An Overview of Automated Program Analysis

Raphaël Monat – SyCoMoRES team

`rmonat.fr`

CRIStAL’s CyberSecurity Seminar
21 Oct. 2024

Introduction

whoami

Research Scientist at Inria since Sep. 2022.

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

- Static analysis: C, Python, multi-language paradigms

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations
  Automated Verification of Catala Programs

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations

  Automated Verification of Catala Programs

Other Research Interests in SyCoMoRES

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations
  Automated Verification of Catala Programs

Other Research Interests in SyCoMoRES

- Scheduling for real-time embedded systems

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations
  Automated Verification of Catala Programs

Other Research Interests in SyCoMoRES

- Scheduling for real-time embedded systems
- Binary code analysis [Bal+19] (for worst-case execution time, security)

whoami

Research Scientist at Inria since Sep. 2022.

Research Interests

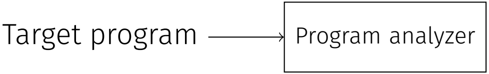
- Static analysis: C, Python, multi-language paradigms
- Formal methods for public administrations
  Automated Verification of Catala Programs

Other Research Interests in SyCoMoRES

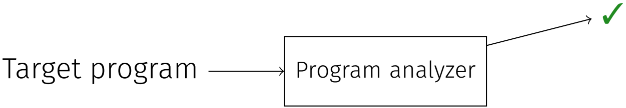
- Scheduling for real-time embedded systems
- Binary code analysis [Bal+19] (for worst-case execution time, security)
- Type systems for privacy

Target program

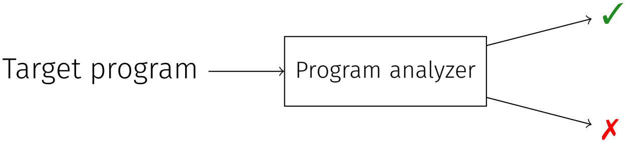
Automated Program Analysis



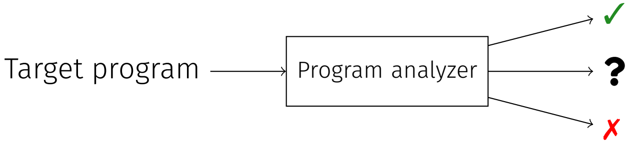
Automated Program Analysis



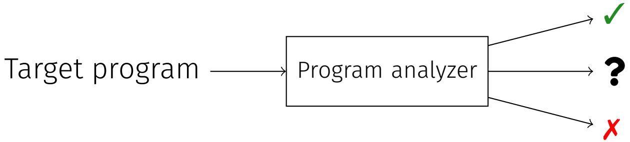
Automated Program Analysis



Automated Program Analysis



Automated Program Analysis



Motivation

Sheer quantity of programs and changes during their life:

Manual processes (e.g. testing, manual verification) will not scale!

A Program Analysis Recipe

Target property φ

A Program Analysis Recipe

Target property φ

- Absence of runtime errors

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

Input format i

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

Input format i

- Source code

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

Input format i

- Source code
- Binary executable

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

Input format i

- Source code
- Binary executable

Requirement: <u>semantics of the program representation</u>

A Program Analysis Recipe

Target property φ

- Absence of runtime errors
- Constant-time execution
- Endianness portability [DOM21]

Benchmarks

- Open-source
- Real-world

Input format i

- Source code
- Binary executable

Requirement: semantics of the program representation

$\Longrightarrow$ now build $\texttt{Analyzer}_{\varphi}(\text{prog} : i)$

Turing & Rice to the Rescue (or not?)

Sound	All errors in program reported by analyzer

Turing & Rice to the Rescue (or not?)

All errors reported by analyzer are replicable in program	Complete	Sound	All errors in program reported by analyzer

Turing & Rice to the Rescue (or not?)

Guaranteed Termination

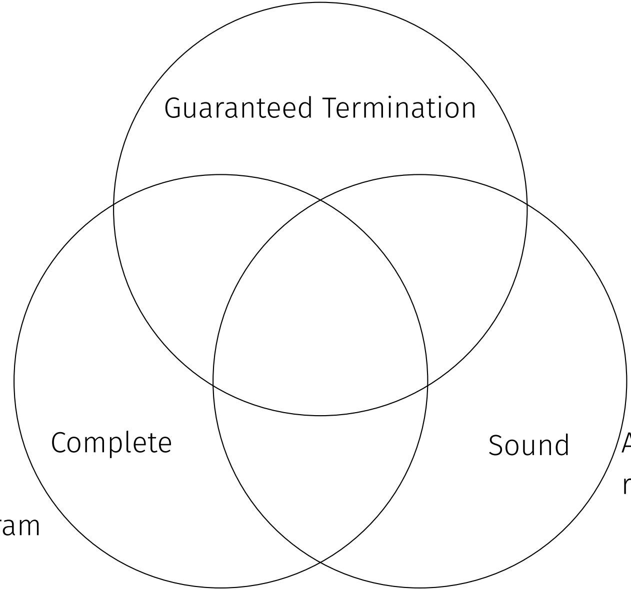
All errors reported by analyzer are replicable in program

Complete

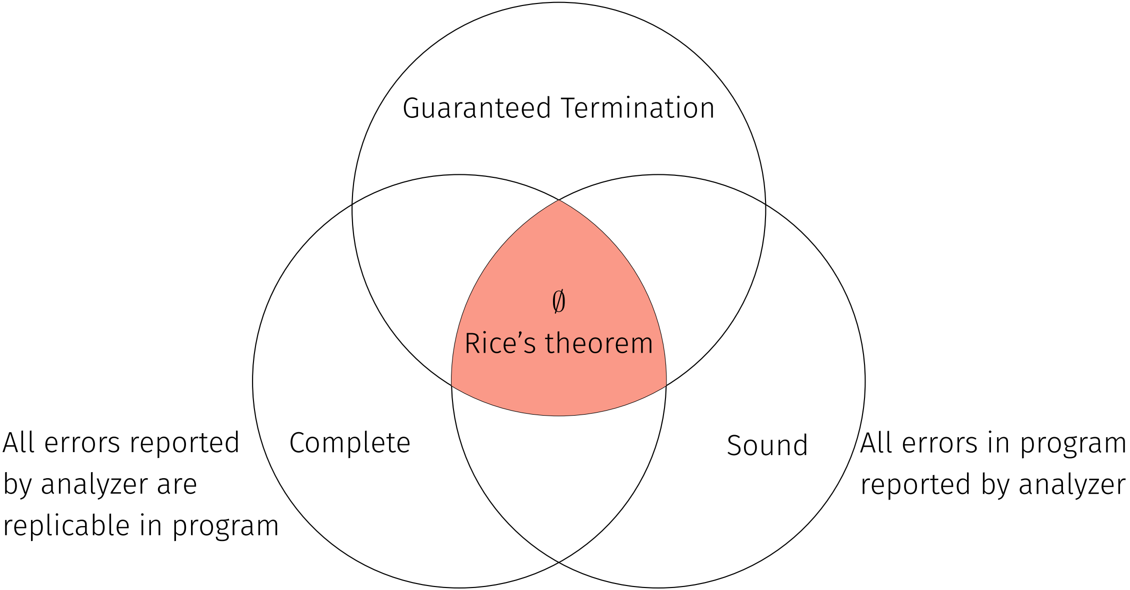
Sound

All errors in program reported by analyzer

Turing & Rice to the Rescue (or not?)



Turing & Rice to the Rescue (or not?)



Outline

Overview of Program Analysis Techniques

Symbolic Execution

Core idea: systematic generation of testcases

Core idea: systematic generation of testcases

Explore all program paths

Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints

Symbolic Execution

Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```

Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```

```
if x > 0:
```

Symbolic Execution

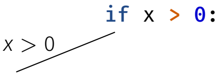
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



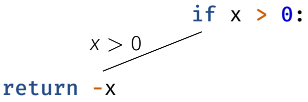
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



Symbolic Execution

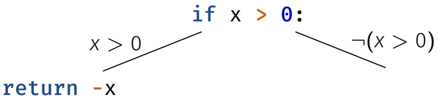
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



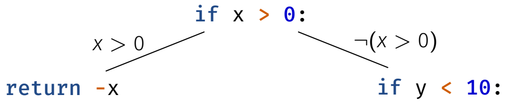
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



Symbolic Execution

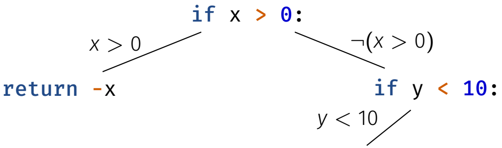
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



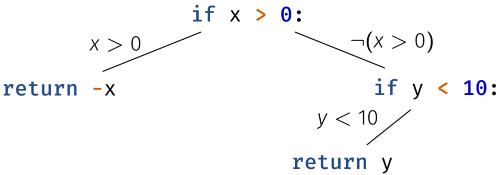
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```python
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



Symbolic Execution

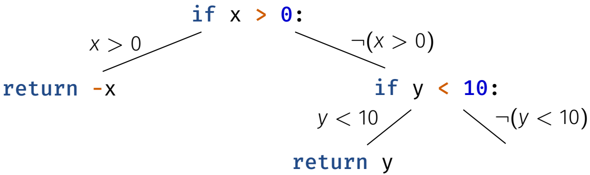
Core idea: systematic generation of testcases

Explore all program paths

- Collect path constraints
- Rely on constraint solvers to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



Symbolic Execution

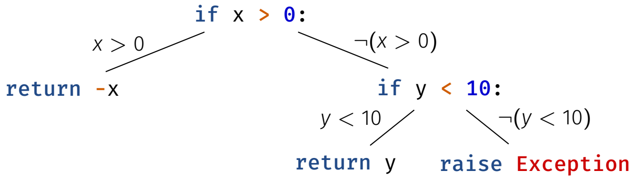
Core idea: systematic generation of testcases

Explore all <u>program paths</u>

- Collect <u>path constraints</u>
- Rely on <u>constraint solvers</u> to generate testcases

Toy example

```
if x > 0:
  return -x
else:
  if y < 10:
    return y
  else:
    raise Exception
```



Symbolic Execution (II)

Risk: combinatorial explosion (loops, . . .)

Symbolic Execution (II)

Risk: combinatorial explosion (loops, . . .)

Further references

- KLEE [CDE08]

Symbolic Execution (II)

Risk: combinatorial explosion (loops, . . .)

Further references

- KLEE [CDE08]
- Symbolic execution survey [Bal+18]

Symbolic Execution (II)

Risk: combinatorial explosion (loops, . . .)

Further references

- KLEE [CDE08]
- Symbolic execution survey [Bal+18]
- Concrete + symbolic = concolic execution [SMA05; GKS05]

Symbolic Execution (II)

Risk: combinatorial explosion (loops, . . .)

Further references

- KLEE [CDE08]
- Symbolic execution survey [Bal+18]
- Concrete + symbolic = concolic execution [SMA05; GKS05]
- Constraint solvers are currently SMT solvers: Z3 [MB08], CVC5 [Bar+22], Alt-Ergo [Con+18], SMT-LIB interface [BFT16]

Overview of Program Analysis Techniques

Fuzzing

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, . . .) $\Longrightarrow$ you may be on to something!

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, . . .) $\Longrightarrow$ you may be on to something!

Various shades of fuzzing

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, . . .) $\Longrightarrow$ you may be on to something!

Various shades of fuzzing

- Black-box: generates new inputs either by

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, ...) $\implies$ you may be on to something!

Various shades of fuzzing

- Black-box: generates new inputs either by
  - mutating input samples (mutational)

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, . . .) $\implies$ you may be on to something!

Various shades of fuzzing

- Black-box: generates new inputs either by
  - mutating input samples (mutational)
  - relying on an input grammar (generational)

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, ...) $\implies$ you may be on to something!

Various shades of fuzzing

- Black-box: generates new inputs either by
  - mutating input samples (mutational)
  - relying on an input grammar (generational)
- Gray-box: rely on instrumented coverage to direct fuzzing (cf. AFL++, LibFuzzer)

Fuzzing

Core idea: throw random stuff at programs

```
cat /dev/random | ./target-program
```

Crash (segmentation fault, . . .) $\implies$ you may be on to something!

Various shades of fuzzing

- Black-box: generates new inputs either by
  - mutating input samples (mutational)
  - relying on an input grammar (generational)
- Gray-box: rely on instrumented coverage to direct fuzzing (cf. AFL++, LibFuzzer)
- White-box = symbolic execution

Fuzzing (II)

Current state

- Popular: easy to set basic version up

Fuzzing (II)

Current state

- Popular: easy to set basic version up
- Difficulty: correct instrumentation/directing of fuzzers

Fuzzing (II)

Current state

- Popular: easy to set basic version up
- Difficulty: correct instrumentation/directing of fuzzers
- May use lots of resources

Fuzzing (II)

Current state

- Popular: easy to set basic version up
- Difficulty: correct instrumentation/directing of fuzzers
- May use lots of resources
- Sanitizers can be added to detect more bugs

Fuzzing (II)

Current state

- Popular: easy to set basic version up
- Difficulty: correct instrumentation/directing of fuzzers
- May use lots of resources
- Sanitizers can be added to detect more bugs
- Google's OSS-FUZZ infrastructure

Overview of Program Analysis Techniques

Abstract Interpretation

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true
  - Usually cannot prove programs incorrect

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true
  - Usually cannot prove programs incorrect
- Traditionally used for certification

Abstract Interpretation

- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true
  - Usually cannot prove programs incorrect
- Traditionally used for certification
  - Airbus A380/A340 control commands with Astrée [Ber+10]

Abstract Interpretation

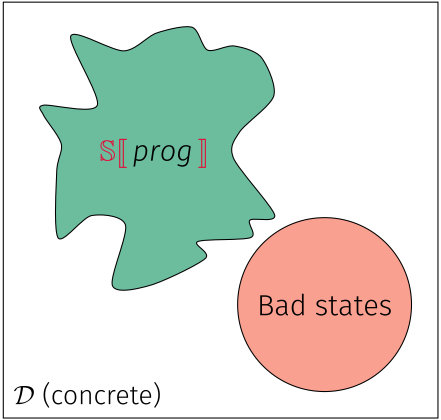
- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true
  - Usually cannot prove programs incorrect
- Traditionally used for certification
  - Airbus A380/A340 control commands with Astrée [Ber+10]
  - Nuclear power plants with Frama-C [BBY17]

Abstract Interpretation

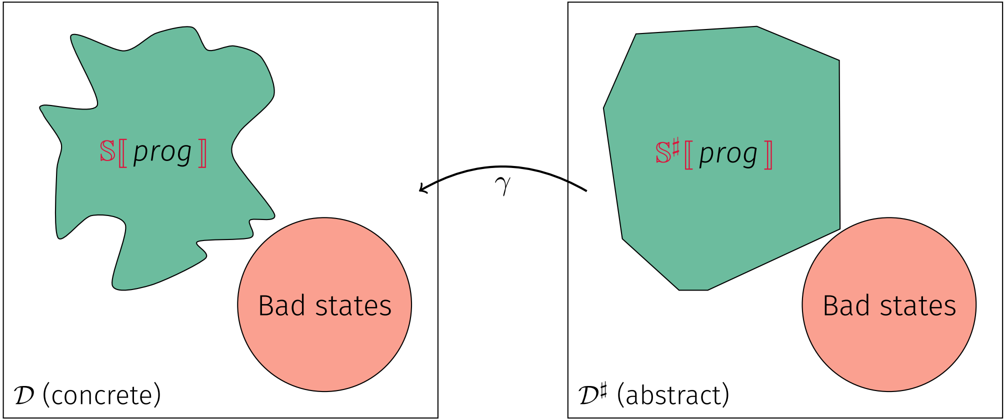
- Approximate analysis, but ensure soundness and termination
- Invented by Patrick and Radhia Cousot in the late 70s.
- Analysis tries to prove program correct.
  - Alarms: deciding which ones are true
  - Usually cannot prove programs incorrect
- Traditionally used for certification
  - Airbus A380/A340 control commands with Astrée [Ber+10]
  - Nuclear power plants with Frama-C [BBY17]
- Suggested entry-point: Miné [Min17]

Core Ideas behind Abstract Interpretation

Approximating a Complex World, with Guarantees



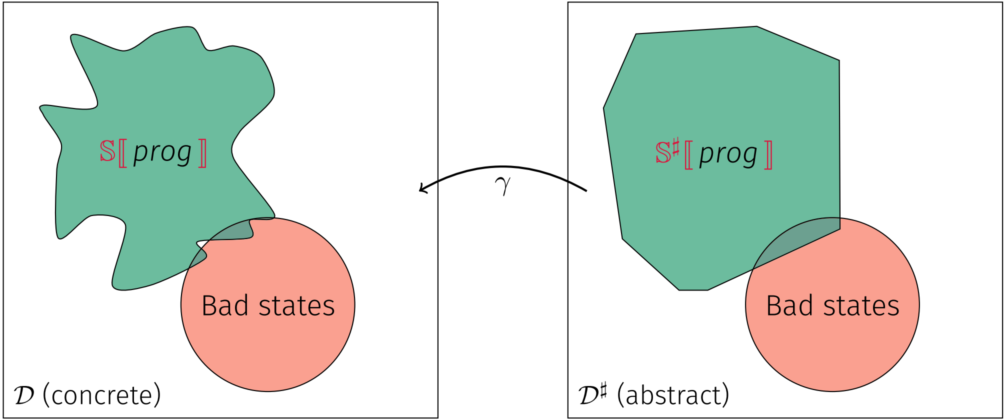
Approximating a Complex World, with Guarantees



Interpret in non-standard domain

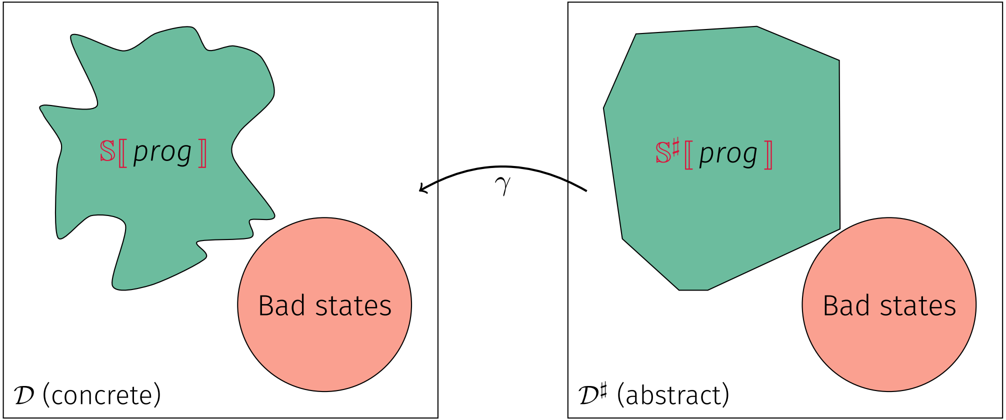
Program proved safe

Approximating a Complex World, with Guarantees

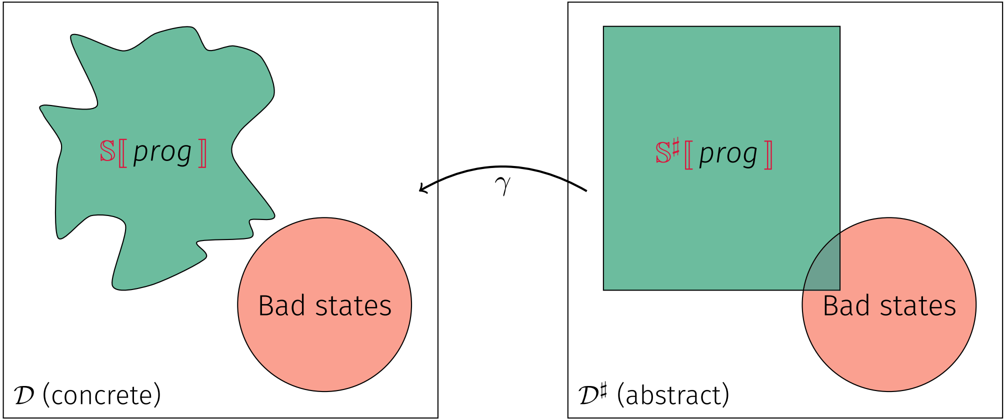


True alarm

Approximating a Complex World, with Guarantees

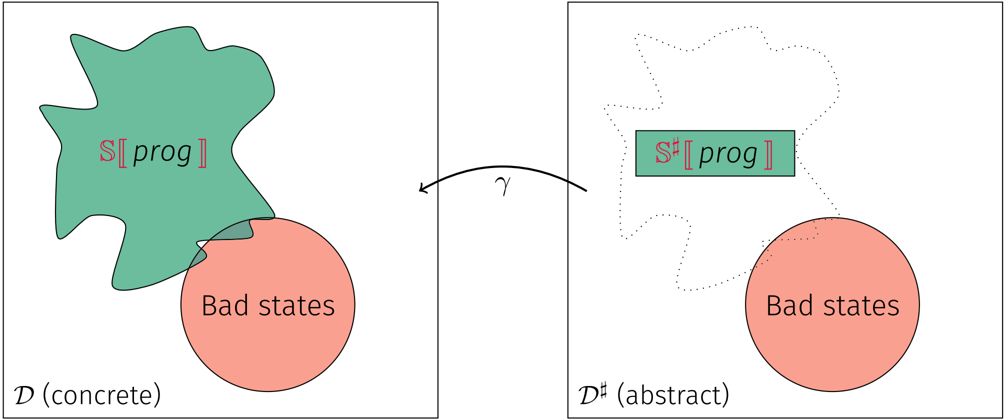


Approximating a Complex World, with Guarantees



False alarm (Abstraction too coarse)

Approximating a Complex World, with Guarantees



Unsound analysis
(shouldn't happen)

Merging Everything to Scale

Merging States

```
int x = rand();
```

Merging States

```
int x = rand();
```

- Concrete World

Merging States

```
int x = rand();
```

- Concrete World
  - Set of program states $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$

Merging States

```
int x = rand();
```

- Concrete World
  - Set of program states $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$
  - $\Sigma = \{ x \mapsto n \mid 0 \leq n < 2^{31} \}$

Merging Everything to Scale

Merging States

```
int x = rand();
```

- Concrete World
  - Set of program states $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$
  - $\Sigma = \{ x \mapsto n \mid 0 \leq n < 2^{31} \}$
- Abstract World

Merging Everything to Scale

Merging States

```
int x = rand();
```

- Concrete World
  - Set of program states $\mathcal{P}(\mathcal{V} \rightarrow \mathbb{Z})$
  - $\Sigma = \{\, x \mapsto n \mid 0 \leq n < 2^{31} \,\}$
- Abstract World
  - Represent multiple concrete states at once $\mathcal{V} \rightarrow \texttt{Intervals}$

Merging States

```
int x = rand();
```

- Concrete World
  - Set of program states $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$
  - $\Sigma = \{ x \mapsto n \mid 0 \leq n < 2^{31} \}$
- Abstract World
  - Represent multiple concrete states at once $\mathcal{V} \to \texttt{Intervals}$
  - $\sigma^\sharp = x \mapsto [0, 2147483647]$

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Precision tradeoffs

⚠ computing an over-approximation, potential imprecision

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Precision tradeoffs

⚠ computing an over-approximation, potential imprecision

- Concrete $\Sigma = \{\, x \mapsto n \mid -1 \leq n \leq 11 \wedge n \neq 10 \,\}$

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Precision tradeoffs

⚠ computing an over-approximation, potential imprecision

- Concrete $\Sigma = \{ x \mapsto n \mid -1 \leq n \leq 11 \land n \neq 10 \}$
- Abstract $\sigma^\sharp = x \mapsto [-1, 11]$

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Precision tradeoffs

⚠ computing an over-approximation, potential imprecision

- Concrete $\Sigma = \{ x \mapsto n \mid -1 \leq n \leq 11 \land n \neq 10 \}$
- Abstract $\sigma^\sharp = x \mapsto [-1, 11]$

$\implies$ may require better abstractions!

Merging Everything to Scale (II)

Merging Paths

```
int x = rand(); if(x > 10) { x = 11; } else { x--; }; print(x);
```

- Contrary to symbolic execution, merge paths
- Rely on least upper bound operator ($\sqcup$) and lattice structure

Precision tradeoffs

⚠ computing an over-approximation, potential imprecision

- Concrete $\Sigma = \{\, x \mapsto n \mid -1 \leq n \leq 11 \wedge n \neq 10 \,\}$
- Abstract $\sigma^\sharp = x \mapsto [-1, 11]$

$\Longrightarrow$ may require better abstractions!

Merging can also be applied to arrays, ...

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
…	…
99	[0, 99]
100	[0, 99]

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!

Widening – Generalization Operator

```c
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Introduce generalization operator ∇

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!

! large nb(iterations)

Introduce generalization operator ∇

- Over-approximating least upper bound $\sqcup$

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of **i** in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Introduce generalization operator ∇

- Over-approximating least upper bound $\sqcup$
- Ensures finite termination of loop iterations

Widening – Generalization Operator

```c
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Introduce generalization operator ∇

- Over-approximating least upper bound $\sqcup$
- Ensures finite termination of loop iterations
- nb(iterations) does not depend on loop bound

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Introduce generalization operator ∇

- Over-approximating least upper bound $\sqcup$
- Ensures finite termination of loop iterations
- nb(iterations) does not depend on loop bound

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
2	$[0, 0] \nabla [0, 1] = [0, +\infty]$
3	$[0, +\infty]$

Widening – Generalization Operator

```
int i = 0;
while(i < 100) {
  i++;
}
```

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
...	...
99	[0, 99]
100	[0, 99]

- Stabilization reached!
- ! large nb(iterations)

Introduce generalization operator ∇

- Over-approximating least upper bound $\sqcup$
- Ensures finite termination of loop iterations
- nb(iterations) does not depend on loop bound

Iteration	Values of `i` in loop
0	[0, 0]
1	[0, 1]
2	$[0, 0] \nabla [0, 1] = [0, +\infty]$
3	$[0, +\infty]$

Precision can be recovered through decreasing iterations

$\Longrightarrow$ $\mathtt{i} = [0, 99]$

A Modern Abstract Interpreter: Mopsa

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals

- Explore new designs

  Including multi-language support

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals

- Explore new designs
  - Including multi-language support
- Ease development of relational static analyses
  - High expressivity

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals

- Explore new designs
  - Including multi-language support
- Ease development of relational static analyses
  - High expressivity
- Open-source (LGPL)

Modular **O**pen **P**latform for **S**tatic **A**nalysis [Jou+19]

`gitlab.com/mopsa/mopsa-analyzer`

Goals

- Explore new designs
  - Including multi-language support
- Ease development of relational static analyses
  - High expressivity
- Open-source (LGPL)
- Can be used as an experimentation platform

Contributors (2018–2024, chronological arrival order)

- A. Miné
- A. Ouadjaout
- M. Journault
- A. Fromherz
- D. Delmas
- R. Monat
- G. Bau
- F. Parolini
- M. Milanese
- M. Valnet
- J. Boillot

Contributors (2018–2024, chronological arrival order)

- **A. Miné**
- **A. Ouadjaout**
- M. Journault
- A. Fromherz
- D. Delmas
- **R. Monat**
- G. Bau
- F. Parolini
- M. Milanese
- M. Valnet
- J. Boillot

Maintainers in bold.

Software-Verification Competition – Monat et al. [Mon+24]

- Tools have to

Software-Verification Competition – Monat et al. [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)

Software-Verification Competition – Monat et al. [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)

Software-Verification Competition – Monat et al. [Mon+24]

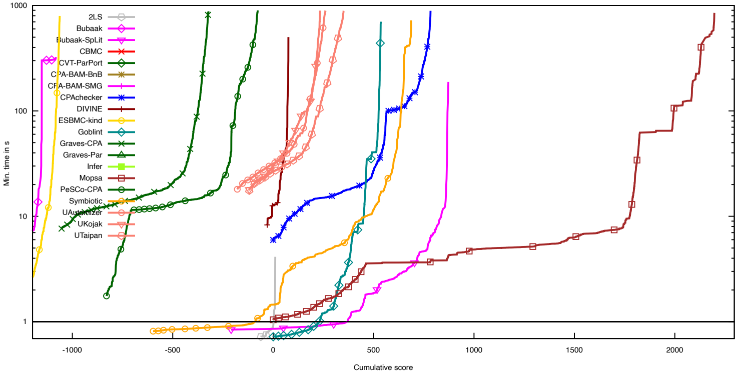
- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference

Software-Verification Competition – Monat et al. [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference
- For our second participation, Mopsa won the “Software Systems” track!

Software-Verification Competition – Monat et al. [Mon+24]

- Tools have to
  - Decide whether a program is correct (large penalties if wrong)
  - Within limited machine resources (15 minutes CPU time, 8GB RAM)
- Corpus of $\simeq$ 23,000 C benchmarks, now acts as a reference
- For our second participation, Mopsa won the “Software Systems” track!



Multilanguage Analysis – Monat, Ouadjaout, and Miné [MOM21]

Assessment 20% of the 200 most popular Python libraries rely on C code

Multilanguage Analysis – Monat, Ouadjaout, and Miné [MOM21]

Assessment 20% of the 200 most popular Python libraries rely on C code

Dangers: different values ($\mathbb{Z}$ vs. `Int32`); shared memory state

Multilanguage Analysis – Monat, Ouadjaout, and Miné [MOM21]

Assessment 20% of the 200 most popular Python libraries rely on C code

Dangers: different values ($\mathbb{Z}$ vs. `Int32`); shared memory state

Our approach: Combined analysis of C, Python and interface code

Library	C + Py. Loc	Tests	⏲/test	$\frac{\text{\# proved checks}}{\text{\# checks}}$ %	# checks
`noise`	1397	15/15	1.2s	99.7%	6690
`cdistance`	2345	28/28	4.1s	98.0%	13716
`llist`	4515	167/194	1.5s	98.8%	36255
`ahocorasick`	4877	46/92	1.2s	96.7%	6722
`levenshtein`	5798	17/17	5.3s	84.6%	4825
`bitarray`	5841	159/216	1.6s	94.9%	25566

Coreutils – Ouadjaout and Miné [OM20]

- Large support of `libc` through stubs

Coreutils – Ouadjaout and Miné [OM20]

- Large support of `libc` through stubs
- Check for all C runtime errors

Coreutils – Ouadjaout and Miné [OM20]

- Large support of `libc` through stubs
- Check for all C runtime errors
- Ability to analyze real-world programs

Coreutils – Ouadjaout and Miné [OM20]

- Large support of `libc` through stubs
- Check for all C runtime errors
- Ability to analyze real-world programs

Benchmark	Time	Selectivity	# checks
basename	33.79s	98.65%	11,731
dirname	21.68s	99.61%	11,307
echo	19.26s	99.43%	11,010
false	14.50s	99.72%	10,774
pwd	22.04s	99.62%	11,502
rmdir	39.00s	99.22%	11,699
sleep	23.79s	99.46%	11,546
tee	35.69s	98.76%	12,057
timeout	32.28s	98.51%	12,420
true	9.55s	99.72%	10,774
uname	20.61s	99.52%	11,943
users	20.82s	99.06%	11,668
whoami	13.03s	99.66%	11,329

Non-exploitability – Parolini and Miné [PM24]

- Focus on bugs that a user can trigger through program interaction

Non-exploitability – Parolini and Miné [PM24]

- Focus on bugs that a user can trigger through program interaction
- Relies on combination of taint+value analysis

Non-exploitability – Parolini and Miné [PM24]

- Focus on bugs that a user can trigger through program interaction
- Relies on combination of taint+value analysis

Test suite	Domain	Analyzer	Alarms	Time
Coreutils	Intervals	MOPSA	4,715	1:17:06
		MOPSA-NEXP	1,217 (-74.19%)	1:28:42 (+15.05%)
	Octagons	MOPSA	4,673	2:22:29
		MOPSA-NEXP	1,209 (-74.13%)	2:43:06 (+14.47%)
	Polyhedra	MOPSA	4,651	2:12:21
		MOPSA-NEXP	1,193 (-74.35%)	2:30:44 (+13.89%)
Juliet	Intervals	MOPSA	49,957	11:32:24
		MOPSA-NEXP	13,906 (-72.16%)	11:48:51 (+2.38%)
	Octagons	MOPSA	48,256	13:15:29
		MOPSA-NEXP	13,631 (-71.75%)	13:41:47 (+3.31%)
	Polyhedra	MOPSA	48,256	12:54:21
		MOPSA-NEXP	13,631 (-71.75%)	13:21:26 (+3.50%)

Future Work Around Mopsa

- Scalability (compositional function analyses)

Future Work Around Mopsa

- Scalability (compositional function analyses)
- Usability

Future Work Around Mopsa

- Scalability (compositional function analyses)
- Usability
  - Resource aware analyses, tailoring for best precision (ANR RAISIN)

Future Work Around Mopsa

- Scalability (compositional function analyses)
- Usability
  - Resource aware analyses, tailoring for best precision (ANR RAISIN)
  - Handling of false alarms (ongoing work by Marco Milanese [MM24])

Future Work Around Mopsa

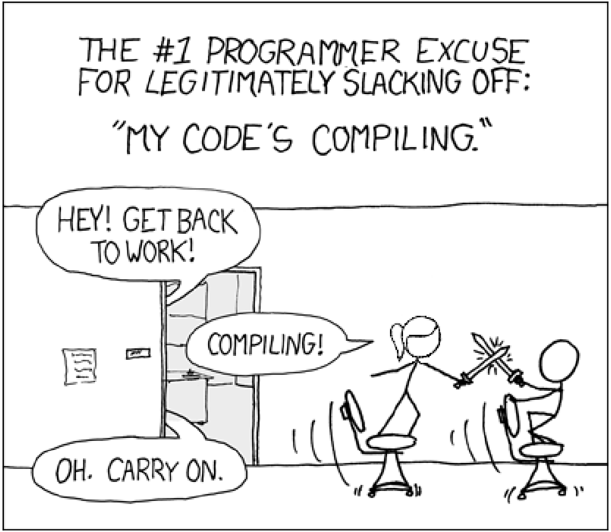
- Scalability (compositional function analyses)
- Usability
  - Resource aware analyses, tailoring for best precision (ANR RAISIN)
  - Handling of false alarms (ongoing work by Marco Milanese [MM24])
- Maintenance and development effort

Future Work Around Mopsa

- Scalability (compositional function analyses)
- Usability
  - Resource aware analyses, tailoring for best precision (ANR RAISIN)
  - Handling of false alarms (ongoing work by Marco Milanese [MM24])
- Maintenance and development effort
- New languages, properties, specific programs

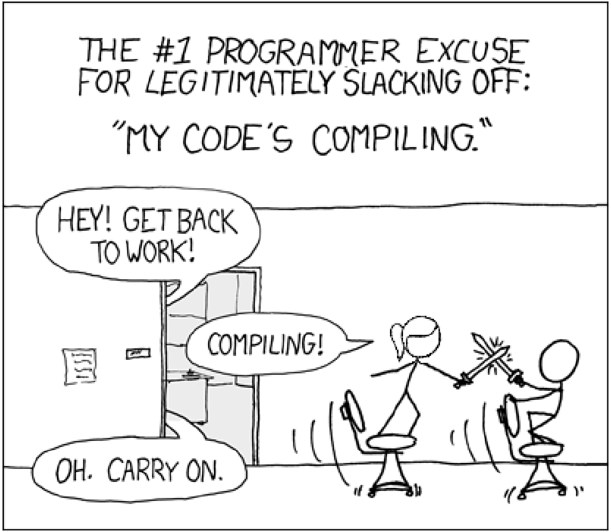
Conclusion

Conclusion



xkcd.com/303

Conclusion

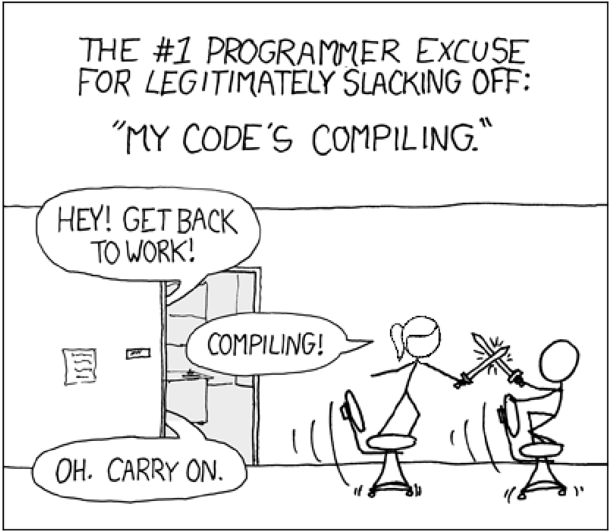


xkcd.com/303

Techniques

- Symbolic execution

Conclusion

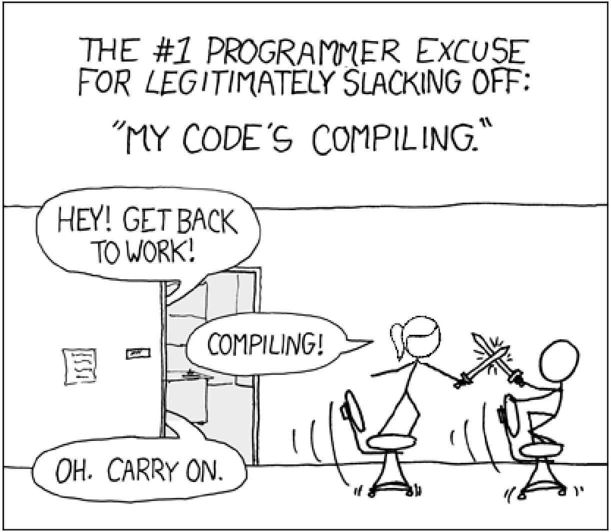


xkcd.com/303

Techniques

- Symbolic execution
- Fuzzing

Conclusion

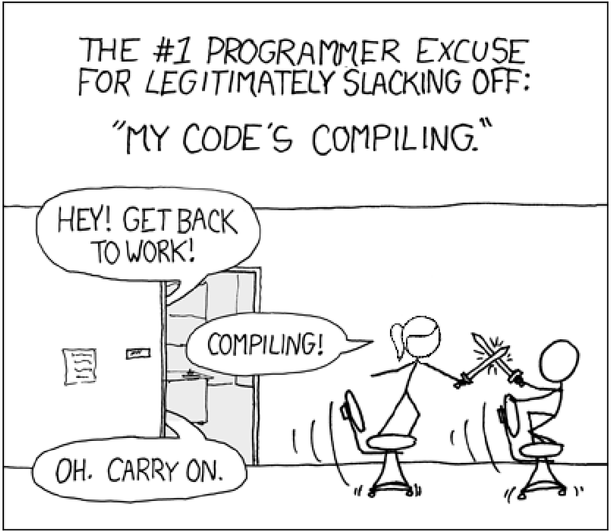


xkcd.com/303

Techniques

- Symbolic execution
- Fuzzing
- Abstract interpretation

Conclusion



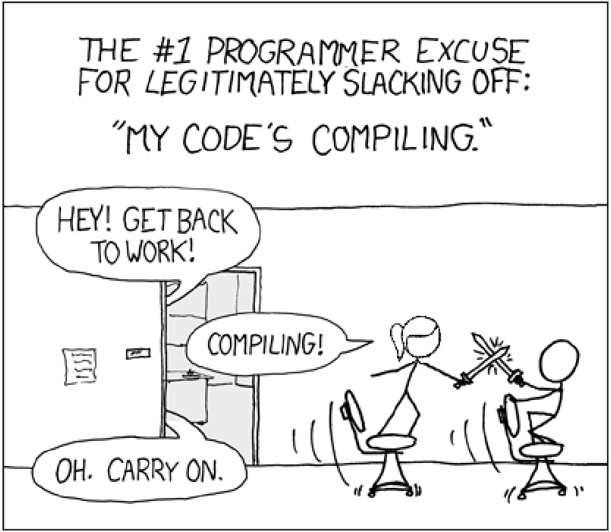
xkcd.com/303

Techniques

- Symbolic execution
- Fuzzing
- Abstract interpretation

Requirements

Conclusion



xkcd.com/303

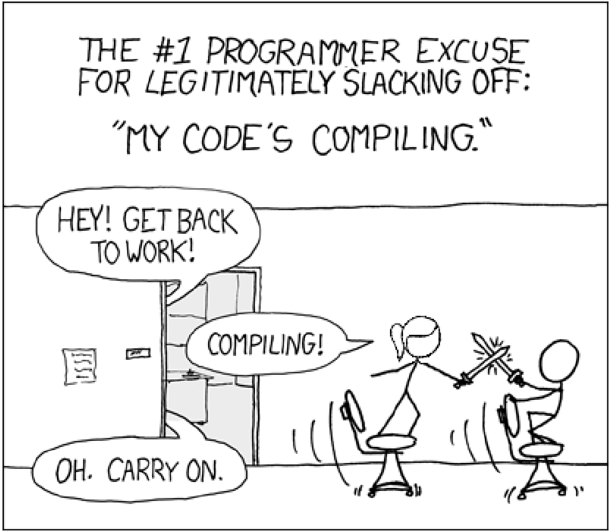
Techniques

- Symbolic execution
- Fuzzing
- Abstract interpretation

Requirements

- Property to verify

Conclusion



xkcd.com/303

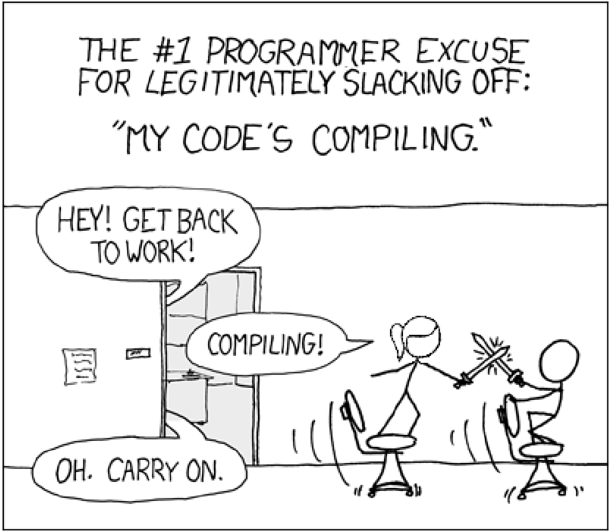
Techniques

- Symbolic execution
- Fuzzing
- Abstract interpretation

Requirements

- Property to verify
- Semantics of language

Conclusion



xkcd.com/303

Techniques

- Symbolic execution
- Fuzzing
- Abstract interpretation

Requirements

- Property to verify
- Semantics of language
- Benchmarks; usecases

References – I

[Bal+18] Roberto Baldoni et al. **"A Survey of Symbolic Execution Techniques".** In: ACM Comput. Surv. 3 (2018), 50:1–50:39.

[Bal+19] Clément Ballabriga et al. **"Static Analysis of Binary Code with Memory Indirections Using Polyhedra".** In: Lecture Notes in Computer Science. Springer, 2019, pp. 114–135.

[Bar+22] Haniel Barbosa et al. **"cvc5: A Versatile and Industrial-Strength SMT Solver".** In: Lecture Notes in Computer Science. Springer, 2022, pp. 415–442.

[BBY17] S. Blazy, D. Bühler, and B. Yakobowski. **"Structuring Abstract Interpreters Through State and Value Abstractions".** In: LNCS. Springer, 2017, pp. 112–130.

References – II

[Ber+10] J. Bertrane et al. "Static analysis and verification of aerospace software by abstract interpretation". In: AIAA-2010-3385. 2010.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org. 2016.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: 2008.

[Con+18] Sylvain Conchon et al. "Alt-Ergo 2.2". In: Oxford, United Kingdom, July 2018.

References – III

[DOM21] David Delmas, Abdelraouf Ouadjaout, and Antoine Miné. **"Static Analysis of Endian Portability by Abstract Interpretation".** In: Lecture Notes in Computer Science. Springer, 2021, pp. 102–123.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. **"DART: directed automated random testing".** In: ACM, 2005, pp. 213–223.

[Jou+19] M. Journault et al. **"Combinations of reusable abstract domains for a multilingual static analyzer".** In: New York, USA, July 2019, pp. 1–17.

[MB08] Leonardo de Moura and Nikolaj Bjørner. **"Z3: An efficient SMT solver".** In: 2008.

References – IV

[Min17] Antoine Miné. “Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation”. In: Found. Trends Program. Lang. 3-4 (2017), pp. 120–372.

[MM24] Marco Milanese and Antoine Miné. “Generation of Violation Witnesses by Under-Approximating Abstract Interpretation”. In: Lecture Notes in Computer Science. Springer, 2024, pp. 50–73.

[MOM21] R. Monat, A. Ouadjaout, and A. Miné. “A Multilanguage Static Analysis of Python Programs with Native C Extensions”. In: 2021.

References – V

[Mon+24] Raphaël Monat et al. **"Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)".** In: Lecture Notes in Computer Science. Springer, 2024, pp. 387–392.

[OM20] A. Ouadjaout and A. Miné. **"A Library Modeling Language for the Static Analysis of C Programs".** In: ed. by David Pichardie and Mihaela Sighireanu. Lecture Notes in Computer Science. Springer, 2020, pp. 223–247. DOI: `10.1007/978-3-030-65474-0_11`.

[PM24] Francesco Parolini and Antoine Miné. **"Sound Abstract Nonexploitability Analysis".** In: Lecture Notes in Computer Science. Springer, 2024, pp. 314–337.

References – VI

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. **"CUTE: a concolic unit testing engine for C".** In: ACM, 2005, pp. 263–272. DOI: 10.1145/1081706.1081750.