self-composition*)
- Instrument new program with assertions obtained from the property
- Feed new program to off the shelf verifier

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

```
void new_func(int x, int y) {
  assume(x == y);
  while
    x++;
  }
  ret1 =
  while (y
    y++;
  }
  ret2 = y;
  assert(ret1 == ret2);
}
```



Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

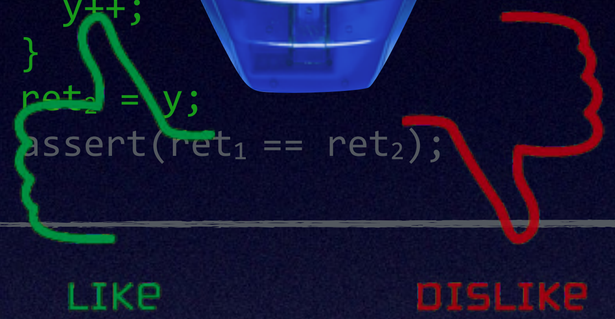
Given a program and a *2*-safety property:

- Construct a new program from the original one (simplest method is called *self-composition*)
- Instrument new program with assertions obtained from the property
- Feed new program to off the shelf verifier
- Program satisfies *2*-safety property iff new program is verified

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

```
void new_func(int x, int y) {
  assume(x == y);
  while
    x++;
  }
  ret1 =
  while (y
    y++;
  }
  ret2 = y;
  assert(ret1 == ret2);
}
```



Methodology of Prior Work

Method

From *2*-safety to *1*-safety!

Given a program and a *2*-safety property:

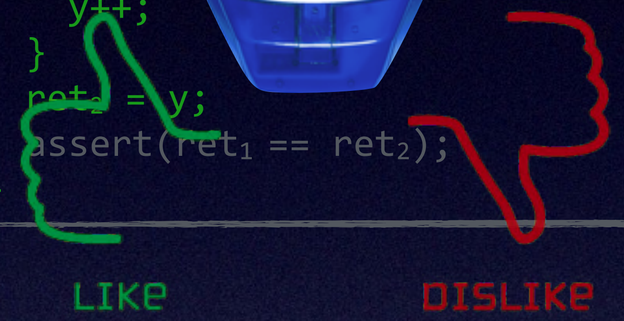
- Construct a new program from the original one (simplest method is call
osition)
- Ins
ns
- Feed
er
- Progra

new
d

2-safety property: $\forall x,y.\ x = y \Rightarrow func(x) = func(y)$

```
int func(int x) {
  while (x < 10){
    x++;
  }
  return x;
}
```

```
void new_func(int x, int y) {
  assume(x == y);
  while
    x++;
  }
  ret1 =
  while (y
    y++;
  }
  ret2 = y;
  assert(ret1 == ret2);
}
```



Problems with this approach

Method

From *2*-safety to *1*-safety!

Problems with this approach

- Verifying the new program can be very difficult

With self-composition, verification requires exact (or very precise) post-conditions

Method

From *2*-safety to *1*-safety!

Problems with this approach

Method

From *2*-safety to *1*-safety!

- Verifying the new program can be very difficult

  With self-composition, verification requires exact (or very precise) post-conditions

- Scalability: the new program can be very large

  Many strategies cause exponential blow-up

Our work

Cartesian Hoare Logic
avoids both problems!

Our work

Similar to previous work, we reduce verification of *k*-safety to verification of *1*-safety

Cartesian Hoare Logic
avoids both problems!

Our work

Cartesian Hoare Logic
avoids both problems!

Similar to previous work, we reduce verification of *k*-safety to verification of *1*-safety

Instead of using one monolithic program, *CHL* decomposes the proof into many small Hoare Triples that are easy to verify

Our work

Cartesian Hoare Logic avoids both problems!

Similar to previous work, we reduce verification of *k*-safety to verification of *1*-safety

Instead of using one monolithic program, *CHL* decomposes the proof into many small Hoare Triples that are easy to verify

Furthermore, *CHL* is easy to automate: devised verification algorithm that performs backtracking search over the proof rules

Cartesian Hoare Triples

Cartesian Hoare Triples

Hoare Logic can be used
to prove Hoare Triples

$\{ P \}\ \mathbf{S}\ \{ Q \}$

If input x satisfies P
then output $\mathbf{S}(x)$ satisfies Q

Cartesian Hoare Triples

Hoare Logic can be used to prove Hoare Triples

$\{ P \}\ \mathbf{S}\ \{ Q \}$

If input x satisfies P then output $\mathbf{S}(\mathrm{x})$ satisfies Q

CHL can be used to prove Cartesian Hoare Triples

$[\![P]\!]\ \mathbf{S}\ [\![Q]\!]$

If vector of inputs $\boldsymbol{x}$ satisfies P then vector of outputs $\mathbf{S}(\boldsymbol{x})$ satisfies Q

Cartesian Hoare Triples

Hoare Logic can be used to prove Hoare Triples

$\{ P \}\ \mathbf{S}\ \{ Q \}$

If input x satisfies P then output $\mathbf{S}(x)$ satisfies Q

CHL can be used to prove Cartesian Hoare Triples

$[\![P]\!]\ \mathbf{S}\ [\![Q]\!]$

If vector of inputs $\boldsymbol{x}$ satisfies P then vector of outputs $\mathbf{S}(\boldsymbol{x})$ satisfies Q

$$[\![x_1 = y_2 \wedge x_2 = y_1]\!]$$

$$z := x - y$$

$$[\![z_1 = z_2]\!]$$

Cartesian Hoare Triples

Hoare Logic can be used to prove Hoare Triples

$\{ P \}\ \mathbf{S}\ \{ Q \}$

If input x satisfies P then output $\mathbf{S}(x)$ satisfies Q

CHL can be used to prove Cartesian Hoare Triples

$[\![P]\!]\ \mathbf{S}\ [\![Q]\!]$

If vector of inputs $\boldsymbol{x}$ satisfies P then vector of outputs $\mathbf{S}(\boldsymbol{x})$ satisfies Q

$$[\![x_1 = y_2 \wedge x_2 = y_1]\!]$$

$$z := x - y$$

$$[\![z_1 = z_2]\!]$$

Cartesian Hoare Triples

Hoare Logic can be used to prove Hoare Triples

$\{ P \}\ \mathbf{S}\ \{ Q \}$

If input x satisfies P then output $\mathbf{S}(x)$ satisfies Q

CHL can be used to prove Cartesian Hoare Triples

$[\![P]\!]\ \mathbf{S}\ [\![Q]\!]$

If vector of inputs $\boldsymbol{x}$ satisfies P then vector of outputs $\mathbf{S}(\boldsymbol{x})$ satisfies Q

$$[\![x_1 = y_2 \wedge x_2 = y_1]\!]$$

$$z := x + y$$

$$[\![z_1 = z_2]\!]$$

✔

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; S_1 \circledast \ldots \circledast \; S_n \; \langle \Psi \rangle$$

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; \boxed{S_1 \circledast \ldots \circledast S_n} \; \langle \Psi \rangle$$

$S_1, S_2 \ldots S_n$
do not share state

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; S_1 \circledast \ldots \circledast S_n \; \langle \Psi \rangle$$

$S_1, S_2 \ldots S_n$
do not share state

$[\![\Phi]\!] \; \mathbf{S} \; [\![\Psi]\!]$ is valid

if we can prove

$\langle \Phi \rangle \; S_1 \circledast \ldots \circledast S_n \; \langle \Psi \rangle$

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; \boxed{S_1 \circledast \ldots \circledast S_n} \; \langle \Psi \rangle$$

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; \boxed{S_1 \circledast \ldots \circledast S_n} \; \langle \Psi \rangle$$

Set of all programs equivalent to $S_1 ; \ldots ; S_n$

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; \boxed{S_1 \circledast \ldots \circledast S_n} \; \langle \Psi \rangle$$

Set of all programs equivalent to $S_1 ; \ldots ; S_n$

CHL proof rules allow exploring the provability of ▯ Φ ▯ **S** ▯ Ψ ▯ for all **S'** in the space $S_1 \circledast \ldots \circledast S_n$

CHL Overview

To prove Cartesian Hoare triples, *CHL* proof rules derive judgements of the following form:

$$\vdash \langle \Phi \rangle \; \boxed{S_1 \circledast \ldots \circledast S_n} \; \langle \Psi \rangle$$

Set of all programs equivalent to $S_1 ; \ldots ; S_n$

CHL proof rules allow exploring the provability of $\langle \Phi \rangle$ **S** $\langle \Psi \rangle$ for all **S'** in the space $S_1 \circledast \ldots \circledast S_n$

However, we don't explicitly construct each **S'** in $S_1 \circledast \ldots \circledast S_n$ allowing the verification algorithm to scale

CHL Proof Rules

$$\frac{\vdash \langle \Phi \rangle\ S_1 \circledast \ldots \circledast\ S_n \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ S\ \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi_1 \circledast (\chi_2 \circledast \chi_3) \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ (\chi_1 \circledast \chi_2) \circledast \chi_3 \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ S_1 \circledast \ldots \circledast S_n \langle \Phi' \rangle \quad \vdash \langle \Phi' \rangle\ R_1 \circledast \ldots \circledast R_n \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ S_1;R_1 \circledast \ldots \circledast S_n;R_n \langle \Psi \rangle}$$

$$\frac{\vdash \{\Phi\}\ S\ \{\Psi\}}{\vdash \langle \Phi \rangle\ S\ \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi_2 \circledast \chi_1 \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ \chi_1 \circledast \chi_2 \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ S;^{\flat} \circledast \chi \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ S \circledast \chi \langle \Psi \rangle}$$

$$\frac{\vdash \{\Phi\}\ S_1\ \{\Phi'\} \quad \vdash \langle \Phi' \rangle\ S_2 \circledast \chi \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ S_1;S_2 \circledast \chi \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \wedge c \rangle\ S_1;\ S \circledast \chi \langle \Psi \rangle \quad \vdash \langle \Phi \wedge \neg c \rangle\ S_2;\ S \circledast \chi \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ (S_1 \oplus_c S_2;\ S) \circledast \chi \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi \circledast {}^{\flat} \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}$$

$$\frac{V = \mathrm{vars}(\chi)}{\vdash \langle \Phi \rangle\ \chi \langle \exists V.\Phi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ \chi \circledast {}^{\flat} \langle \Psi \rangle}$$

$$\frac{\Phi \Rightarrow \Phi' \quad \Psi \Rightarrow \Psi' \quad \vdash \langle \Phi' \rangle\ \chi \langle \Psi' \rangle}{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}$$

CHL Proof Rules

$$\frac{\vdash \langle \Phi \rangle\ S_1 \circledast \ldots \circledast\ S_n \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ S\ \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi_1 \circledast (\chi_2 \circledast \chi_3) \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ (\chi_1 \circledast \chi_2) \circledast \chi_3 \langle \Psi \rangle}$$

$$\frac{\vdash \{\Phi\}\ S\ \{\Psi\}}{\vdash \langle \Phi \rangle\ S\ \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi_2 \circledast \chi_1 \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ \chi_1 \circledast \chi_2 \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ S_1 \circledast \ldots \circledast S_n \langle \Phi' \rangle \quad \ldots \circledast R_n \langle \Psi \rangle}{\ldots \langle \Psi \rangle}$$

$$\frac{\{\Phi'\}\ \vdash \langle \Phi' \rangle\ S_2 \circledast \chi \langle \ldots}{\vdash \langle \Phi \rangle\ S_1;S_2 \circledast \chi \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \wedge \neg c \rangle \ldots}{\vdash \langle \Phi \rangle\ (S_1 \oplus_c S_2;\ S) \circledast \chi \langle \Psi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle \ldots}{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}$$

$$\frac{V = \mathrm{vars}(\chi)}{\vdash \langle \Phi \rangle\ \chi \langle \exists V.\Phi \rangle}$$

$$\frac{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}{\vdash \langle \Phi \rangle\ \chi \circledast {}^{b} \langle \Psi \rangle}$$

$$\frac{\Phi \Rightarrow \Phi' \quad \Psi \Rightarrow \Psi' \quad \vdash \langle \Phi' \rangle\ \chi \langle \Psi' \rangle}{\vdash \langle \Phi \rangle\ \chi \langle \Psi \rangle}$$

NO TIME

Details in the paper!

Automation

Automation

Prototype tool for verifying *k*-safety properties of Java programs[1]

[1]https://github.com/marcelosousa/descartes

Automation

DESCARTES

Prototype tool for verifying *k*-safety properties of Java programs[1]

- Performs backtracking search over the space of proofs
- Uses a guess-and-check procedure for loop invariant generation
- Able to generate counter-examples (i.e. inputs that violate property)

[1]https://github.com/marcelosousa/descartes

Evaluation

Check correctness of custom relational operators in Java

Evaluation

Check correctness of custom relational operators in Java

Evaluation

Check correctness of custom relational operators in Java

equals

totality

Symmetry

compare

Anti-symmetry

transitivity

compareTo

Criteria

1. Uncover k-safety violations in buggy programs?
2. Sufficiently precise? (False positive rate?)
3. Practical? (How long does verification take?)

Experiments

Experiments

32 buggy and fixed comparators
from stack**overflow**

Experiments

32 buggy and fixed comparators
from stack**overflow**

...

Around 2K LOC
from 62 relational operators
retrieved from popular open
source projects

Results

Benchmark	P1 V	P1 t(s)	P2 V	P2 t(s)	P3 V	P3 t(s)
ArrayInt	✔	0.14	✔	0.16	✘	0.27
ArrayInt†	✔	0.19	✔	0.41	✔	0.38
CatBPos	✘	0.33	✘	8.35	✘	2.59
Chromosome	✔	0.15	✘	0.19	✘	0.34
Chromosome†	✔	0.36	✔	4.26	✔	3.90
ColItem	✘	0.09	✘	0.15	✘	0.17
ColItem†	✔	0.07	✔	0,17	✔	0.17
Contact	✔	0.11	✘	1.14	✘	1.85
Container-v1	✘	0.04	✘	0.04	✘	0.04
Container-v2	✘	0.05	✘	0.04	✘	0.04
Container†	✔	0.27	✔	3.44	✔	1.28
FileItem	✔	0.03	✔	0.03	✘	0.09
FileItem†	✔	0.05	✔	0.06	✔	0.08
IsoSprite-v1	✘	0.06	✘	0.04	✘	0.07
IsoSprite-v2	✘	0.40	✘	1.82	✔	0.18
Match	✘	0.05	✔	0.04	✘	0.06
Match†	✔	0.05	✔	0.05	✔	0.06
Name	✘	0.15	✔	0.38	✔	0.27
Name†	✔	0.16	✔	0.40	✔	0.50
Node	✔	0.03	✔	0.03	✘	0.09
Node†	✔	0.04	✔	0.06	✔	0.07
PokerHand	✔	0.55	✘	0.71	✘	2.34
PokerHand†	✔	0.61	✔	0.78	✔	1.61
...						

- Correctly identifies the violated property in all buggy programs
- Able to retrieve counterexamples
- No false positives
- Average runtime of 1.2s

Results

…

Around 2K LOC
from 62 relational operators
retrieved from popular open
source projects

- Low false positive rate (8%)
- Uncovered 5 buggy operators
- Average runtime of 9.1s

Conclusion

CHL: a new program logic for verifying *k*-safety

Prototype for proving *k*-safety for **Java**

Evaluation shows that technique is practical and has a low false positive rate!

Cartesian
k-safety
program
relational
approach
product
proof
loop
Figure
can
first
return
false
rule
rules
using
verifying
form
case
loops
Example
Hence
execute
true
valid
Furthermore
procedure
algorithm
acm
logic
consider
pages
Hoare
Fusion
triple
verify
property
construction
benchmarks
comparator
CHL
triples
different