UpProver Tool:
Incremental Verification by SMT-based Summary Repair

Sepideh Asadi [1] Martin Blicha [1] Antti Hyvärinen [1]

Grigory Fedyukovich [2] Natasha Sharygina [1]

Università della Svizzera italiana

1- Università della Svizzera italiana (USI), Lugano, Switzerland

2- Florida State University, Tallahassee, USA

FMCAD 2020

How to (improve scalability)^2 ?

Software passes through small frequent changes (re-verification is impractical)

Improving scalability by:

1) **Avoiding repetition** in analysis of closely-related program versions (e.g., Incremental verification)

2) Leveraging success of **SMT** (e.g., word-level vs. bit-blasting)

Our take:

- **Reusing** efforts from one verification run to another:

  Function Summarization based on Craig interpolation

- **SMT**-encoding and reasoning

Context

Software **Bounded Model Checking** **[Biere et al. 1999, Clarke et al. 2004]**

- Loops/recursion unrolled up to a predefined bound

Program encoded into a symbolic **BMC formula**

- Via constructing a Static Single Assignment Form

BMC formula conjoined with the negation of **assertions**

- To encode undesired behaviors

Satisfiability check by a SAT/SMT-solver

UNSAT → Bounded program is safe 

SAT → Error found (Satisfying assignment identifies an error trace) 

Need for theory reasoning

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Function summary for func() w.r.t 1st assert

$(a > 0) \Rightarrow (a = c)$

Need for theory reasoning

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

version2.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m + 1;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

Function summary for func() w.r.t 1st assert

$(a > 0) \Rightarrow (a = c)$

Need for theory reasoning

EXPENSIVE SAT encoding

CHEAP

CHEAP

LRA and EUF

version1.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

✔ safe

version2.c

```
int a, b, c;
int func() {
    c = b;
    if (a > 0)
        a = b;
    int i, m = 0;
    for (i = 0; i < 100; i++)
        m += a*b;
    return m + 1;
}
void main() {
    a=nondet(); b=nondet();
    if (a <= 0) return -1;
    b = func();
    assert(a == c);

    int p = func();
    int q = func();
    assert(p == q);
}
```

safe ??

Function summary for func() w.r.t 1st assert

$(a > 0) \Rightarrow (a = c)$

Is this summary good enough to over-approximate the function after change?

The UpProver tool

A BMC tool, incrementally verifies program versions in C

The UpProver tool

A BMC tool, incrementally verifies program versions in C

In contrast to its predecessor, a SAT-based tool eVolCheck:

- ✓ UpProver offers two more levels of SMT-encoding:
  - Equality Logic & Uninterpreted Functions (EUF)
  - Linear Real Arithmetic (LRA)
- ✓ UpProver leverages tree-interpolation systems in SMT to localize and speed up the checks of new versions
- ✓ UpProver offers a trade-off between precision and performance
- ✓ UpProver performs an automated summary **repair**

Function Summarization

Function Summarization

- A technique to create and reuse over-approximation of the function behavior

Function Summarization

- A technique to create and reuse over-approximation of the function behavior

- Expressed using function's in/out parameters

- Contains only the relevant information to prove properties

Function Summarization

- A technique to create and reuse over-approximation of the function behavior

- Expressed using function's in/out parameters

- Contains only the relevant information to prove properties

- Summary is derived by **Craig interpolation** after SMT-solver returns UNSAT

Craig interpolation [Craig'57]

abstraction from proof

Given mutually unsatisfiable formulas A and B, an **interpolant** is a formula I such that

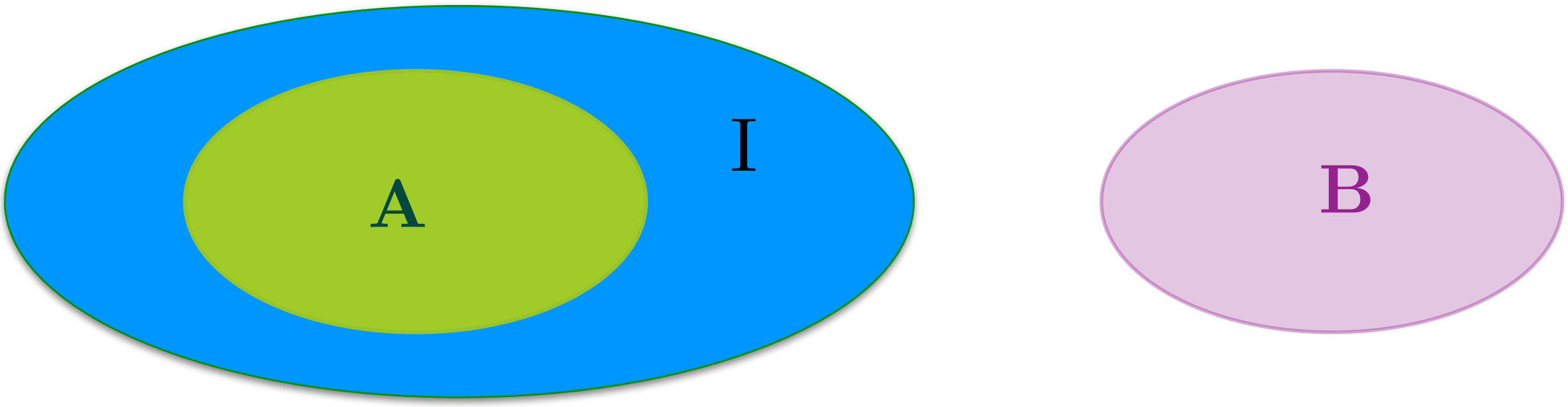
- $\mathrm{A} \rightarrow \mathrm{I}$
- $\mathrm{I} \wedge \mathrm{B}$ is unsatisfiable
- I is defined over common symbols of A and B

Craig interpolation [Craig'57]

abstraction from proof

Given mutually unsatisfiable formulas A and B, an **interpolant** is a formula I such that

- $\mathrm{A} \rightarrow \mathrm{I}$
- $\mathrm{I} \wedge \mathrm{B}$ is unsatisfiable
- I is defined over common symbols of A and B



Craig interpolation [Craig'57]

abstraction from proof

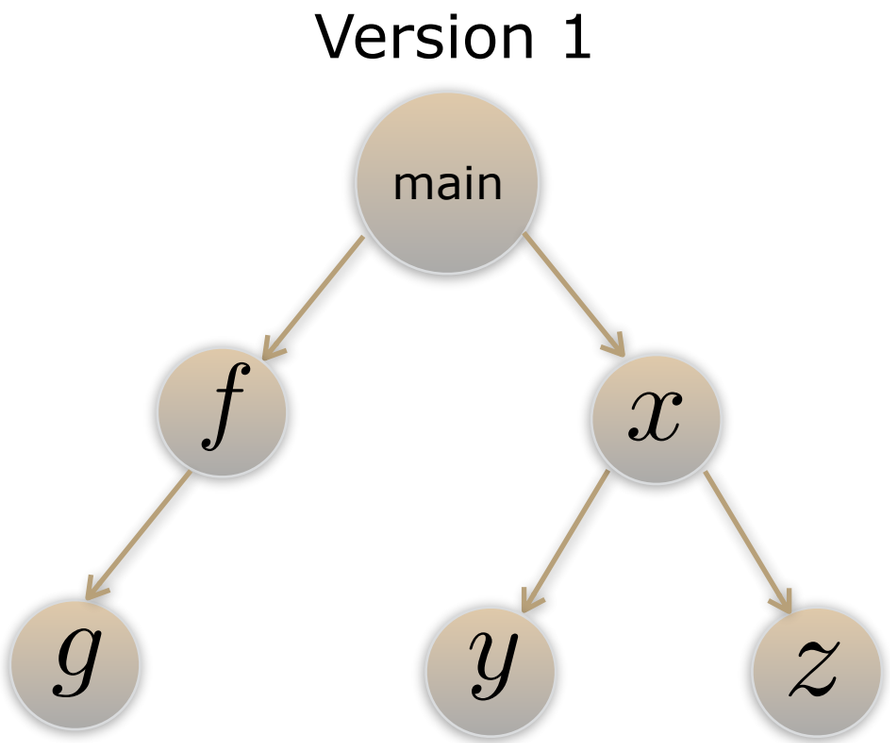
Given mutually unsatisfiable formulas A and B, an **interpolant** is a formula I such that

- $\mathrm{A} \rightarrow \mathrm{I}$
- $\mathrm{I} \wedge \mathrm{B}$ is unsatisfiable
- I is defined over common symbols of A and B

To guarantee the algorithmic correctness of UpProver, tree-interpolation property is required.

For more details see
"Farkas-Based Tree Interpolation", [Asadi et al. SAS'20]

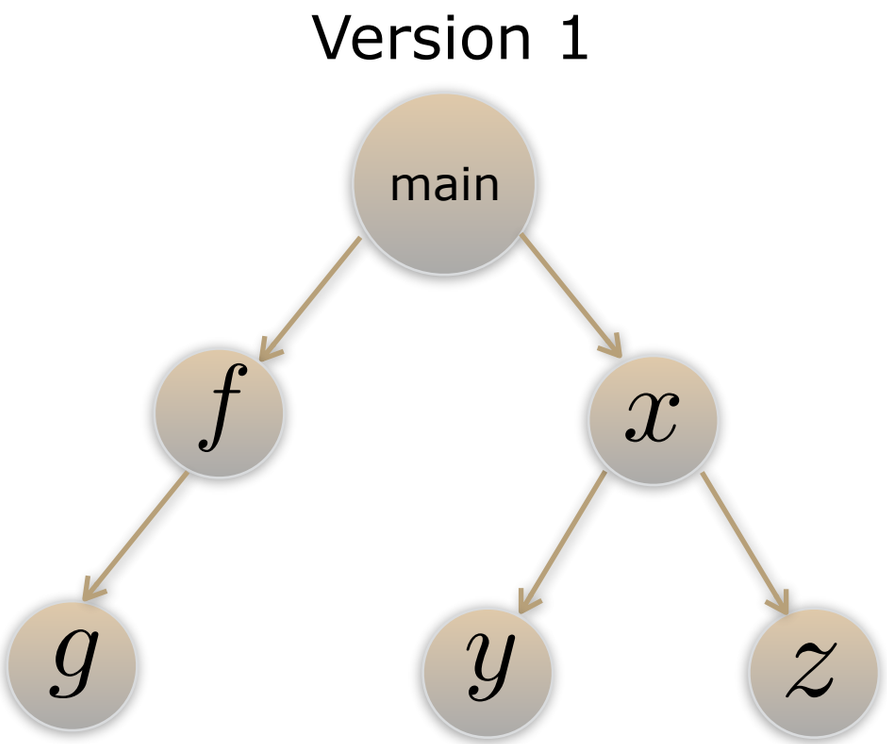
Bootstrap verification



Function call tree

Bootstrap verification

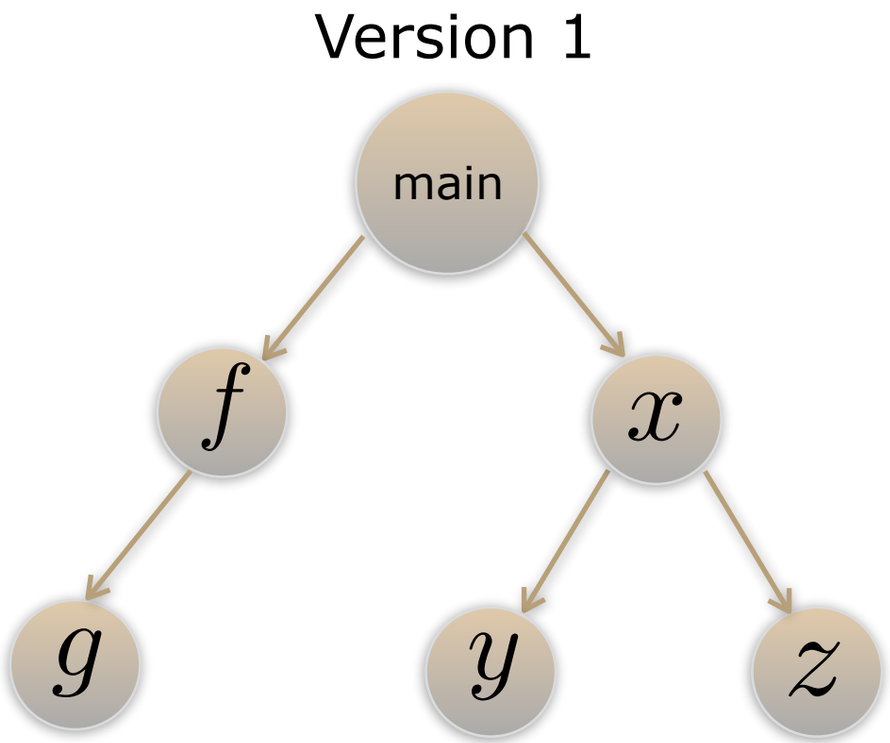
$$\varphi_g \wedge \varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error$$



Function call tree

Bootstrap verification

$$\varphi_g \wedge \varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error$$ → UNSAT ✓

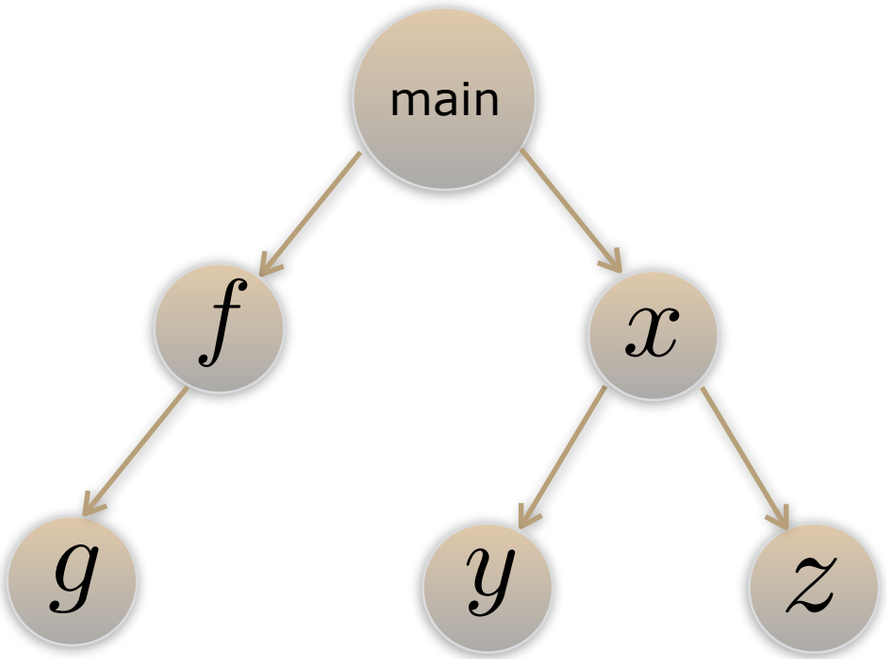


Function call tree

Bootstrap verification

$$\underbrace{\varphi_g}_{} \wedge \underbrace{\varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error}_{} \rightarrow \text{UNSAT} \quad \checkmark$$

Version 1

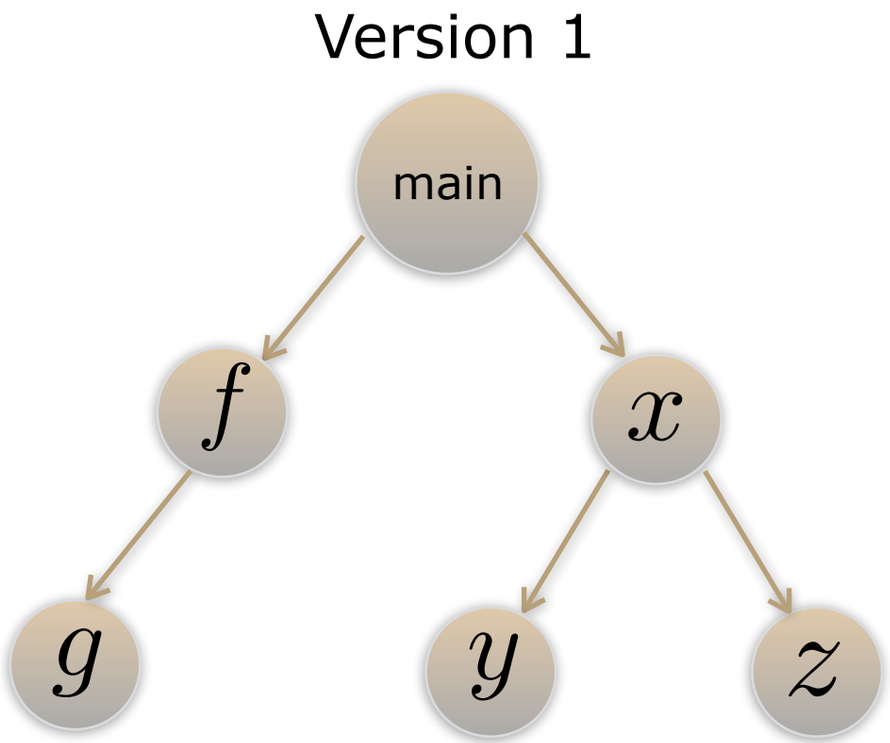


Function call tree

Bootstrap verification

$$\underbrace{\varphi_g}_{\mathcal{S}_g} \wedge \underbrace{\varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error}$$

UNSAT

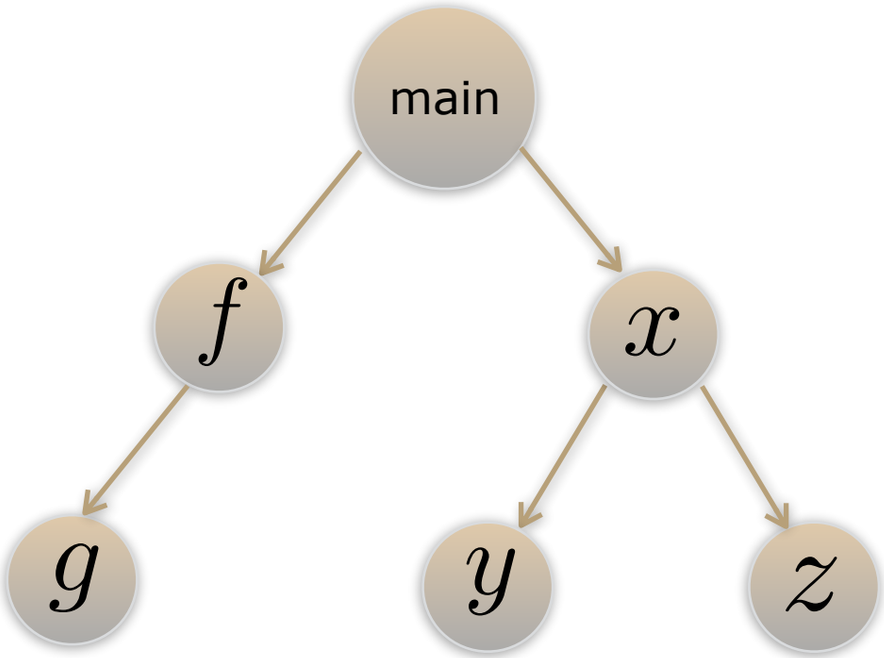


Function call tree

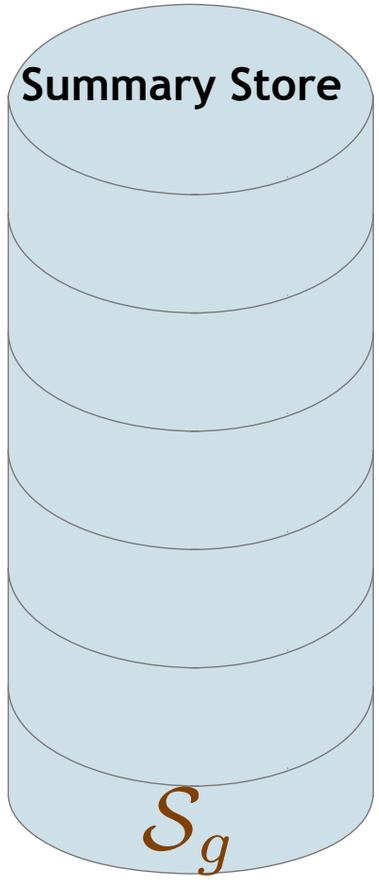
Bootstrap verification

$$\underbrace{\varphi_g}_{} \wedge \underbrace{\varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error}_{} \rightarrow \text{UNSAT} \checkmark$$

Version 1



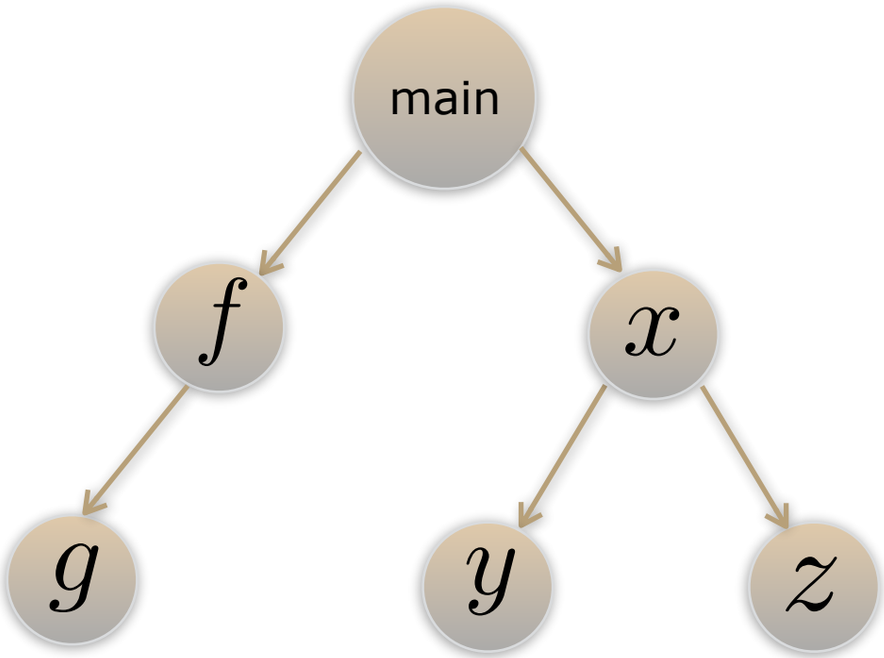
Function call tree



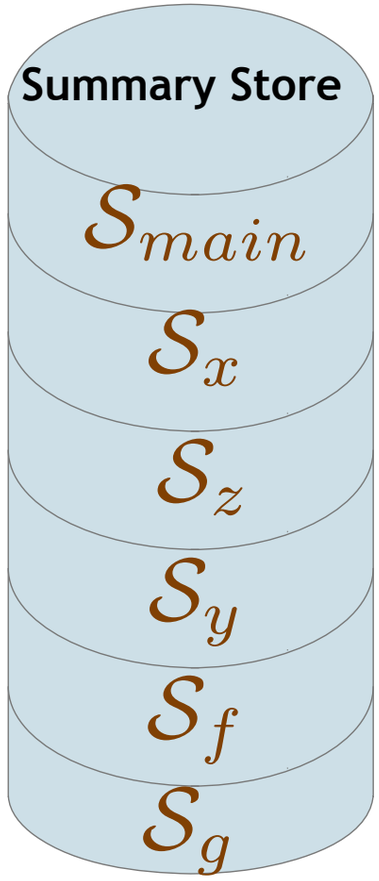
Bootstrap verification

$\varphi_g \wedge \varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error$ → UNSAT ✓

Version 1



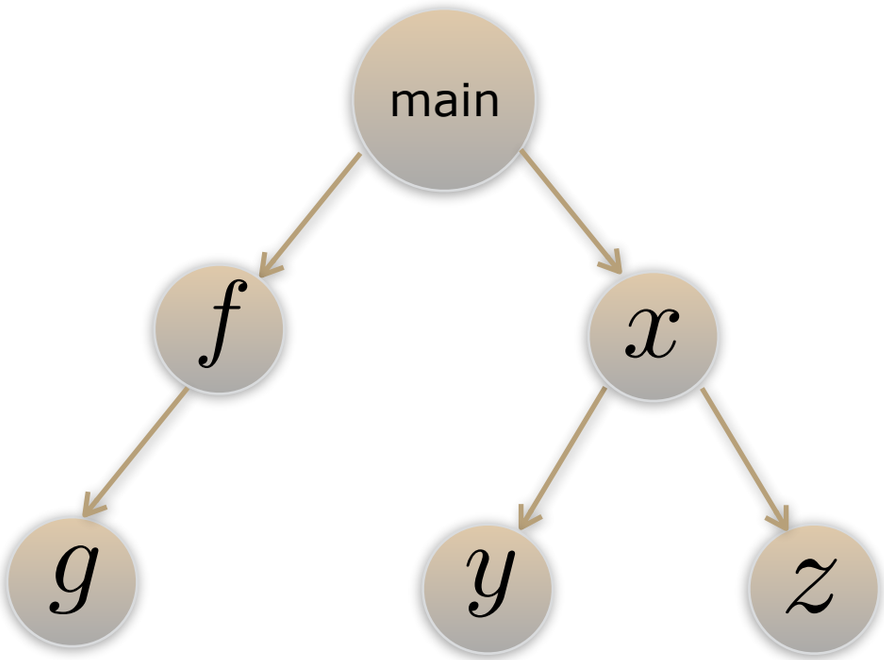
Function call tree



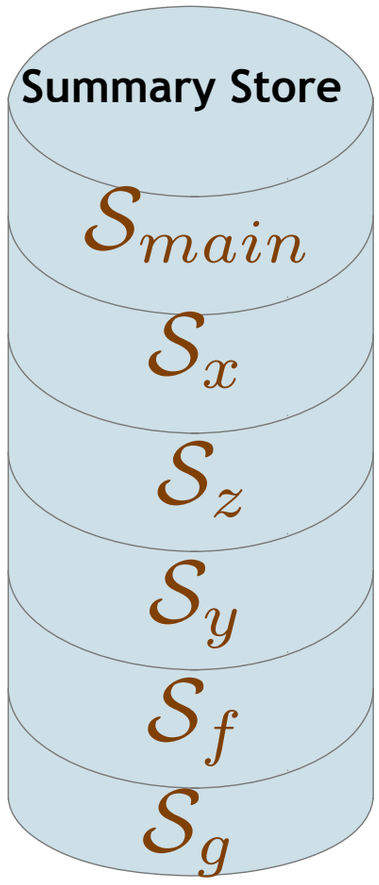
Bootstrap verification

$$\varphi_g \wedge \varphi_f \wedge \varphi_{main} \wedge \varphi_y \wedge \varphi_x \wedge \varphi_z \wedge error$$ → UNSAT ✓

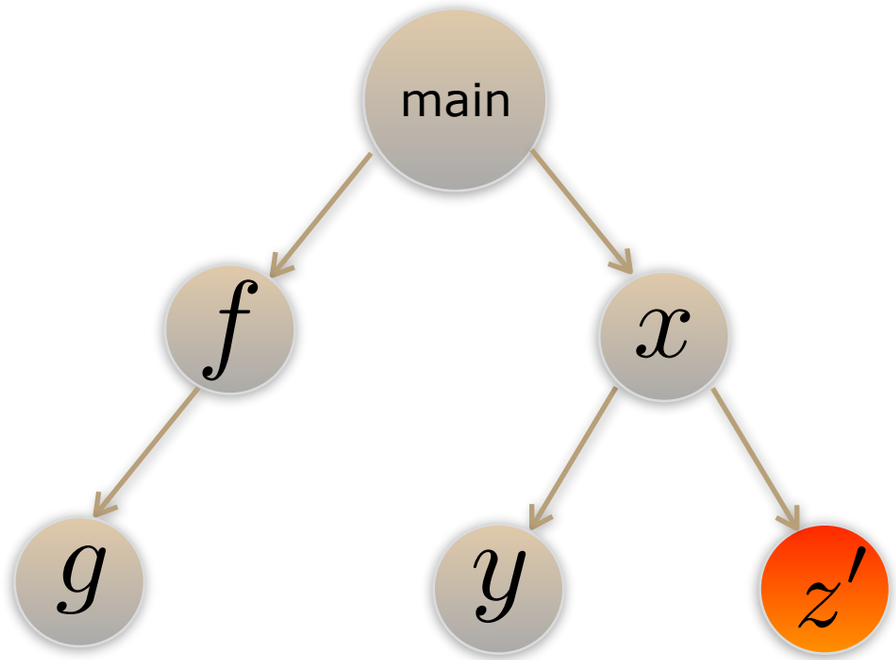
Version 1



Function call tree



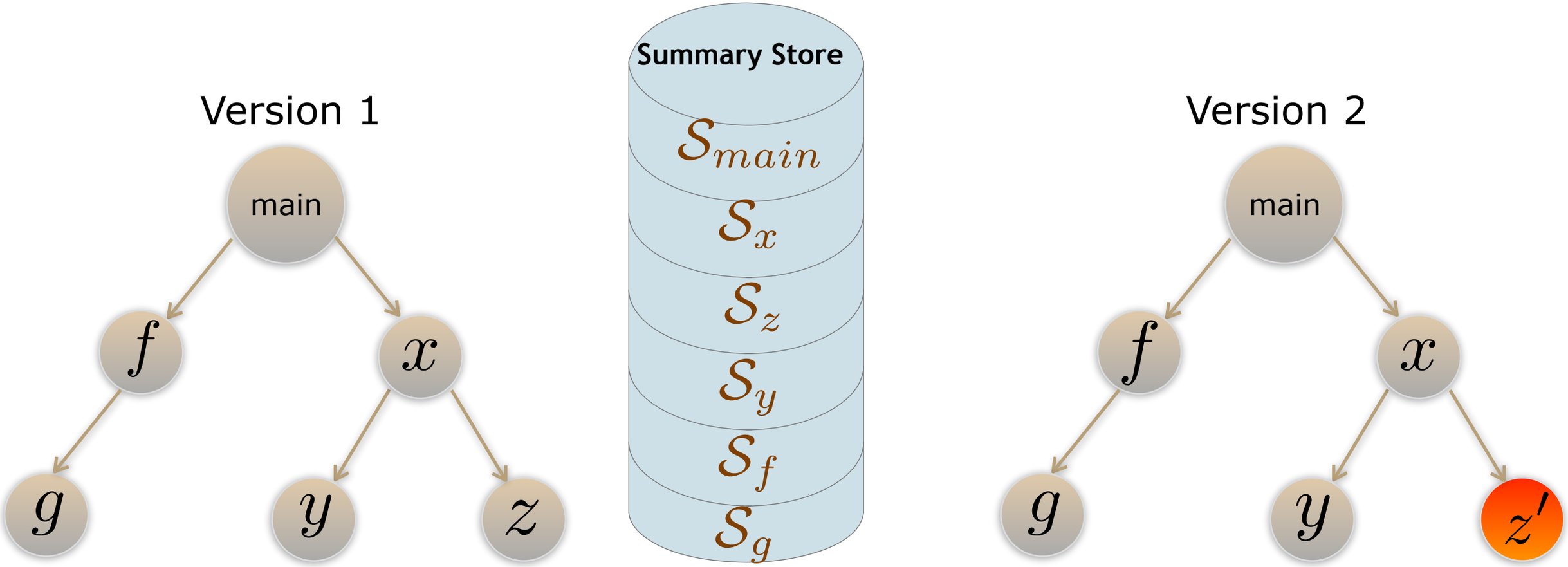
Version 2



Function *z* has changed

Bootstrap verification

We try to check if summary $\mathcal{S}_z$ is still a valid over-approximation of z'

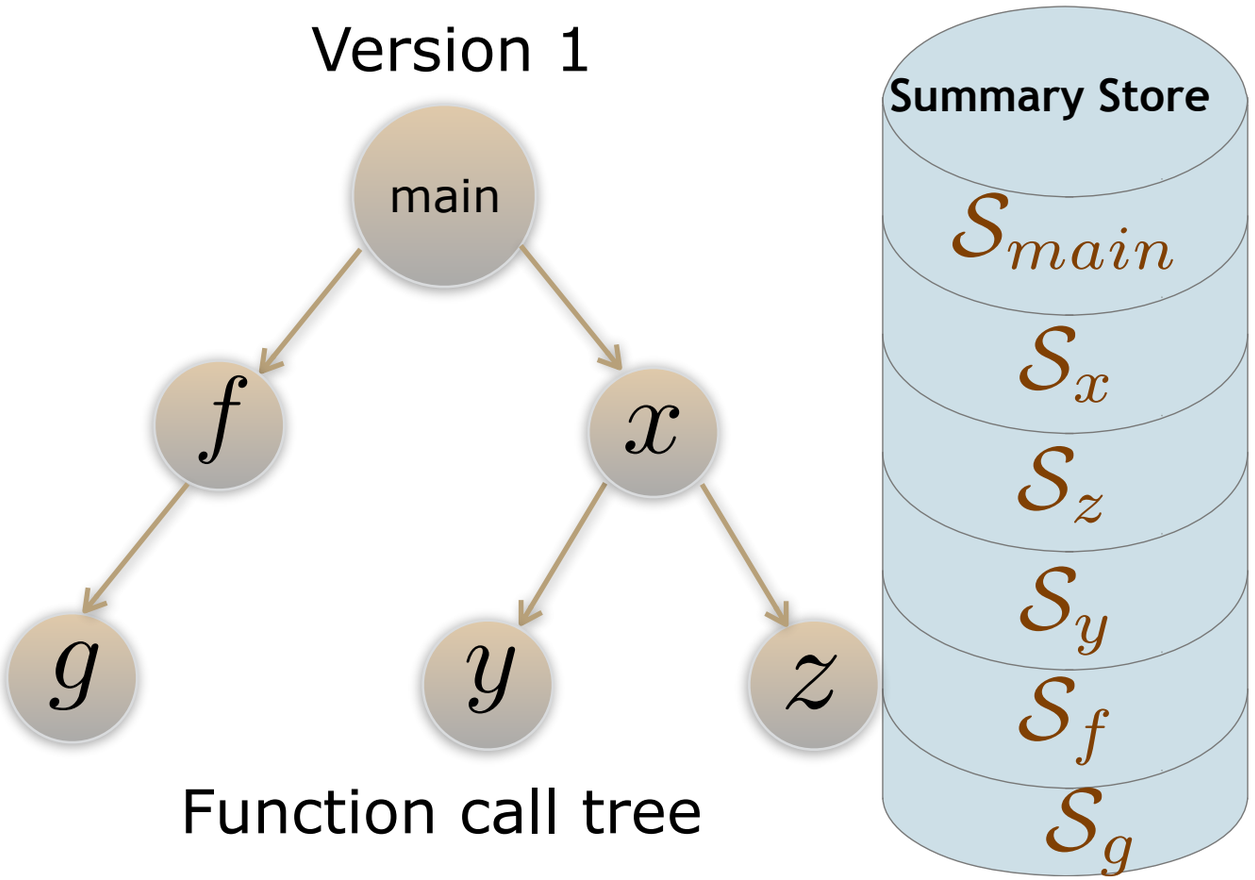


Incremental verification

Validating $\mathcal{S}_z$:

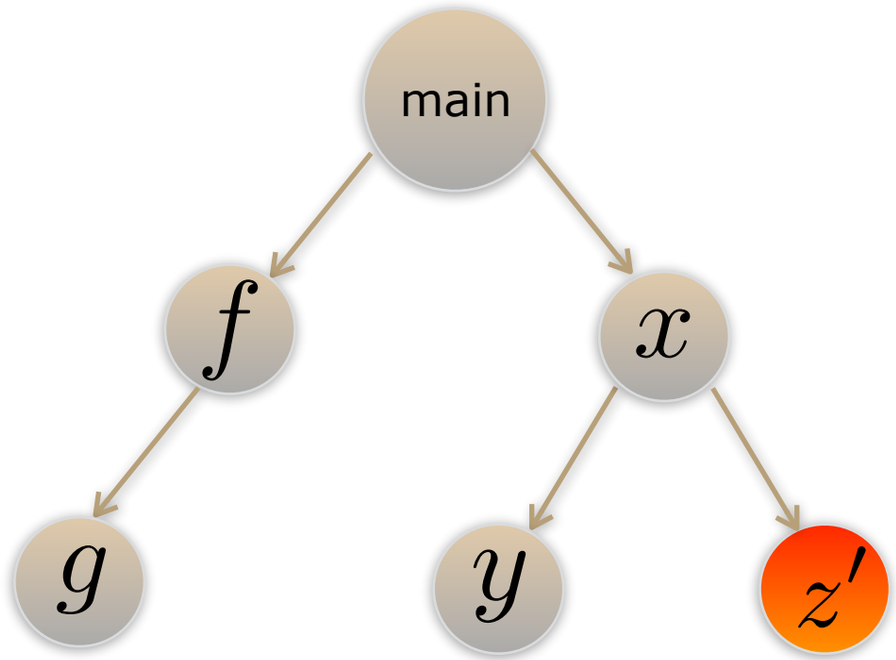
$\varphi_{z'} \implies \mathcal{S}_z$

Version 1



Function call tree

Version 2



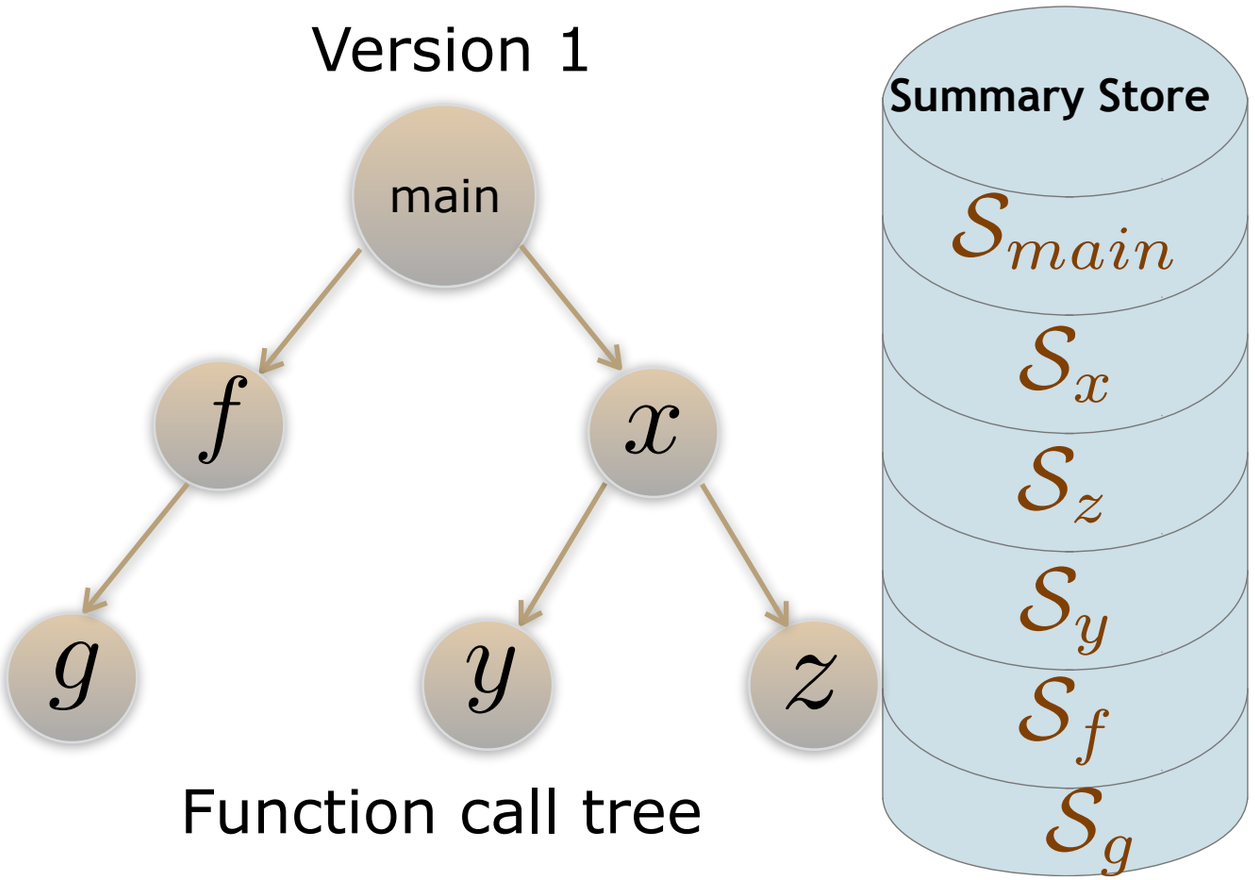
Function *z* has changed

Incremental verification

Validating $\mathcal{S}_z$:

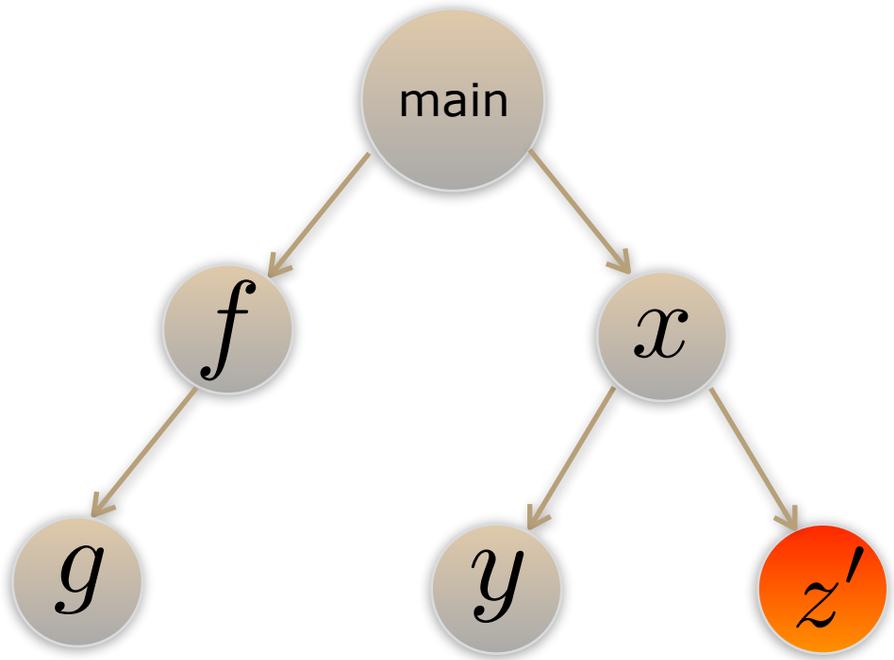
$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Version 1



Function call tree

Version 2



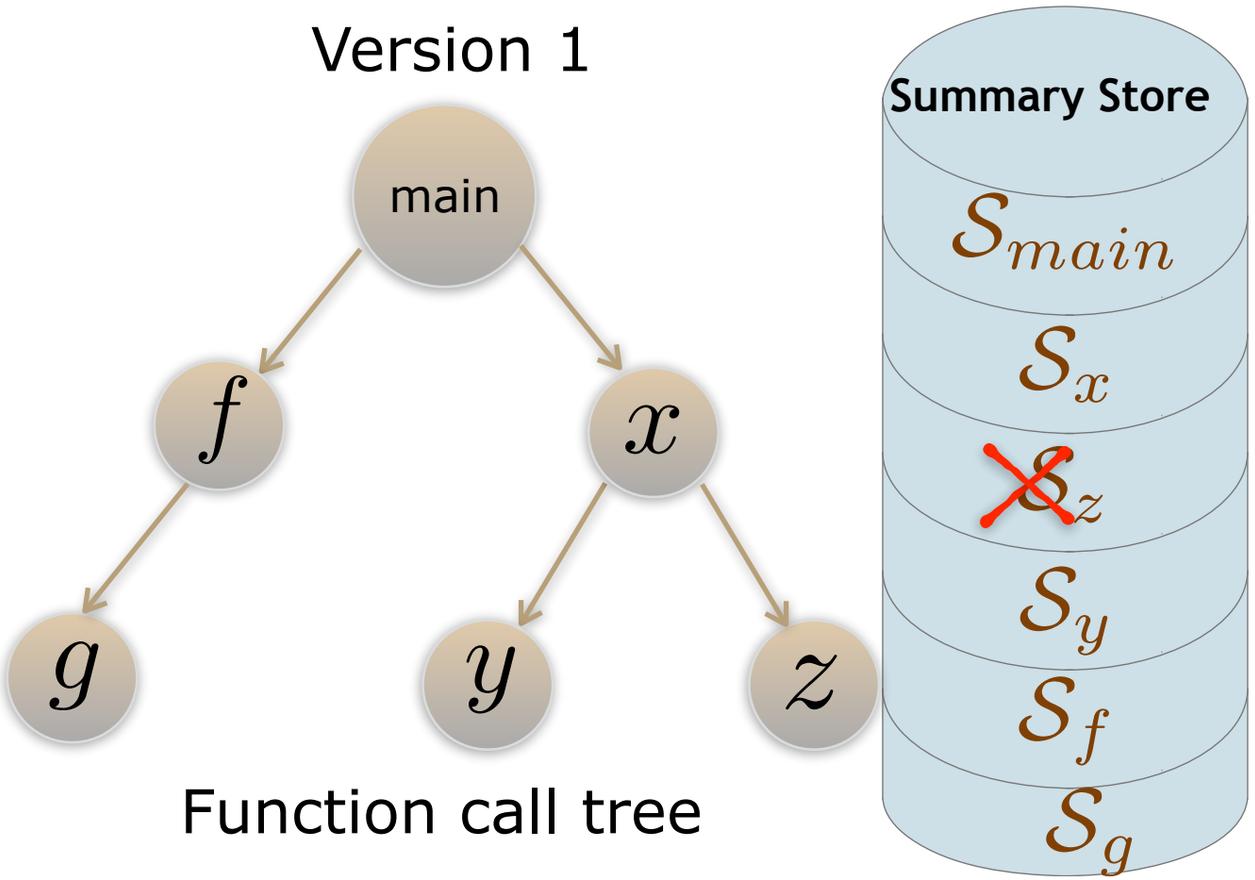
Function *z* has changed

Incremental verification

Validating $\mathcal{S}_z$:

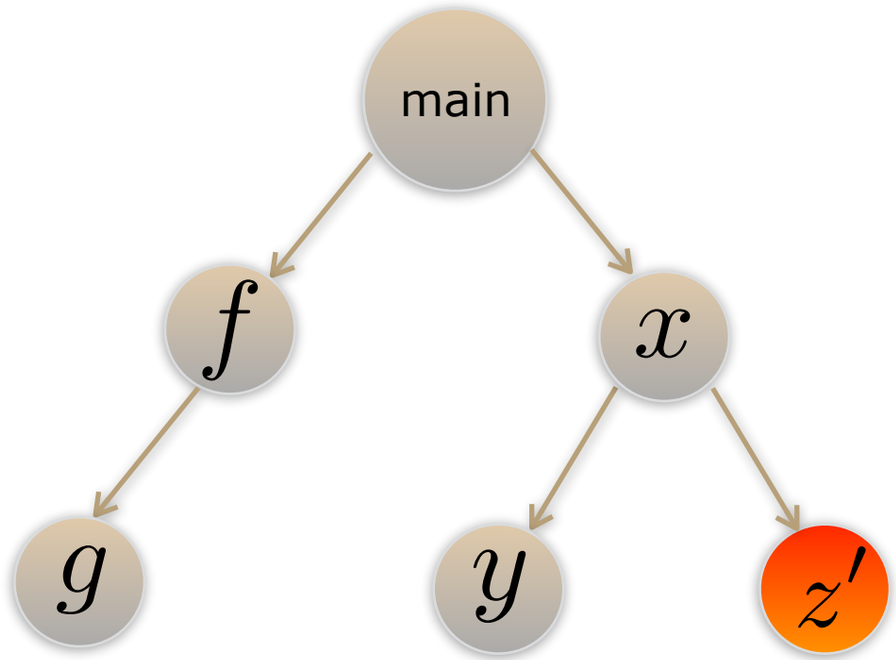
$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Version 1



Function call tree

Version 2



Function *z* has changed

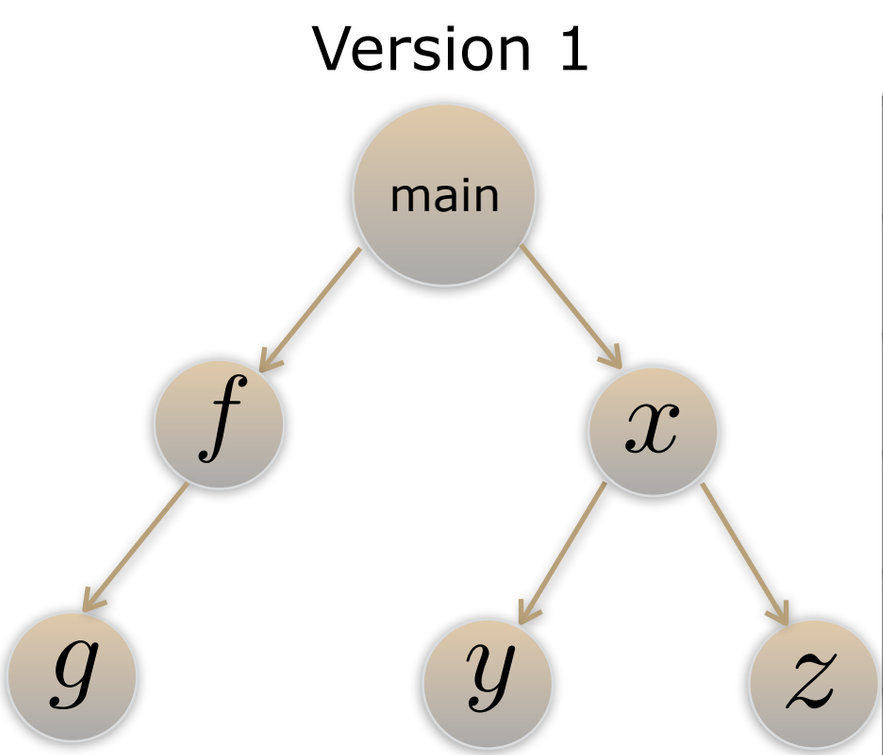
Incremental verification

Validating $\mathcal{S}_z$:

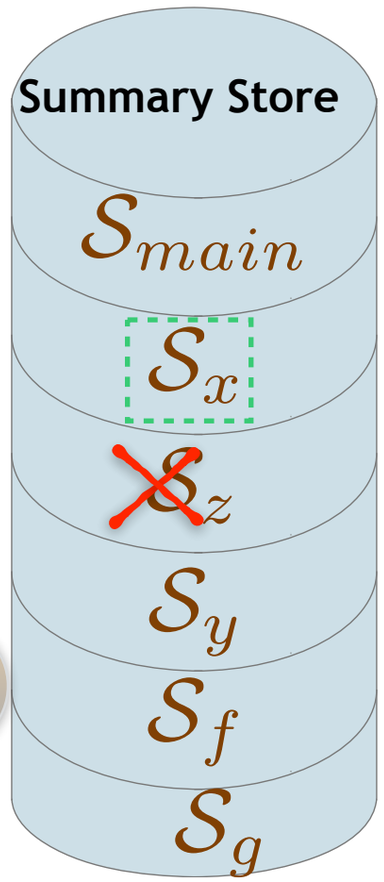
$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Validating $\mathcal{S}_x$:

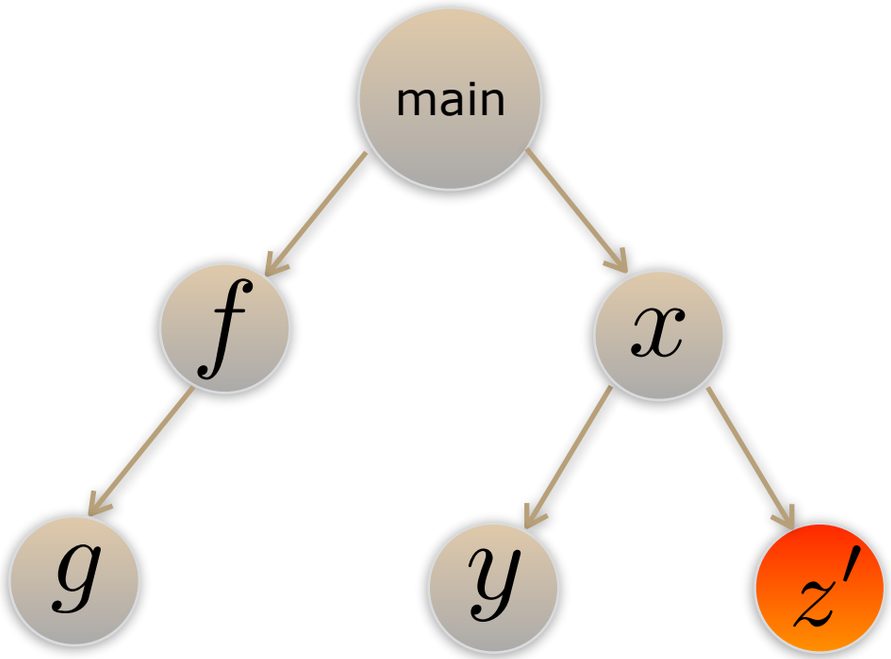
Version 1



Function call tree



Version 2



Function *z* has changed

Incremental verification

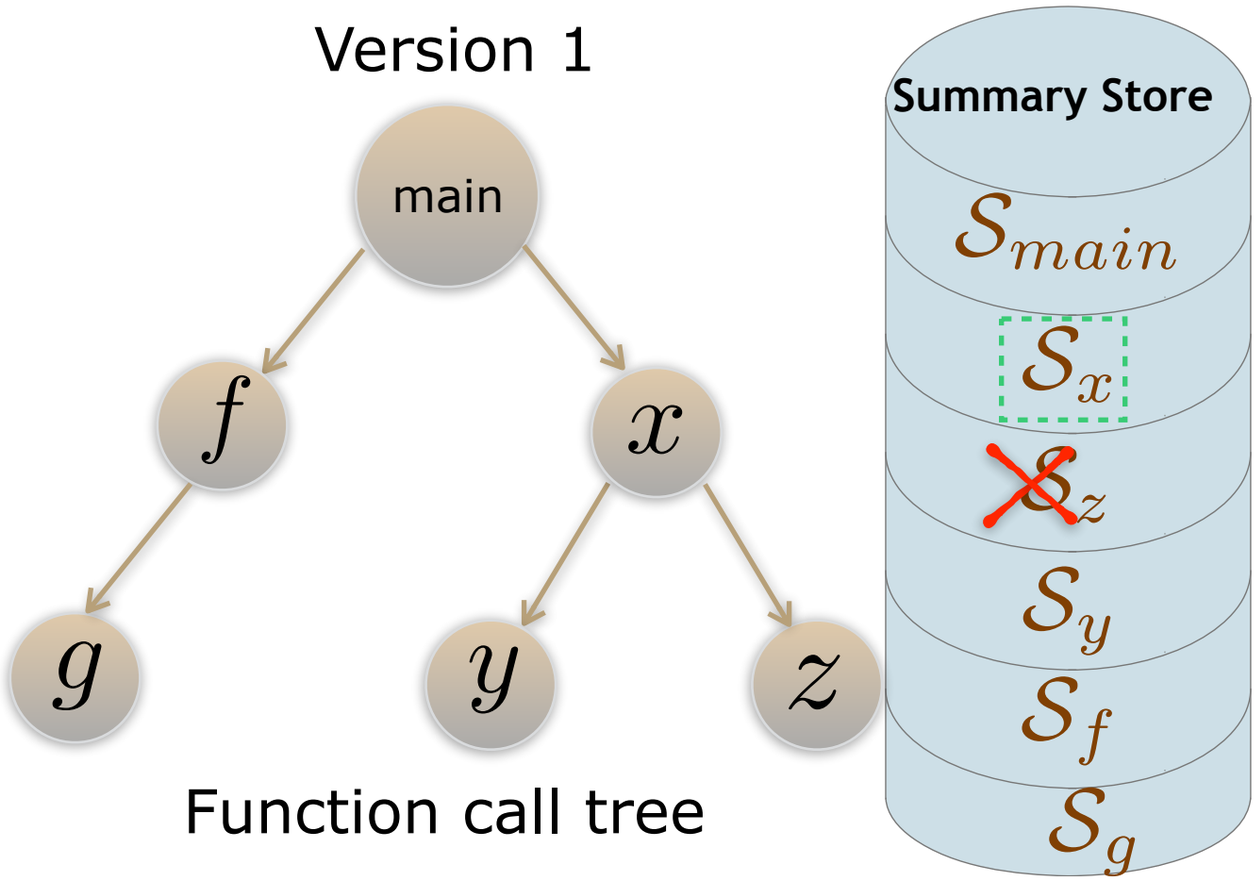
Validating $\mathcal{S}_z$:

$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Validating $\mathcal{S}_x$:

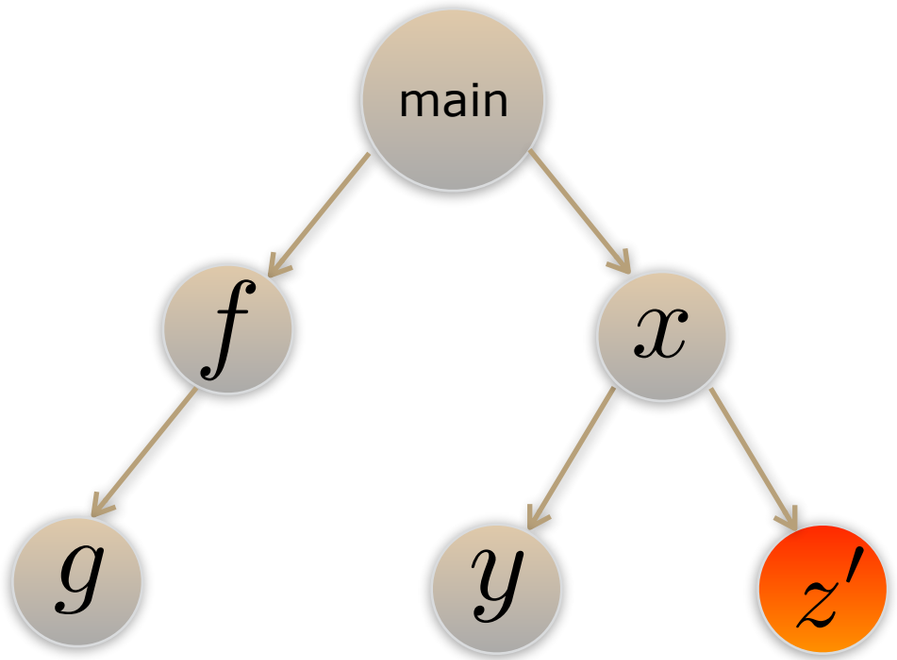
$\varphi'_z \wedge \mathcal{S}_y \wedge \varphi_x \implies \mathcal{S}_x$

Version 1



Function call tree

Version 2



Function *z* has changed

Incremental verification

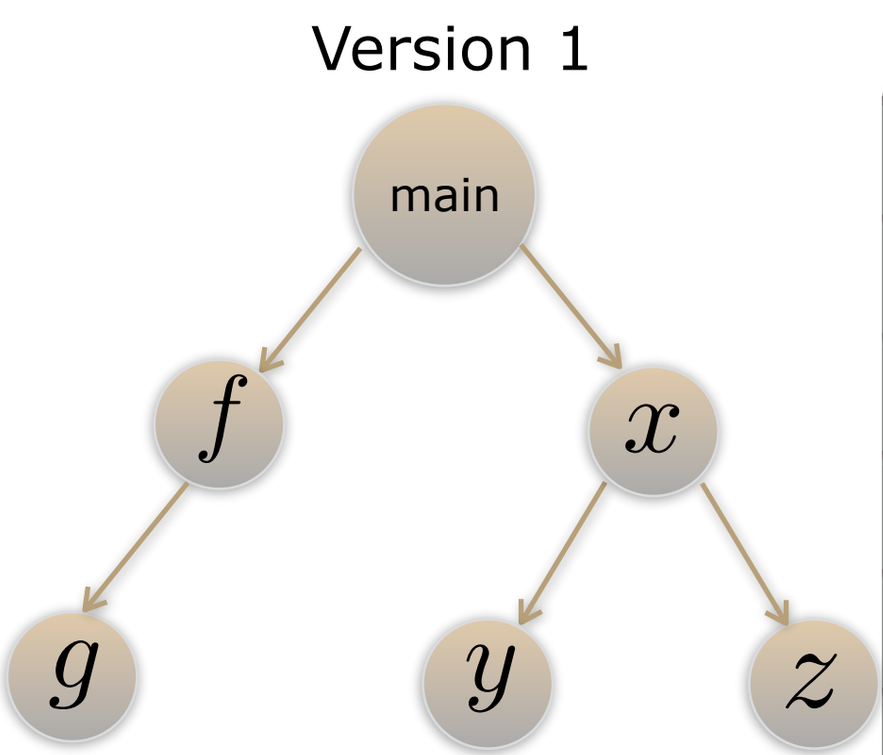
Validating $\mathcal{S}_z$:

$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

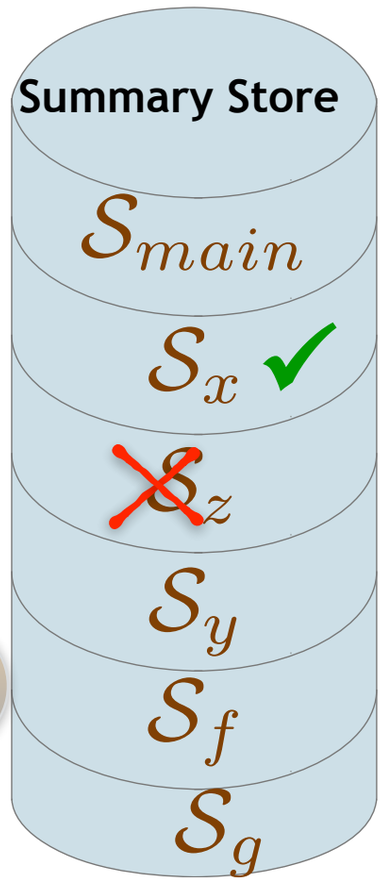
Validating $\mathcal{S}_x$:

$\varphi'_z \wedge \mathcal{S}_y \wedge \varphi_x \implies \mathcal{S}_x$ ✓ **change is safe**

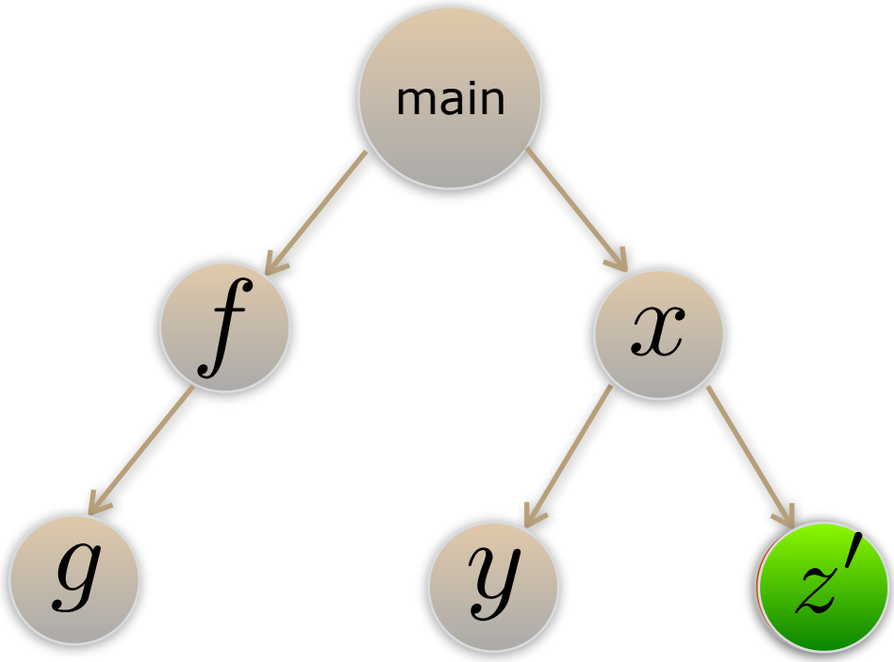
Version 1



Function call tree



Version 2



Function *z* has changed

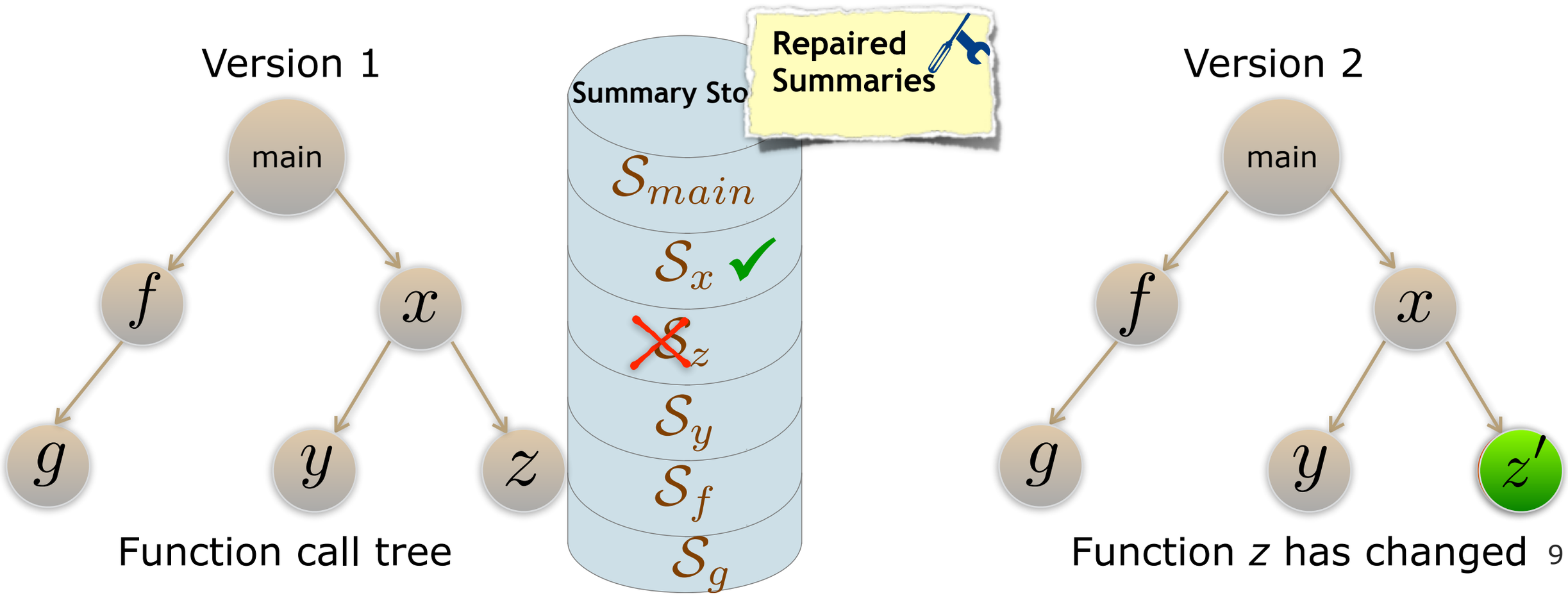
Incremental verification

Validating $\mathcal{S}_z$:

$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Validating $\mathcal{S}_x$:

$\varphi'_z \wedge \mathcal{S}_y \wedge \varphi_x \implies \mathcal{S}_x$ ✓ **change is safe**



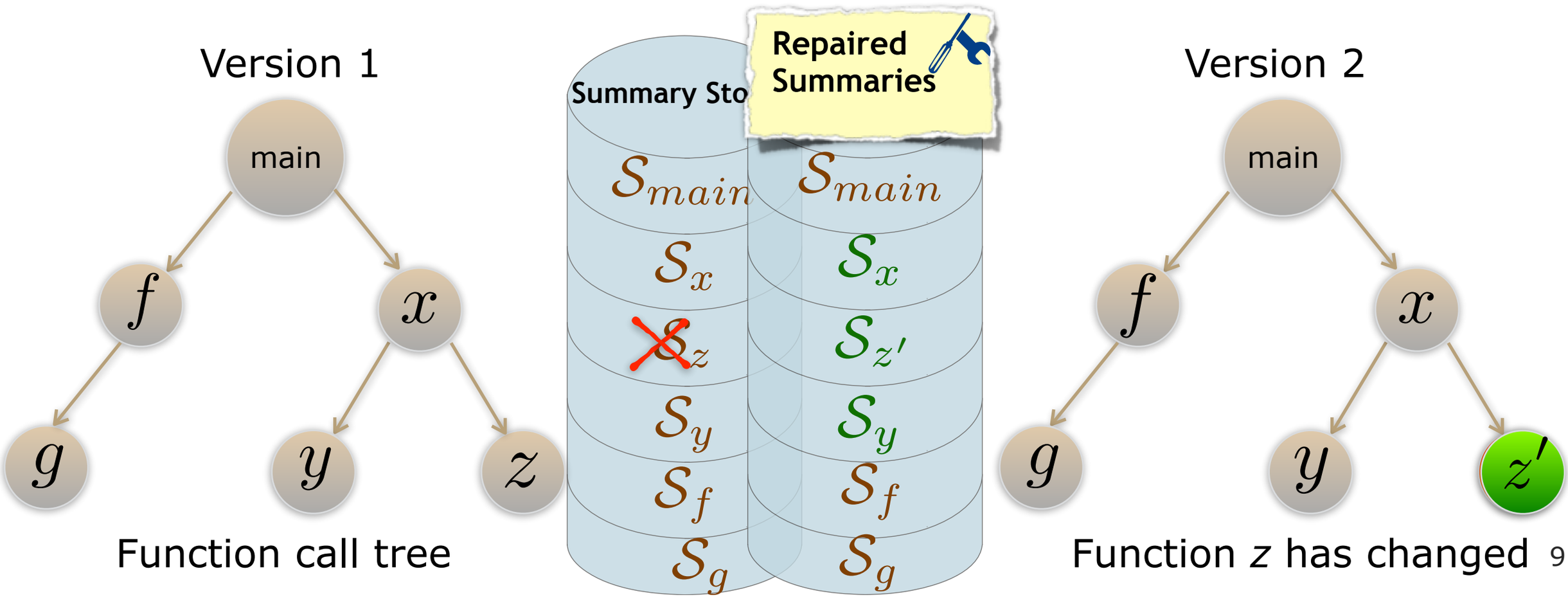
Incremental verification

Validating $\mathcal{S}_z$:

$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Validating $\mathcal{S}_x$:

$\varphi'_z \wedge \mathcal{S}_y \wedge \varphi_x \implies \mathcal{S}_x$ ✓ **change is safe**



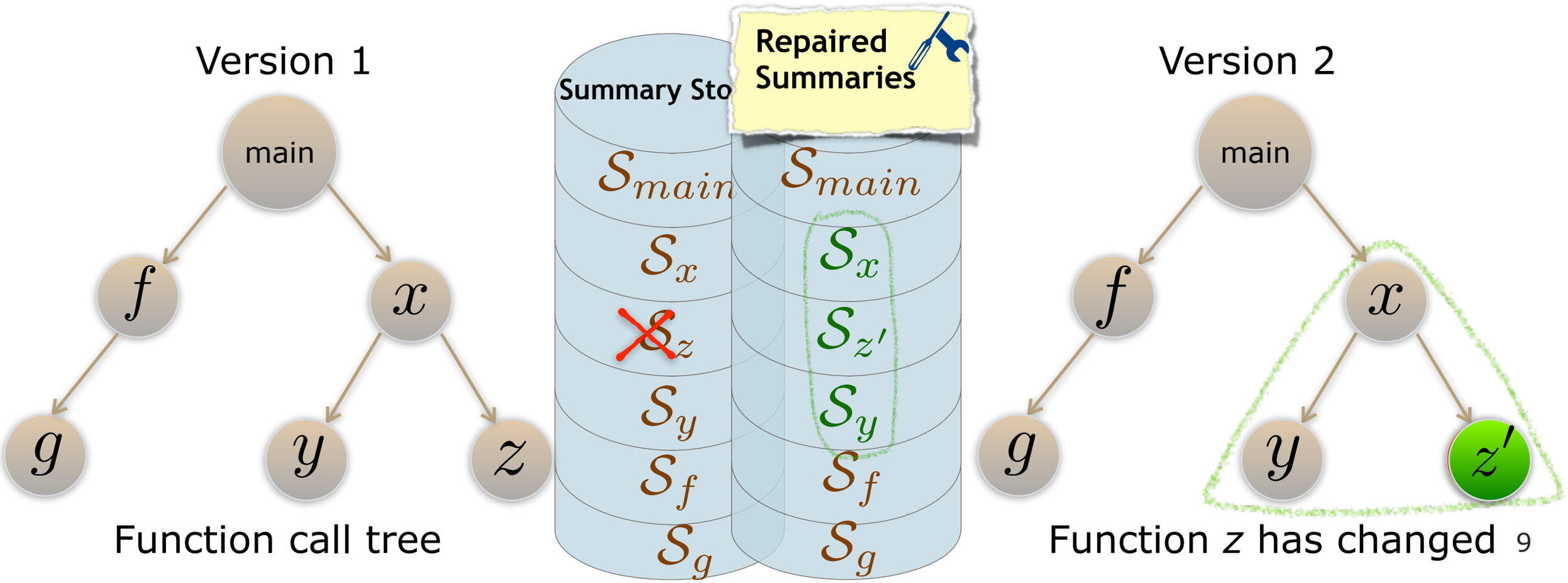
Incremental verification

Validating $\mathcal{S}_z$:

$\varphi_{z'} \implies \mathcal{S}_z$ ✗ **Invalid, Propagate upwards**

Validating $\mathcal{S}_x$:

$\varphi'_z \wedge \mathcal{S}_y \wedge \varphi_x \implies \mathcal{S}_x$ ✓ **change is safe**



UpProver's SMT encodings

- Equality Logic & Uninterpreted Functions (EUF)

  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$

UpProver's SMT encodings

- Equality Logic & Uninterpreted Functions (EUF)

  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$

- Linear Real Arithmetic (LRA)

  - Example: $(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee (x = 1) \vee (y \geq 0))$

UpProver's SMT encodings

- Equality Logic & Uninterpreted Functions (EUF)

  - Example: $(f(x,y) \neq f(u,v)) \wedge (x = u) \wedge (y = v)$

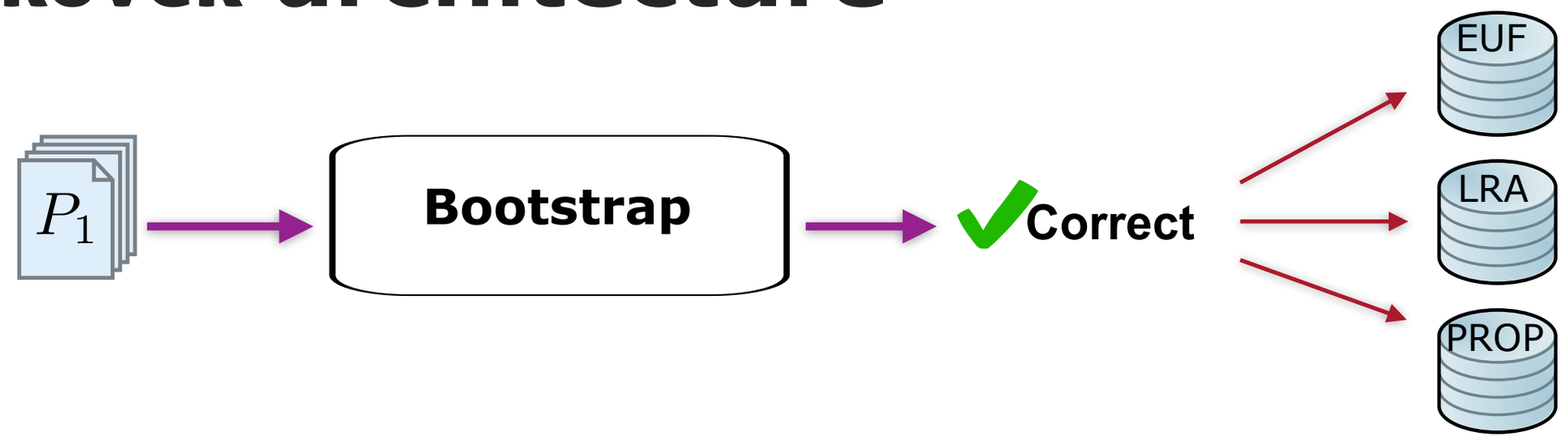
- Linear Real Arithmetic (LRA)

  - Example: $(x + y \leq 0) \wedge (x = 0) \wedge (\neg a \vee (x = 1) \vee (y \geq 0))$

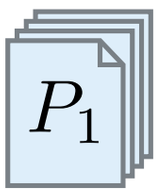
- Propositional Logic (Prop)

  - Example: $\left[(a + b)\,\%\,2 \neq ((a\,\%\,2) + (b\,\%\,2))\,\%\,2\right]$

UpProver architecture



UpProver architecture

safe program + assertions 

Summaries of P_1

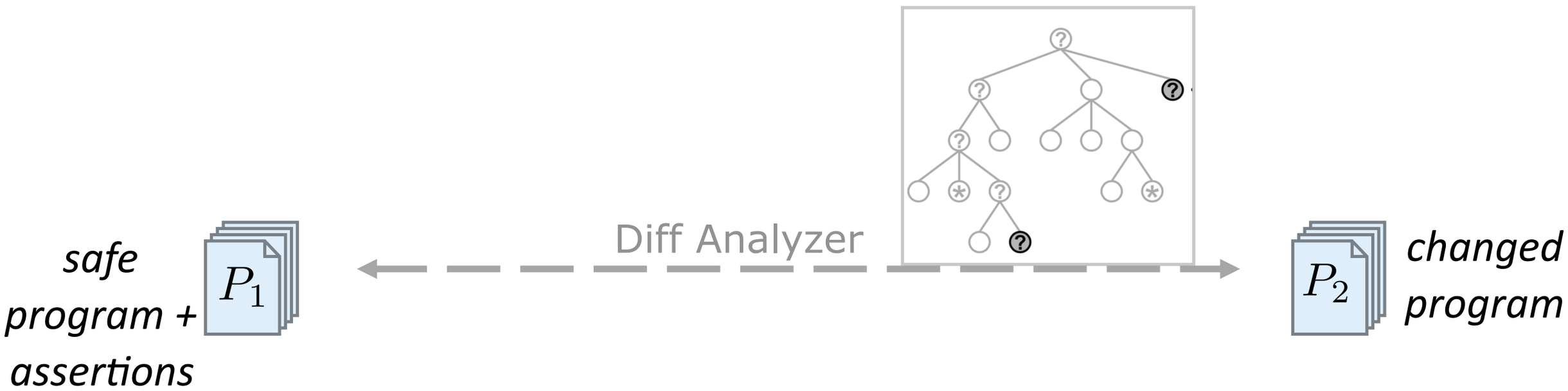
UpProver architecture

safe program + assertions P_1

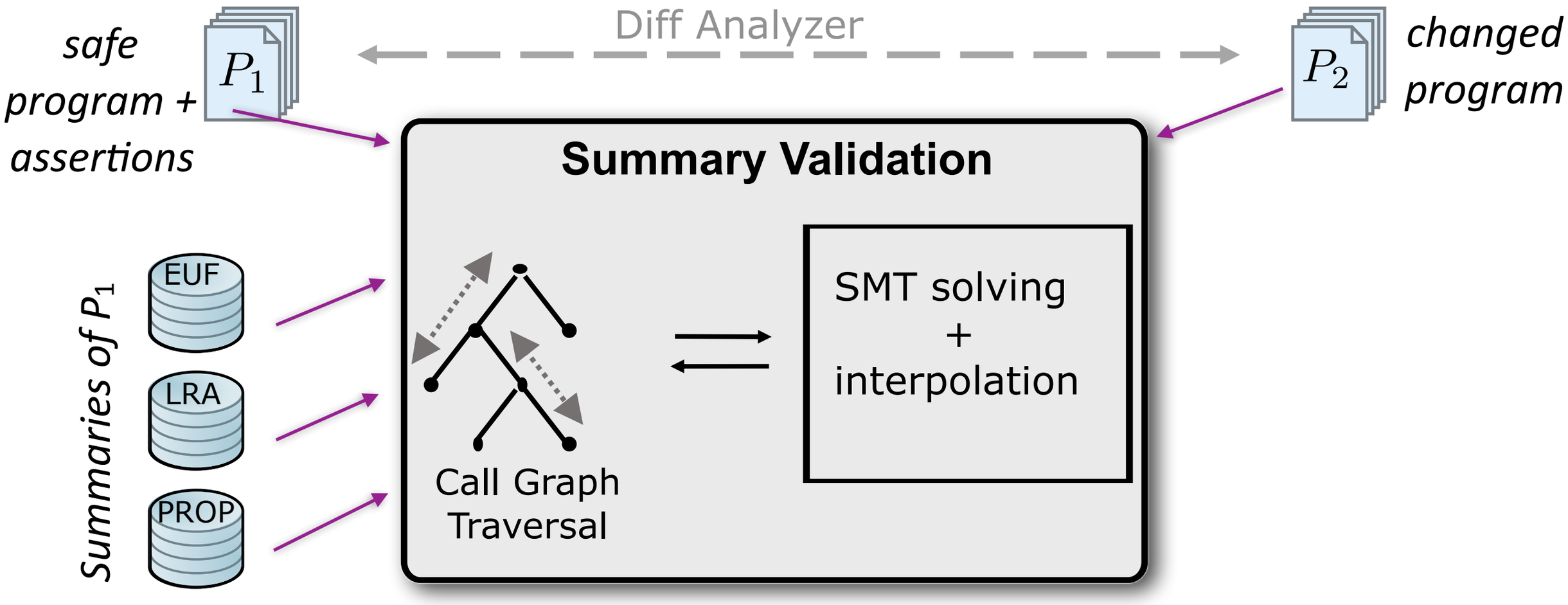
Summaries of P_1

P_2 *changed program*

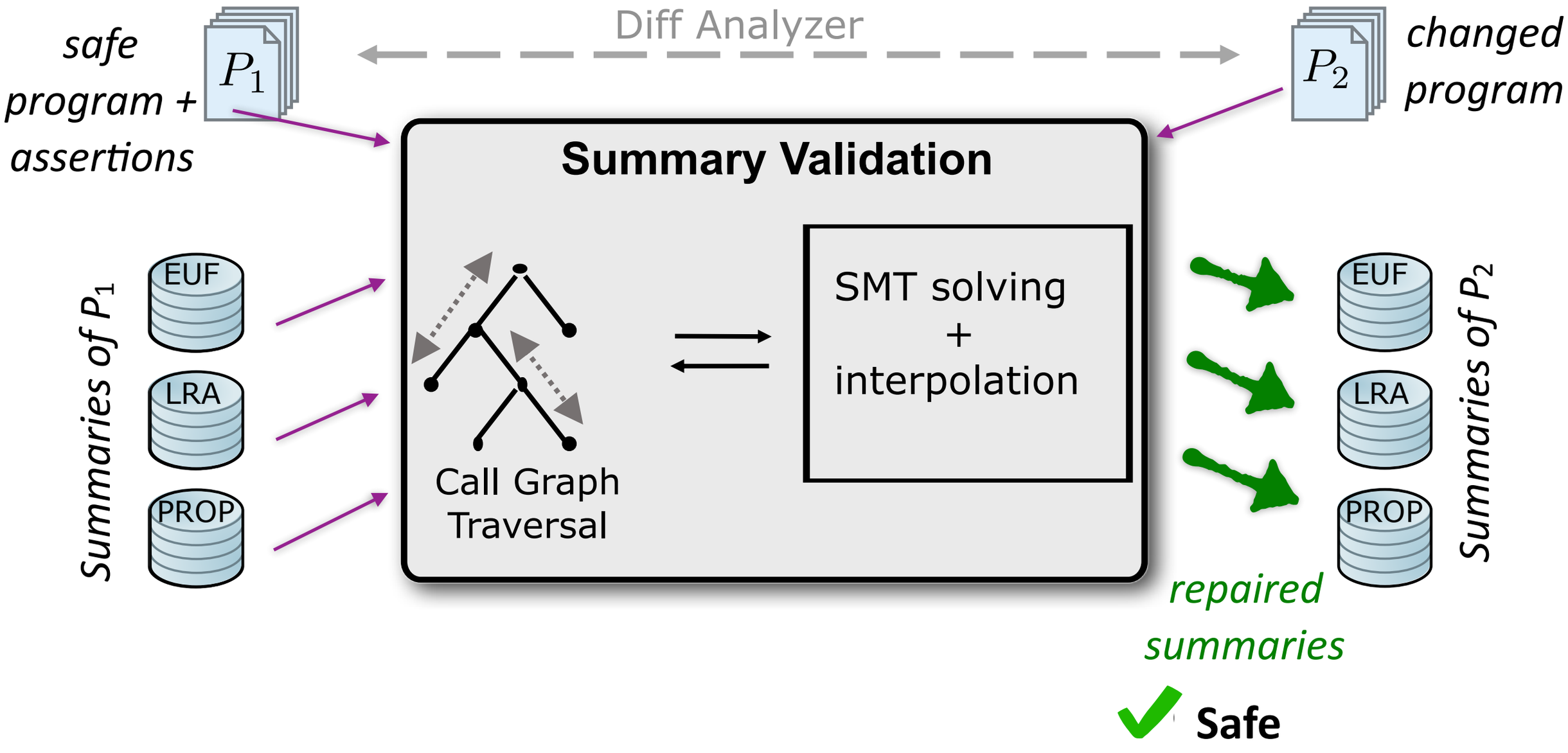
UpProver architecture



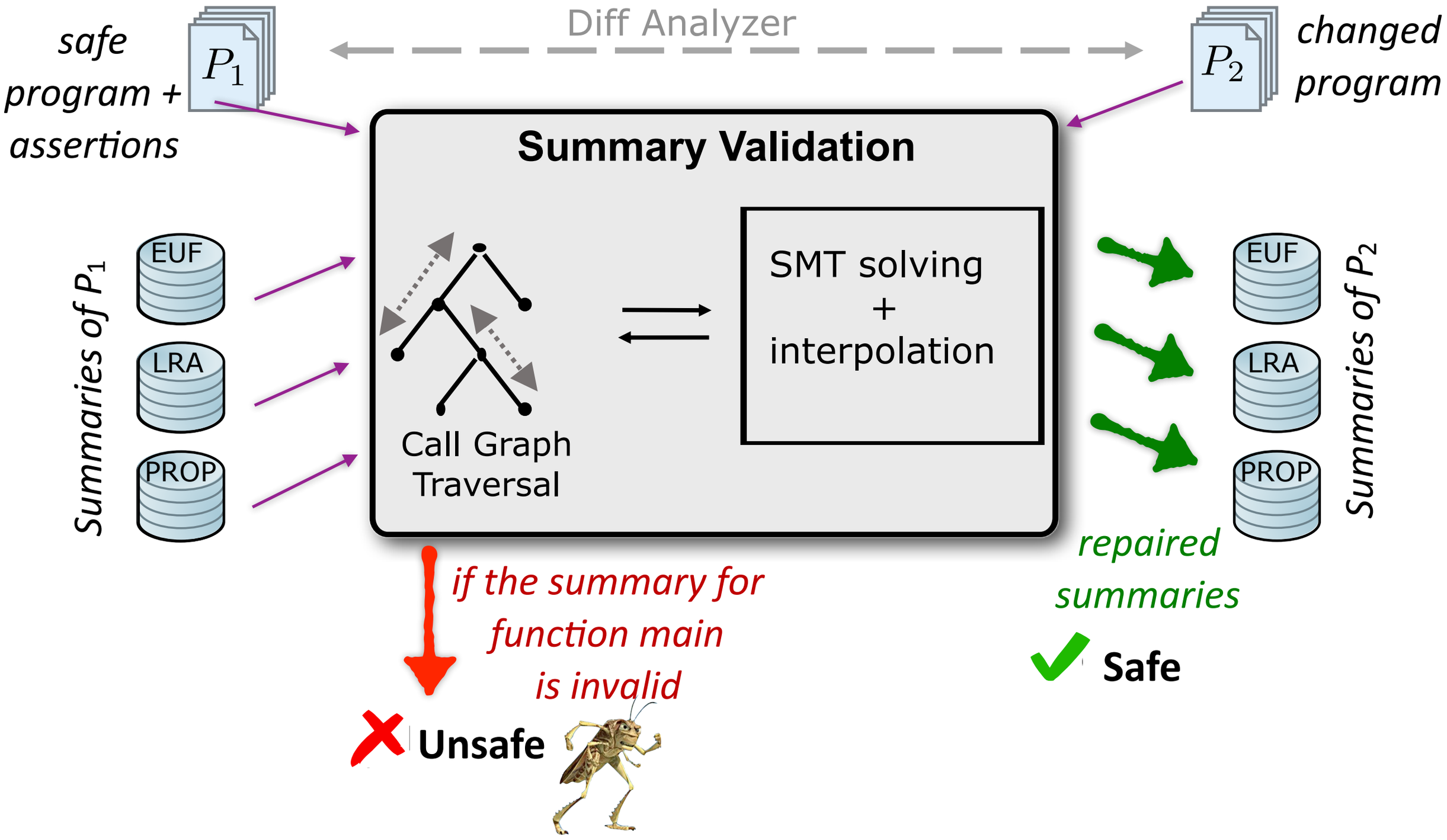
UpProver architecture



UpProver architecture



UpProver architecture



Underlying Technology of UpProver

- UpProver employs our SMT-solver OpenSMT2 for SMT checks & interpolation. [Hyvärinen et al.]
  - https://github.com/usi-verification-and-security/opensmt
- UpProver uses the CProver/CBMC framework for symbolic pre-processing of C programs and the goto-program translation. [Kroening et al.]
  - http://cprover.org
- NEW PART:
  Translation from goto-programs to various theories of SMT

Refinement

When summary validation fails:

- If over-approximative summaries in the sub-tree:
  - **Downward refinement**: Replace summaries by precise representation
  - else **Upward refinement**

When SMT formula of the root function is SAT **:**

- Increase the precision by encoding to a more-precise theory

Evaluation

C Benchmarks

1700 pairs of revisions of Linux device drivers

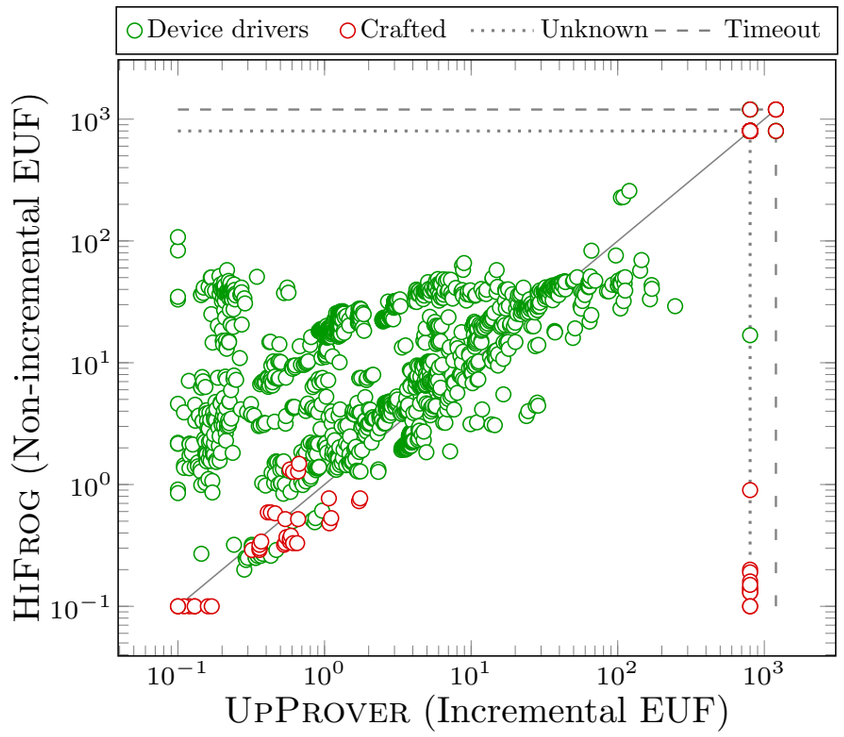
90 pairs crafted benchmarks

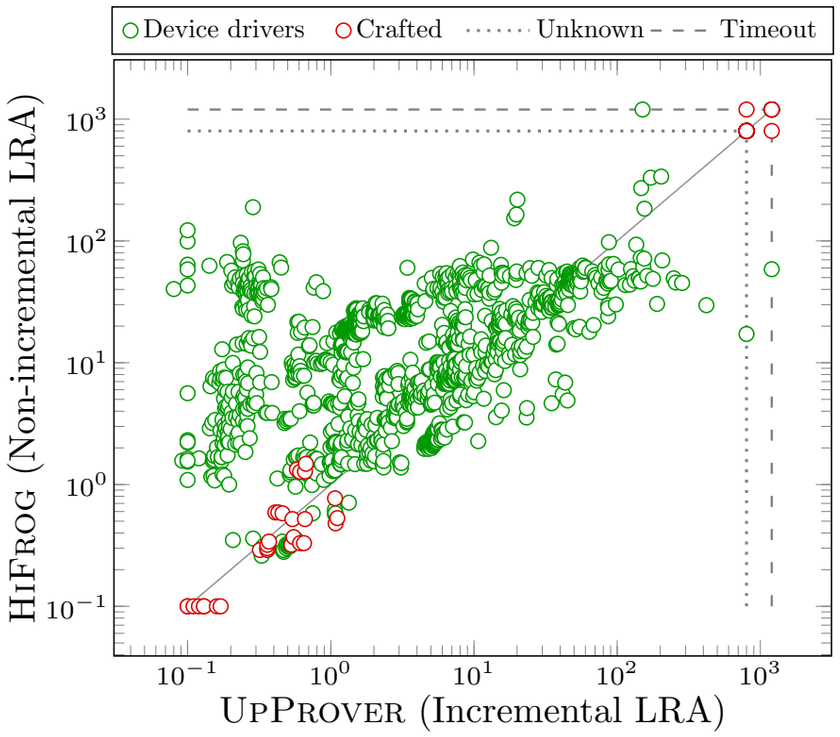
LOC average: **16K**

Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

Impact of summary reuse

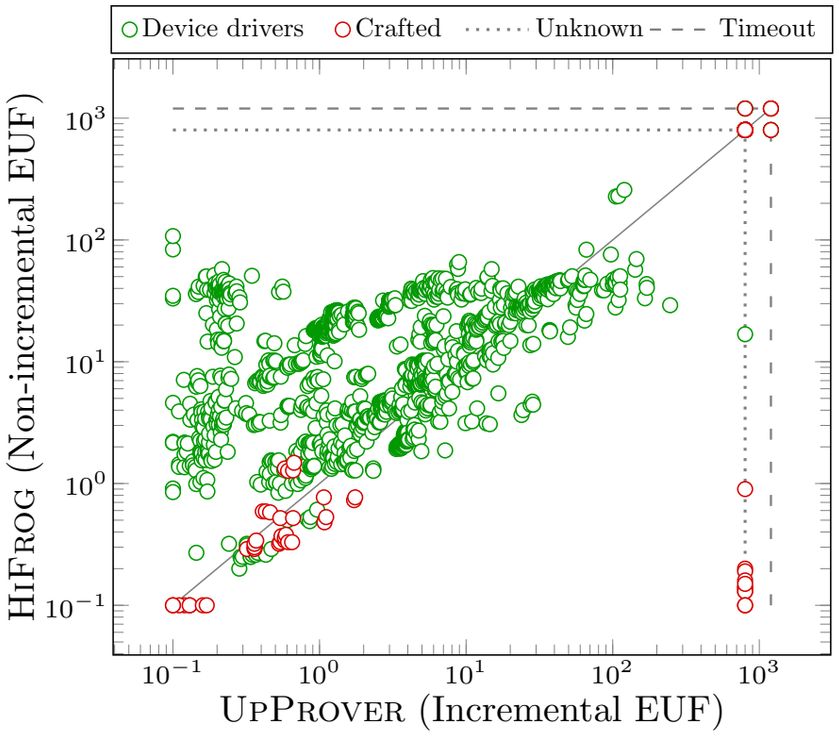


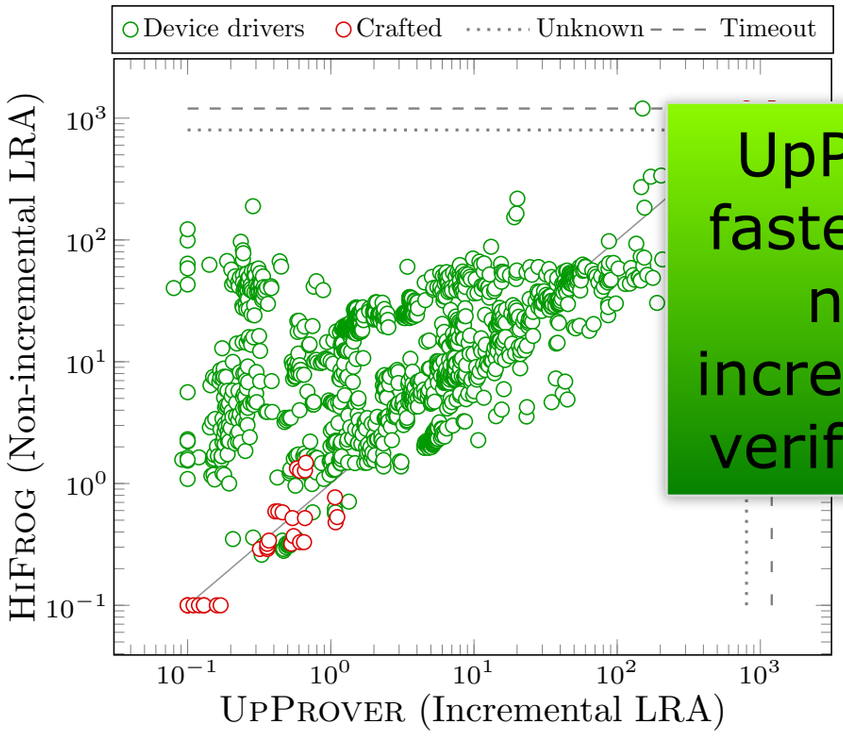


Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

Impact of summary reuse



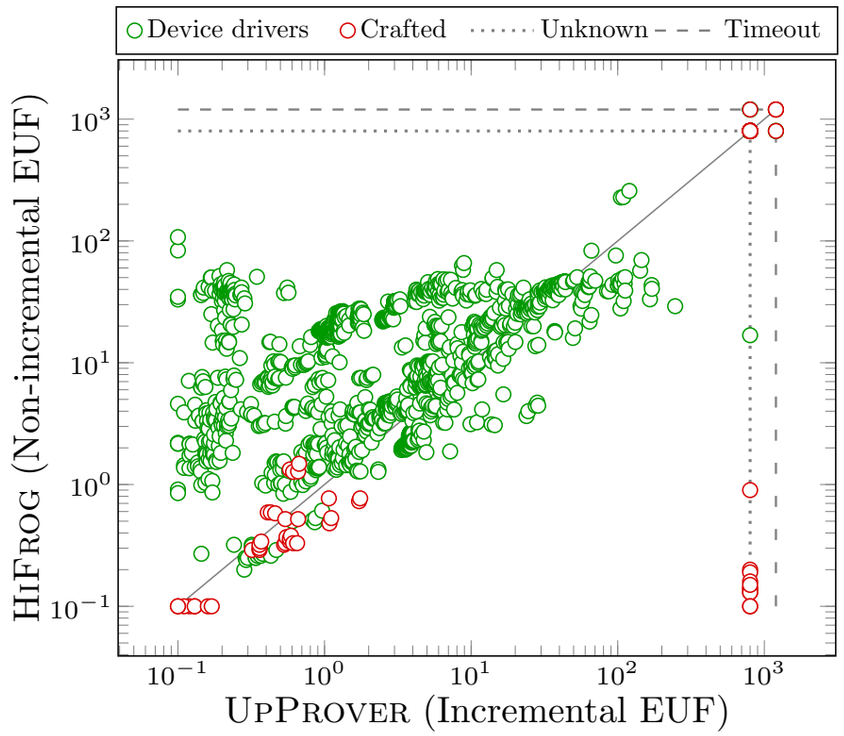


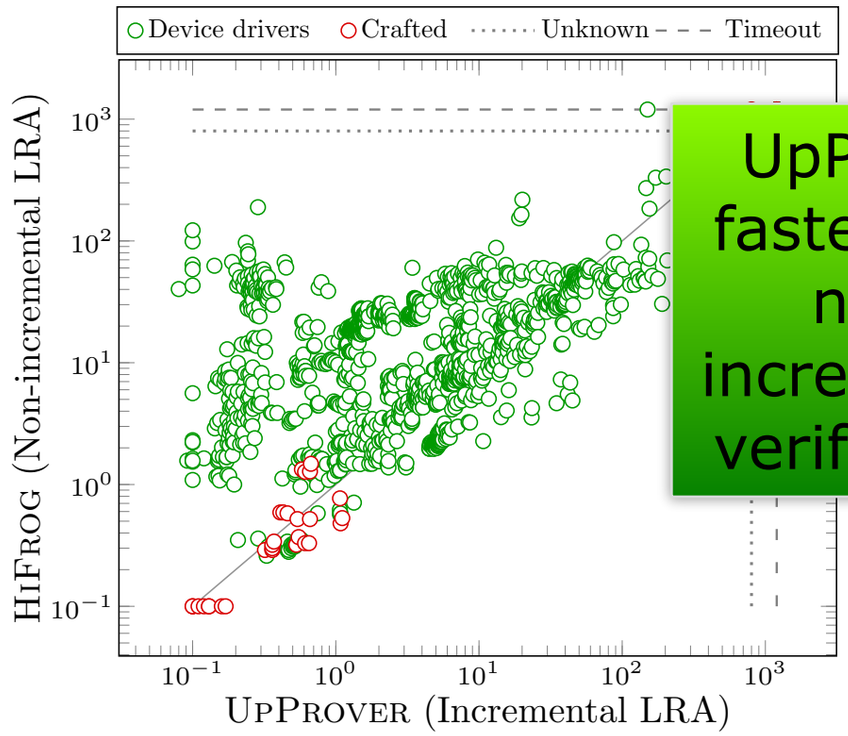
UpProver faster than non-incremental verification

Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

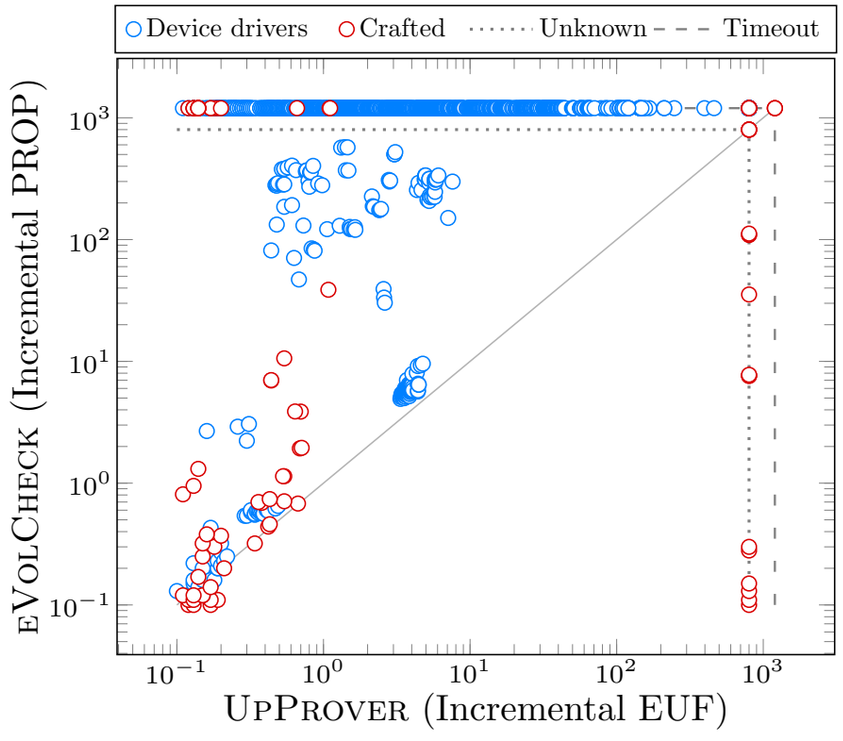
Impact of summary reuse

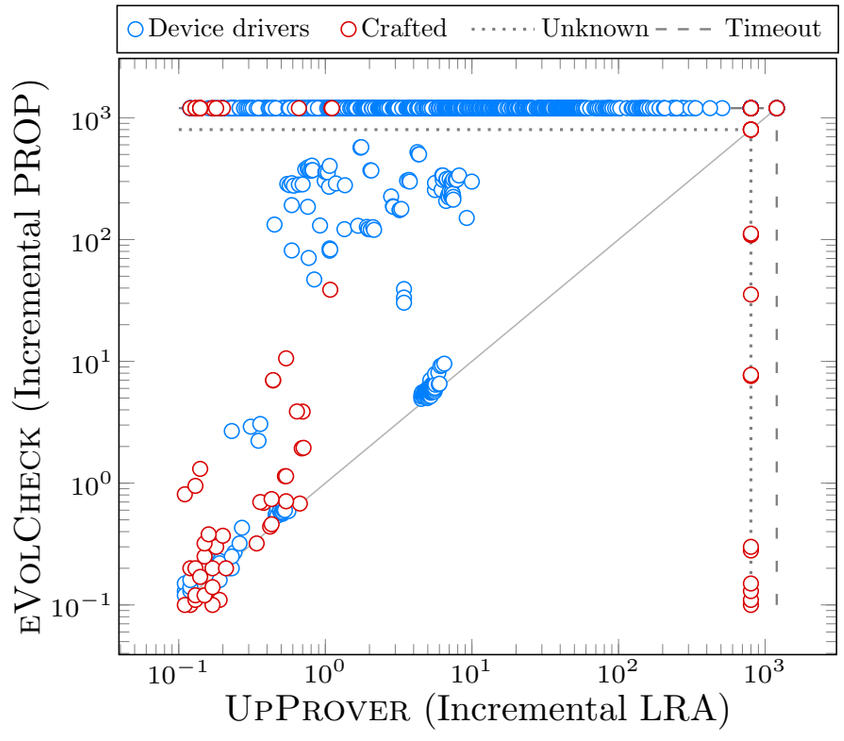




UpProver faster than non-incremental verification

Impact of theory encoding

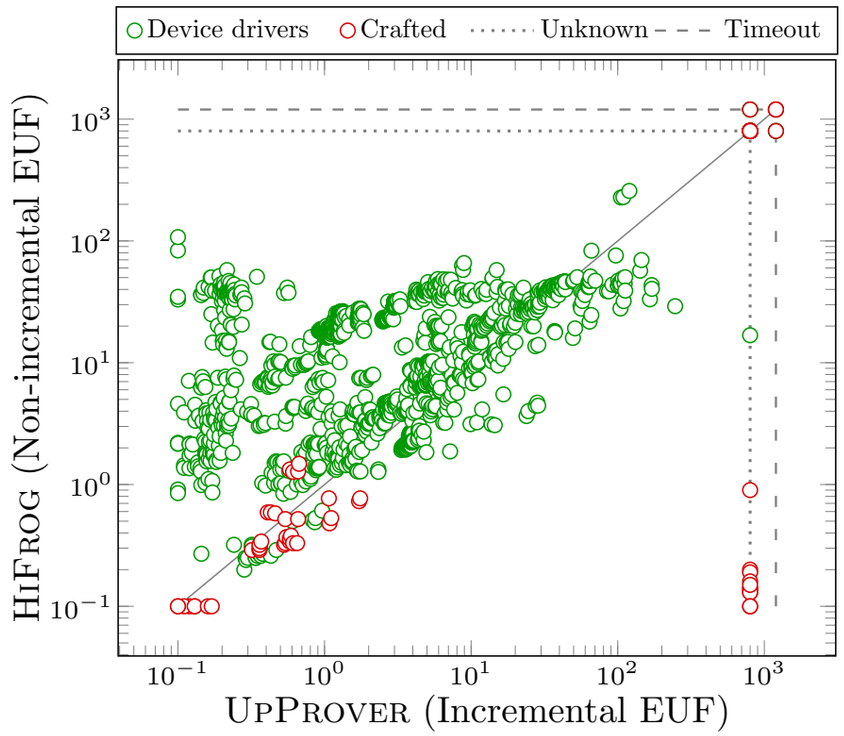


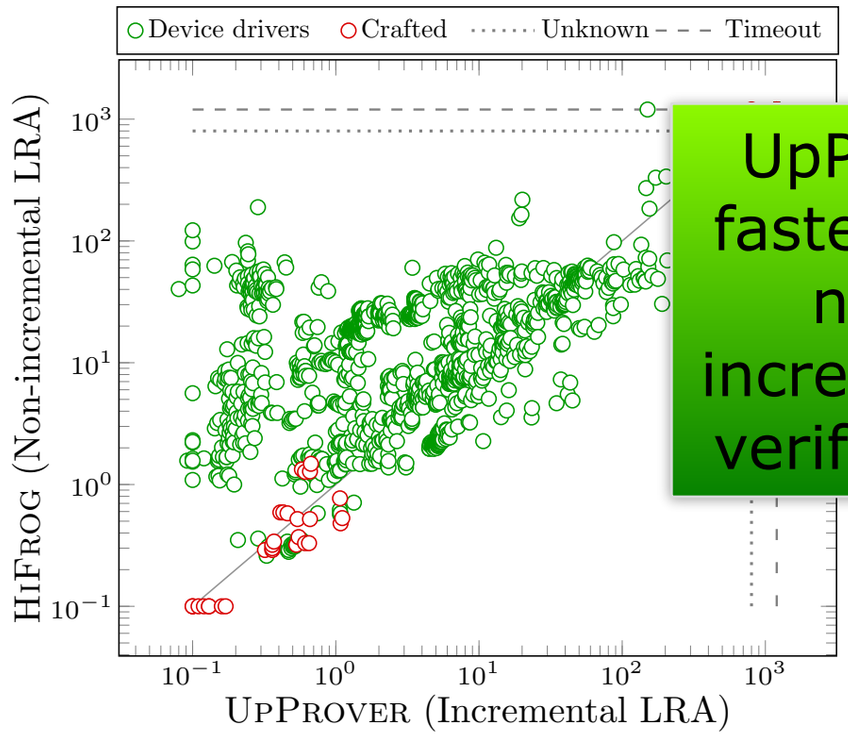


Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

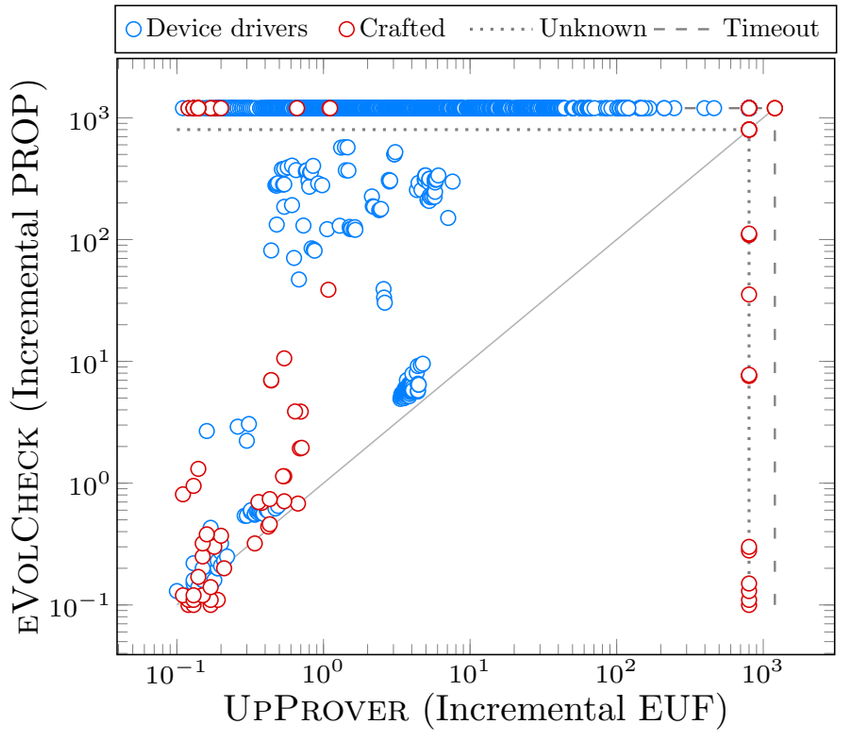
Impact of summary reuse

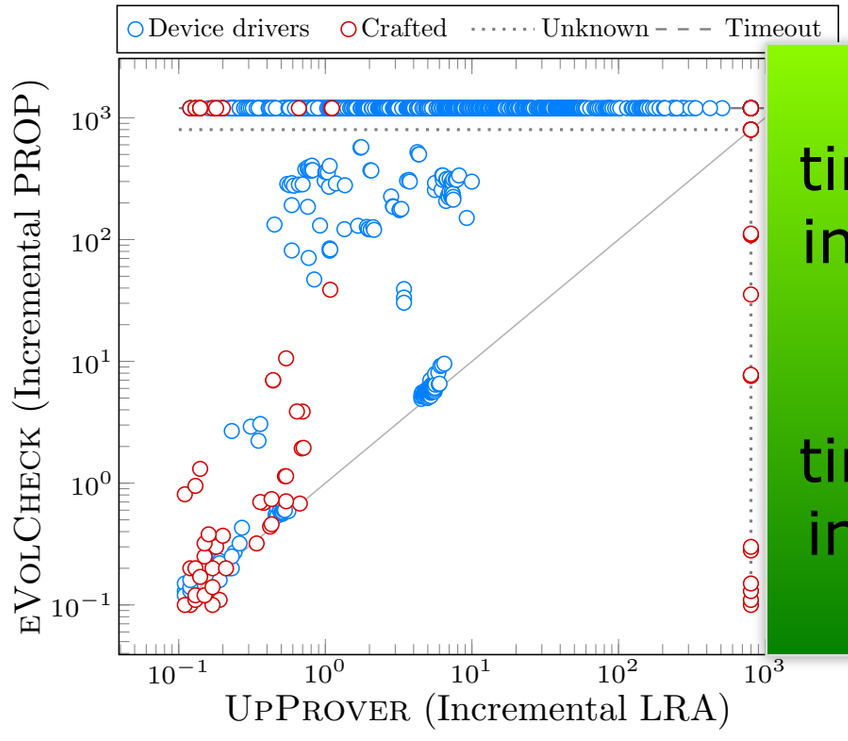




UpProver faster than non-incremental verification

Impact of theory encoding





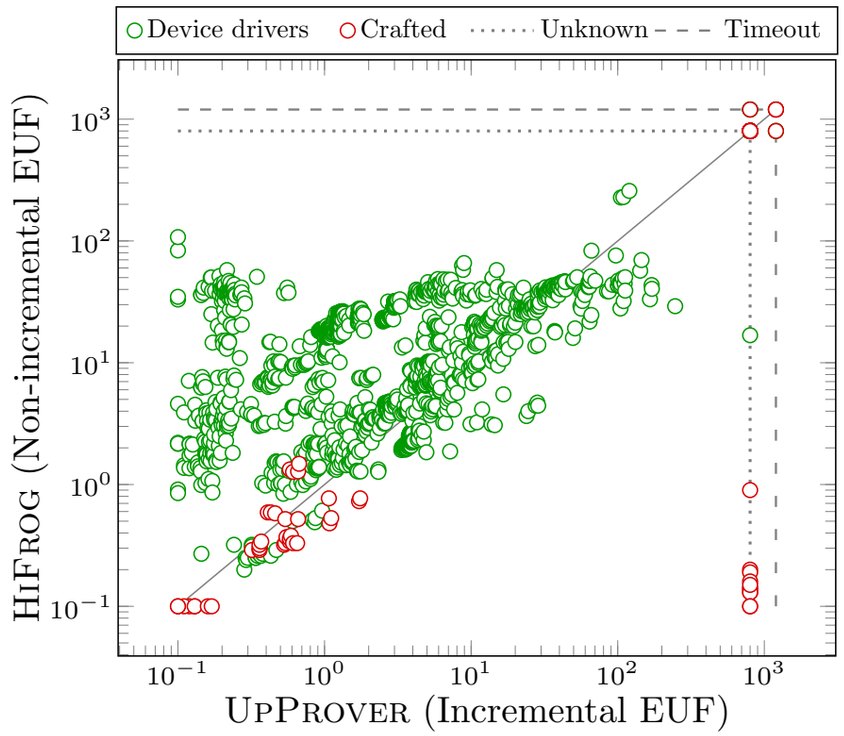
1328 timeouts in **PROP**

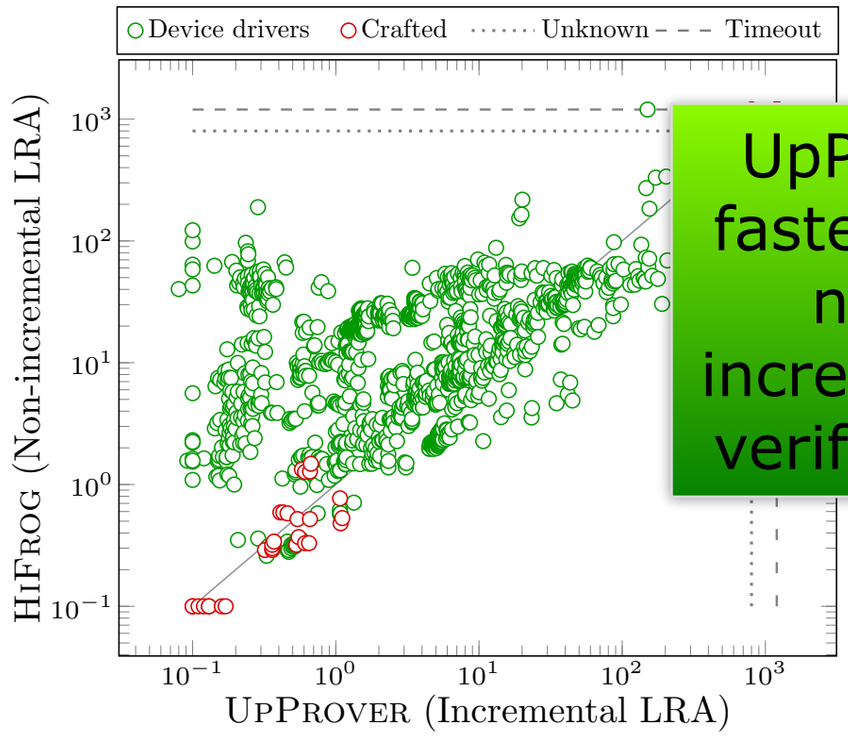
<15 timeouts in **LRA/ EUF**

Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

Impact of summary reuse



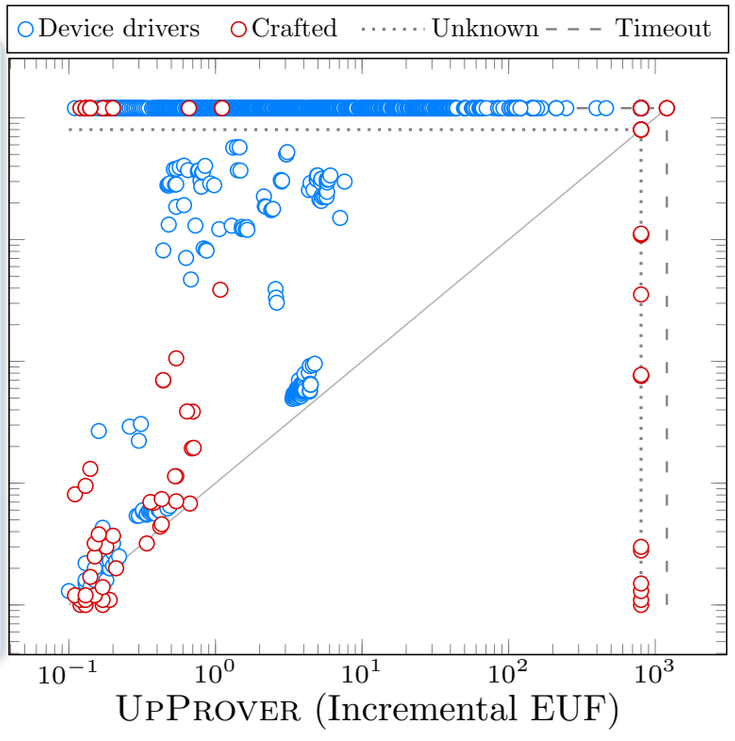


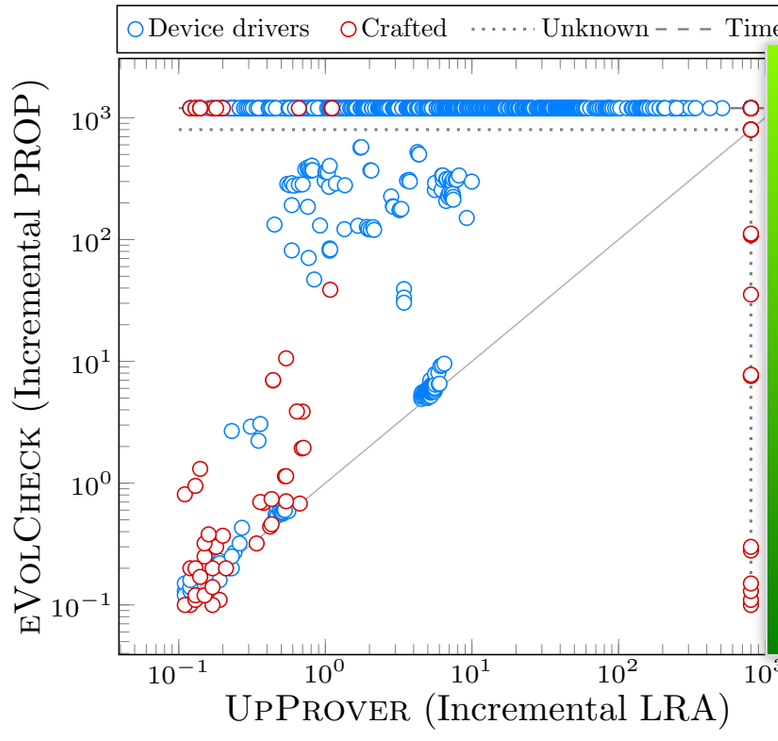
UpProver faster than non-incremental verification

Impact o theory encoding

0 unknown in **PROP**

76+86 unknown in **LRA/ EUF**





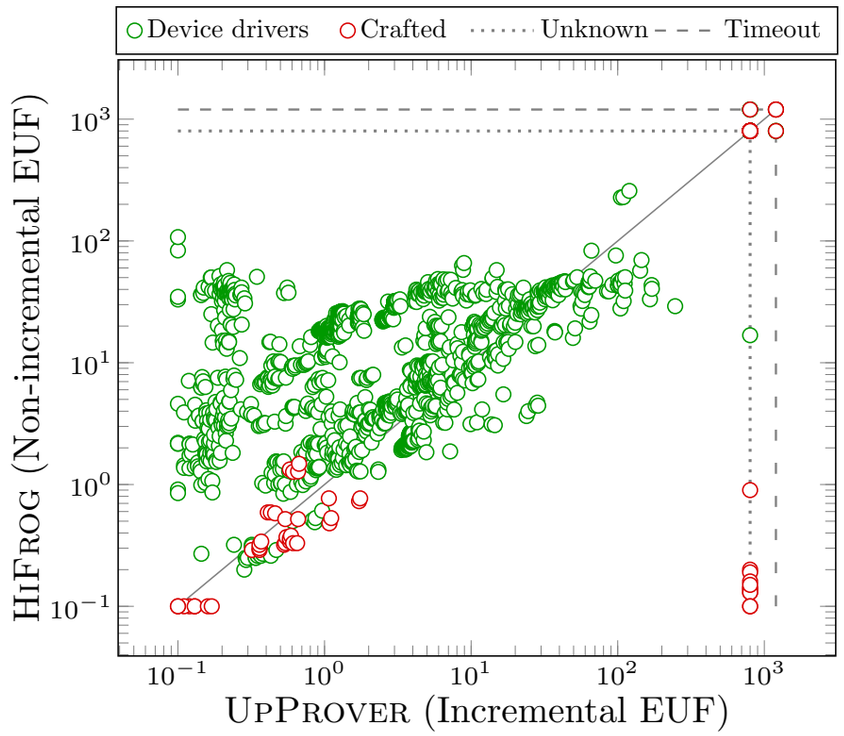
1328 timeouts in **PROP**

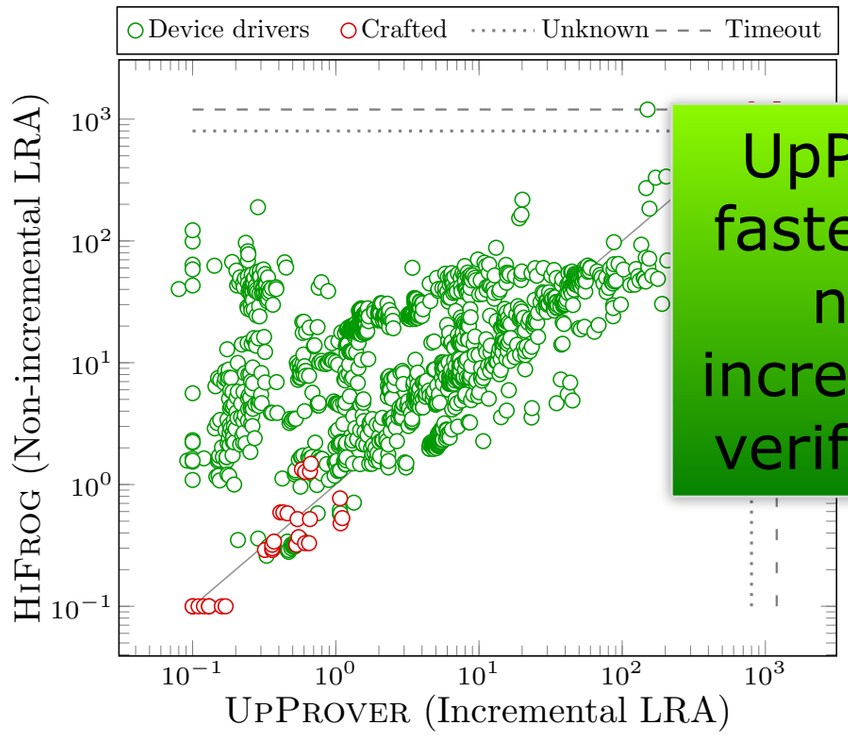
<15 timeouts in **LRA/ EUF**

Evaluation

C Benchmarks
1700 pairs of revisions of Linux device drivers
90 pairs crafted benchmarks
LOC average: **16K**

Impact of summary reuse



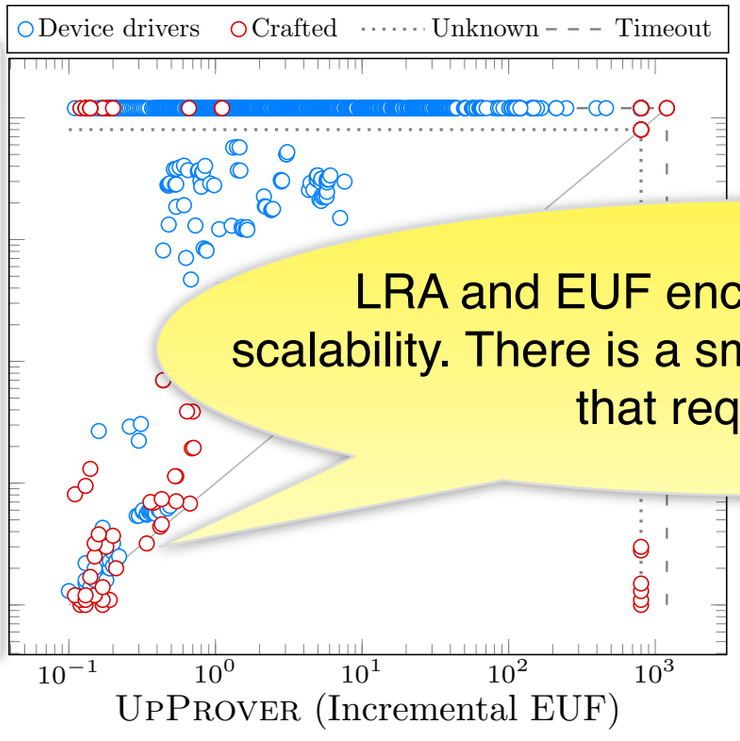


UpProver faster than non-incremental verification

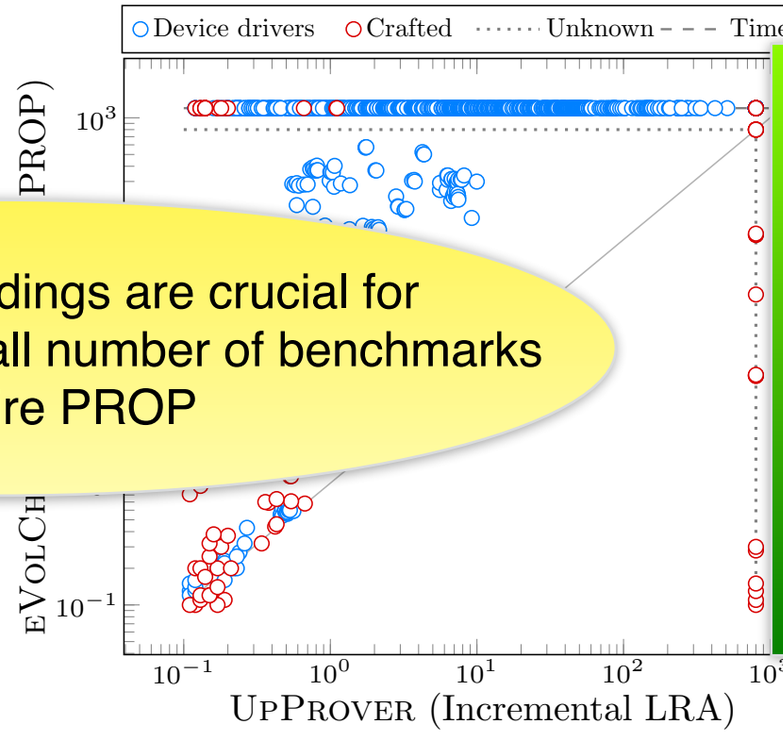
Impact o theory encoding

0 unknown in **PROP**

76+86 unknown in **LRA/ EUF**



LRA and EUF encodings are crucial for scalability. There is a small number of benchmarks that require PROP



1328 timeouts in **PROP**

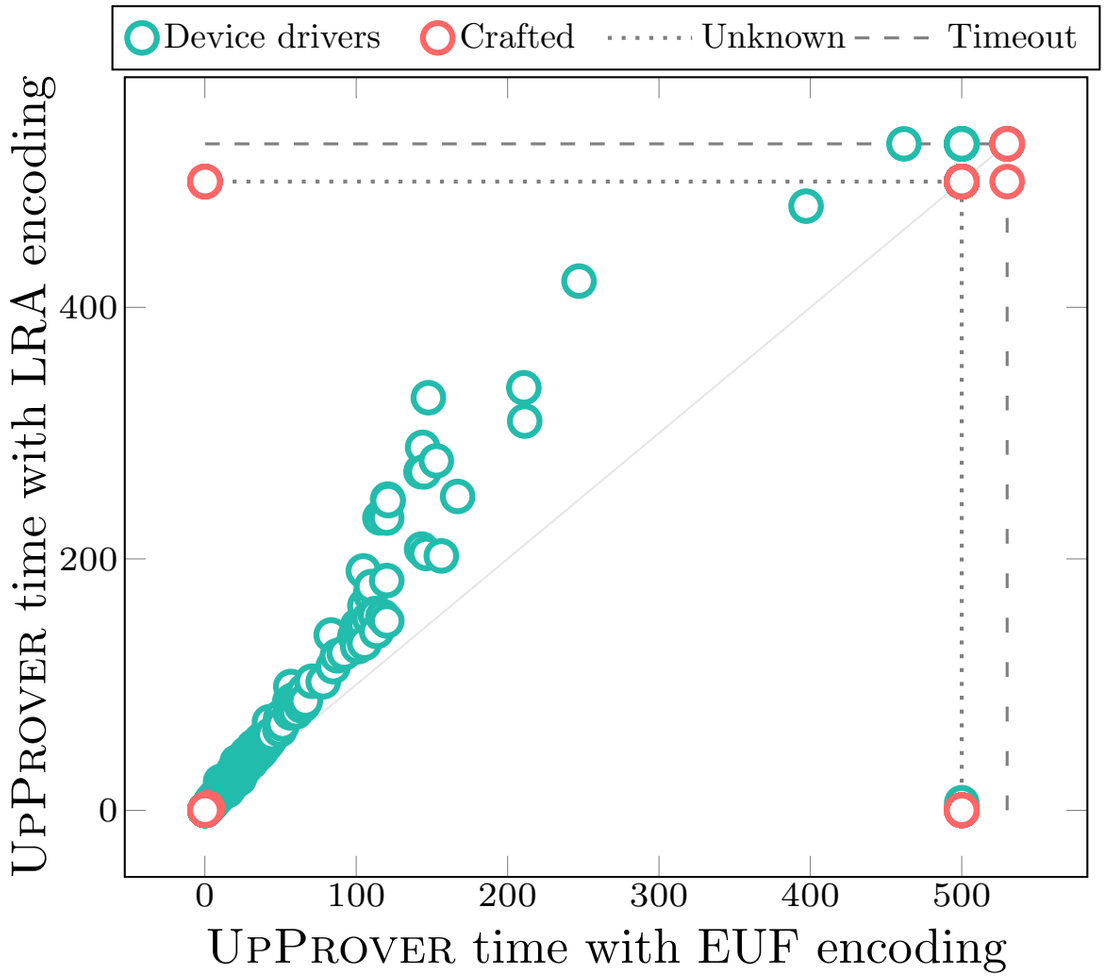
<15 timeouts in **LRA/ EUF**

LRA vs. EUF in UpProver

- LRA encoding vs. EUF has an almost 30% time overhead

- Most likely because of the more expensive Simplex algorithm in LRA compared to EUF congruence algorithm

… but LRA is more precise than EUF

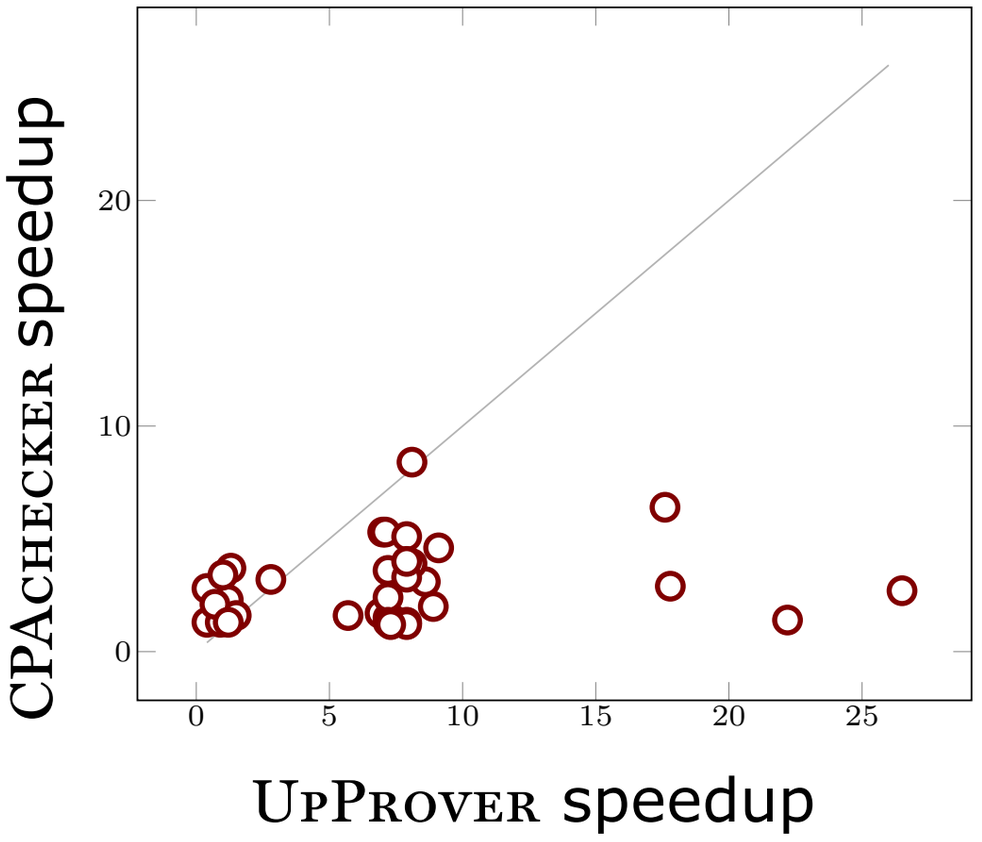


UpProver vs. CPAchecker

Summary reuse in UpProver vs precision reuse in CPAchecker

Speedup comparison for 34 categories
of device driver benchmarks with 903 revisions

Tool	Avg Speedup	Avg Slowdowns
UpProver	7.3 (30 categories)	0.6 (4 categories)
CPAchecker	2.9	no-slowdowns



Note that the approaches are orthogonal (BMC vs loop invariants), thus the comparison is not very accurate

CPAchecker: “Precision reuse for efficient regression verification”, Beyer et. al 2013

Related Work

- **eVolCheck:** Incremental Upgrade Checker for C [Fedyukovich et. al TACAS'13]
- **Niagara:** Incremental CHC solver [Fedyukovich et. al. NFM'14, CAV'16]
- **HiFrog**: SMT-based function summarization for software verification [Alt et. al TACAS'17]
- **CPAchecker:** Precision reuse for efficient regression verification [Beyer et. al 2013]
- **ModDiff** (based on CProver): Modular Demand-Driven Analysis of Semantic Difference for Program Versions [Trostanetski et. al 2017]

Future Work

- Extend the tool to handle summaries across different theories, possibly by allowing checks for the tree-interpolation property on-the-fly

Conclusion

Conclusion

- UpProver: A BMC tool for incremental verification of program versions

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)
- Leverages corresponding SMT-level summarization algorithms

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)
- Leverages corresponding SMT-level summarization algorithms
- Allows the user to adjust the **precision** or **efficiency** of verification

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)
- Leverages corresponding SMT-level summarization algorithms
- Allows the user to adjust the **precision** or **efficiency** of verification
- Repairs the previously computed summaries on-the-fly

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)
- Leverages corresponding SMT-level summarization algorithms
- Allows the user to adjust the **precision** or **efficiency** of verification
- Repairs the previously computed summaries on-the-fly
- Scalable technique for large program versions with multiple properties

Conclusion

- UpProver: A BMC tool for incremental verification of program versions
- Focuses on the modified code & cheap local check

- Offers different SMT-level encodings (EUF, LRA, PROP)
- Leverages corresponding SMT-level summarization algorithms
- Allows the user to adjust the **precision** or **efficiency** of verification
- Repairs the previously computed summaries on-the-fly
- Scalable technique for large program versions with multiple properties

UpProver http://verify.inf.usi.ch/upprover