UNCLASSIFIED

Australian Government
Department of Defence
Science and Technology

Program Analysis for Reverse Engineers

From $\top$ to $\bot$

Adrian Herrera

Defence Science and Technology Group

April 25, 2023

`$ whoami`

- Researcher with the Defence Science and Technology (DST) Group
- Visiting researcher at the Australian National University (ANU)
- Interested in applying academic research to reverse engineering problems

Outline

1. Introduction
2. SMT solvers
3. Symbolic execution
4. Abstract interpretation
5. Conclusion

Introduction

What is program analysis?

- Automatically reason about a computer program's behaviour
- Active research field for decades
  - E.g. compilers
- What do we want to reason about?
  - **Security**: Can we overflow this array?
  - **Correctness**: Does this loop terminate?
  - **Compiler optimisations**: Is this code reachable?

Static vs. dynamic analysis

Two flavours of program analysis

- **Static analysis**: Analyse the program **without** running it
- **Dynamic analysis**: Analyse the program **while** running it

Static vs. dynamic analysis

Two flavours of program analysis

- **Static analysis**: Analyse the program **without** running it
- **Dynamic analysis**: Analyse the program **while** running it

Static analysis

✓ Reason about **all** executions

✗ Less precise

Static vs. dynamic analysis

Two flavours of program analysis

- **Static analysis**: Analyse the program **without** running it
- **Dynamic analysis**: Analyse the program **while** running it

Static analysis

✓ Reason about **all** executions

✗ Less precise

Dynamic analysis

✗ Reason about **observed** executions

✓ More precise

Static vs. dynamic analysis

Two flavours of program analysis

- **Static analysis**: Analyse the program **without** running it
- **Dynamic analysis**: Analyse the program **while** running it

Static analysis

✓ Reason about **all** executions

✗ Less precise

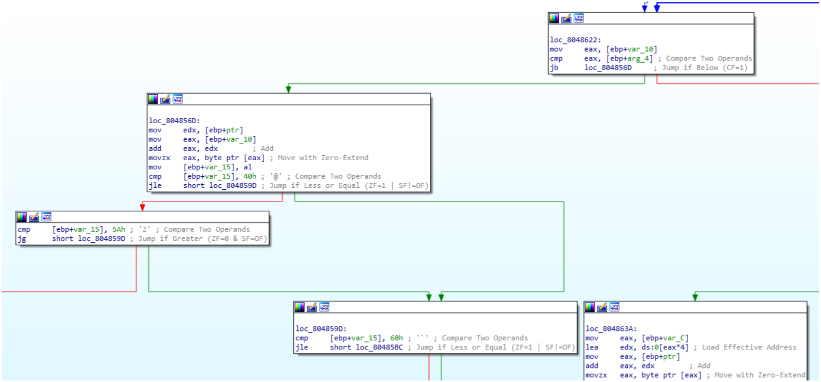
Dynamic analysis

✗ Reason about **observed** executions

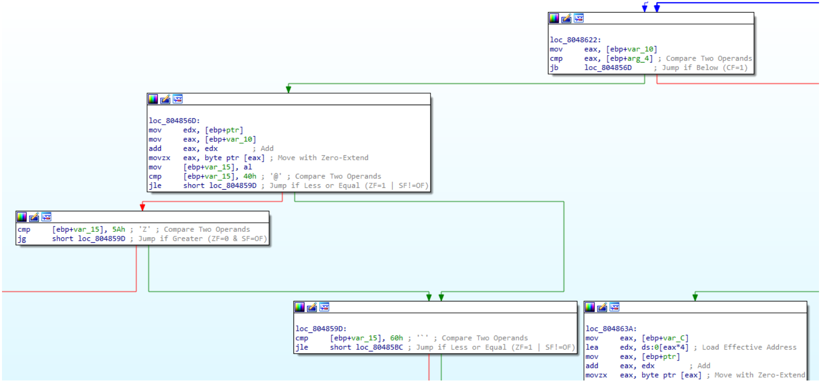
✓ More precise

As a reverse engineer, you already use program analysis

Static analysis

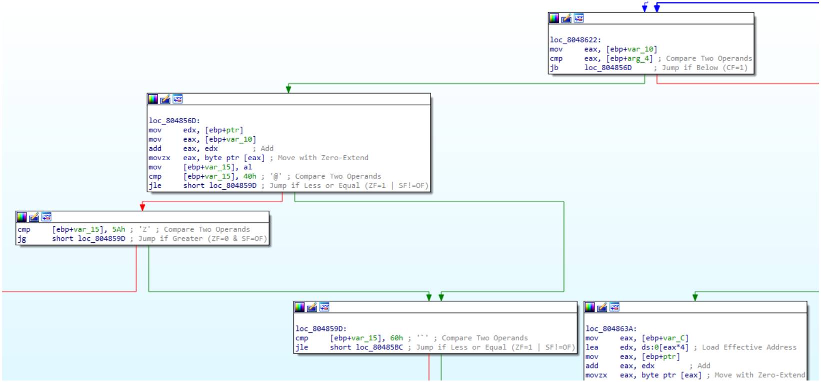


Static analysis



- Disassembly
- Control-flow graph recovery
- Jump-table recovery

Static analysis



- Disassembly
- Control-flow graph recovery
- Jump-table recovery

} **Program analysis**

Dynamic analysis

Input Sample

PID: 3720, Report UID: 00015093-00003720
MD5: c52f20a854efb013a0a1248fd84aaa95
SHA256: cf8533849ee5e82023ad7adbdbd6543cb6db596c53048b1a0c00b3643a72db30

API calls | Registry | Mutants | Handles | Modules | Files | Streams (1)

NtCreateFile@NTDLL.DLL	**pFileHandle**	0
	DesiredAccess	80100080
	ObjectAttributes	1800000000000000ecc92700400000000000000000d8c92700
	IoStatusBlock	185c469400000000
	FileAttributes	0
	ShareAccess	1
	CreateDisposition	1
	CreateOptions	60
	EaBuffer	0
	EaLength	0
	(status)	STATUS_OBJECT_NAME_NOT_FOUND (c0000034)
	(name)	%WINDIR%\assembly\NativeImages_v2.0.50727_32\index23b.dat
NtDelayExecution@NTDLL.DLL	**Alertable**	0
	(originaldelay)	00000030

Dynamic analysis

Input Sample

PID: 3720, Report UID: 00015093-00003720
MD5: c52f20a854efb013a0a1248fd84aaa95
SHA256: cf8533849ee5e82023ad7adbdbd6543cb6db596c53048b1a0c00b3643a72db30

API calls | Registry | Mutants | Handles | Modules | Files | Streams (1)

NtCreateFile@NTDLL.DLL	pFileHandle	0
	DesiredAccess	80100080
	ObjectAttributes	1800000000000000ecc927004000000000000000d8c92700
	IoStatusBlock	185c469400000000
	FileAttributes	0
	ShareAccess	1
	CreateDisposition	1
	CreateOptions	60
	EaBuffer	0
	EaLength	0
	(status)	STATUS_OBJECT_NAME_NOT_FOUND (c0000034)
	(name)	%WINDIR%\assembly\NativeImages_v2.0.50727_32\index23b.dat
NtDelayExecution@NTDLL.DLL	Alertable	0
	(originaldelay)	00000030

- API monitoring
- Code coverage

Dynamic analysis

Input Sample

PID: 3720, Report UID: 00015093-00003720
MD5: c52f20a854efb013a0a1248fd84aaa95
SHA256: cf8533849ee5e82023ad7adbdbd6543cb6db596c53048b1a0c00b3643a72db30

API calls | Registry | Mutants | Handles | Modules | Files | Streams (1)

NtCreateFile@NTDLL.DLL	pFileHandle	0
	DesiredAccess	80100080
	ObjectAttributes	1800000000000000ecc927004000000000000000d8c92700
	IoStatusBlock	185c469400000000
	FileAttributes	0
	ShareAccess	1
	CreateDisposition	1
	CreateOptions	60
	EaBuffer	0
	EaLength	0
	(status)	STATUS_OBJECT_NAME_NOT_FOUND (c0000034)
	(name)	%WINDIR%\assembly\NativeImages_v2.0.50727_32\index23b.dat
NtDelayExecution@NTDLL.DLL	Alertable	0
	(originaldelay)	00000030

- API monitoring
- Code coverage

} **Program analysis**

Program analysis in academia

A Galois Connection Calculus for Abstract Interpretation*

Patrick Cousot
CIMS**, NYU, USA pcousot@cims.nyu.edu

Radhia Cousot
CNRS Emeritus, ENS, France rcousot@ens.fr

Abstract We introduce a Galois connection calculus for language independent specification of abstract interpretations used in programming language semantics, formal verification, and static analysis. This Galois connection calculus and its type system are typed by abstract interpretation.

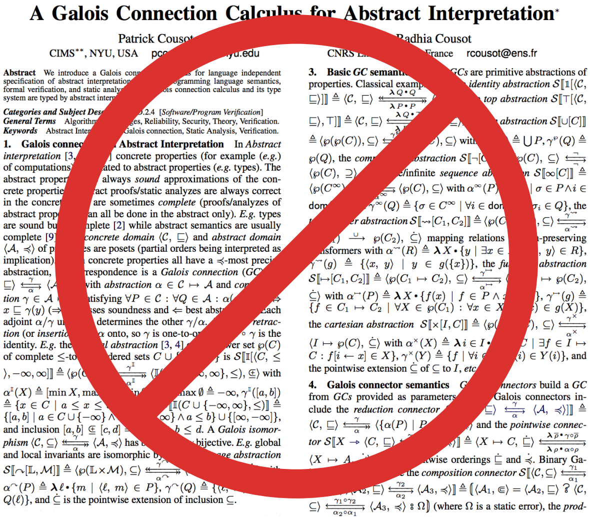
Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]
General Terms Algorithms, Languages, Reliability, Security, Theory, Verification.
Keywords Abstract Interpretation, Galois connection, Static Analysis, Verification.

1. Galois connections in Abstract Interpretation In *Abstract interpretation* [3, 4, 6, 7] concrete properties (for example (*e.g.*) of computations) are related to abstract properties (*e.g.* types). The abstract properties are always *sound* approximations of the concrete properties (abstract proofs/static analyzes are always correct in the concrete) and are sometimes *complete* (proofs/analyzes of abstract properties can all be done in the abstract only). *E.g.* types are sound but incomplete [2] while abstract semantics are usually complete [9]. The *concrete domain* $\langle \mathcal{C}, \sqsubseteq\rangle$ and *abstract domain* $\langle \mathcal{A}, \preccurlyeq\rangle$ of properties are posets (partial orders being interpreted as implication). When concrete properties all have a $\preccurlyeq$-most precise abstraction, the correspondence is a *Galois connection* (*GC*) $\langle \mathcal{C}, \sqsubseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{A}, \preccurlyeq\rangle$ with *abstraction* $\alpha \in \mathcal{C} \mapsto \mathcal{A}$ and *concretization* $\gamma \in \mathcal{A} \mapsto \mathcal{C}$ satisfying $\forall P \in \mathcal{C} : \forall Q \in \mathcal{A} : \alpha(x) \preccurlyeq y \Leftrightarrow x \sqsubseteq \gamma(y)$ ($\Rightarrow$ expresses soundness and $\Leftarrow$ best abstraction). Each adjoint α/γ uniquely determines the other γ/α. A *Galois retraction* (or *insertion*) has α onto, so γ is one-to-one, and $\alpha \circ \gamma$ is the identity. *E.g.* the *interval abstraction* [3, 4] of the power set $\wp(C)$ of complete $\leq$-totally ordered sets $C \cup \{-\infty, \infty\}$ is $\mathcal{S}[\![\mathbb{I}[\langle C, \leq\rangle, -\infty, \infty]]\!] \triangleq \langle \wp(C), \subseteq\rangle \xrightleftharpoons[\alpha^{\mathbb{I}}]{\gamma^{\mathbb{I}}} \langle \mathbb{I}(C \cup \{-\infty, \infty\}, \leq), \Subset\rangle$ with $\alpha^{\mathbb{I}}(X) \triangleq [\min X, \max X]$, $\min \emptyset \triangleq \infty$, $\max \emptyset \triangleq -\infty$, $\gamma^{\mathbb{I}}([a, b]) \triangleq \{x \in C \mid a \leq x \leq b\}$, intervals $\mathcal{S}[\![\mathbb{I}(C \cup \{-\infty, \infty\}, \leq)]\!] \triangleq \{[a, b] \mid a \in C \cup \{-\infty\} \wedge b \in C \cup \{\infty\} \wedge a \leq b\} \cup \{[\infty, -\infty]\}$, and inclusion $[a, b] \Subset [c, d] \triangleq c \leq a \wedge b \leq d$. A *Galois isomorphism* $\langle \mathcal{C}, \subseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{A}, \preccurlyeq\rangle$ has both α and γ bijective. *E.g.* global and local invariants are isomorphic by the *right image abstraction* $\mathcal{S}[\![\curvearrowright[\mathbb{L}, \mathcal{M}]]\!] \triangleq \langle \wp(\mathbb{L} \times \mathcal{M}), \subseteq\rangle \xrightleftharpoons[\alpha^{\curvearrowright}]{\gamma^{\curvearrowright}} \langle \mathbb{L} \mapsto \wp(\mathcal{M}), \dot{\subseteq}\rangle$ with $\alpha^{\curvearrowright}(P) \triangleq \boldsymbol{\lambda} \ell \bullet \{m \mid \langle \ell, m\rangle \in P\}$, $\gamma^{\curvearrowright}(Q) \triangleq \{\langle \ell, m\rangle \mid m \in Q(\ell)\}$, and $\dot{\subseteq}$ is the pointwise extension of inclusion $\subseteq$.

3. Basic *GC* semantics Basic *GCs* are primitive abstractions of properties. Classical examples are the *identity abstraction* $\mathcal{S}[\![\mathbb{1}[\langle \mathcal{C}, \sqsubseteq\rangle]]\!] \triangleq \langle \mathcal{C}, \sqsubseteq\rangle \xrightleftharpoons[\boldsymbol{\lambda} P \bullet P]{\boldsymbol{\lambda} Q \bullet Q} \langle \mathcal{C}, \sqsubseteq\rangle$, the *top abstraction* $\mathcal{S}[\![\top[\langle \mathcal{C}, \sqsubseteq\rangle, \top]]\!] \triangleq \langle \mathcal{C}, \sqsubseteq\rangle \xrightleftharpoons[\boldsymbol{\lambda} P \bullet \top]{\boldsymbol{\lambda} Q \bullet \top} \langle \mathcal{C}, \sqsubseteq\rangle$, the *join abstraction* $\mathcal{S}[\![\cup[C]]\!] \triangleq \langle \wp(\wp(C)), \subseteq\rangle \xrightleftharpoons[\alpha^{\wp}]{\gamma^{\wp}} \langle \wp(C), \subseteq\rangle$ with $\alpha^{\wp}(P) \triangleq \bigcup P$, $\gamma^{\wp}(Q) \triangleq \wp(Q)$, the *complement abstraction* $\mathcal{S}[\![\neg[C]]\!] \triangleq \langle \wp(C), \subseteq\rangle \xrightleftharpoons[\neg]{\neg} \langle \wp(C), \supseteq\rangle$, the finite/infinite *sequence abstraction* $\mathcal{S}[\![\infty[C]]\!] \triangleq \langle \wp(C^{\infty}), \subseteq\rangle \xrightleftharpoons[\alpha^{\infty}]{\gamma^{\infty}} \langle \wp(C), \subseteq\rangle$ with $\alpha^{\infty}(P) \triangleq \{\sigma_i \mid \sigma \in P \wedge i \in \mathsf{dom}(\sigma)\}$ and $\gamma^{\infty}(Q) \triangleq \{\sigma \in C^{\infty} \mid \forall i \in \mathsf{dom}(\sigma) : \sigma_i \in Q\}$, the *transformer abstraction* $\mathcal{S}[\![\rightsquigarrow[C_1, C_2]]\!] \triangleq \langle \wp(C_1 \times C_2), \subseteq\rangle \xrightleftharpoons[\alpha^{\rightsquigarrow}]{\gamma^{\rightsquigarrow}} \langle \wp(C_1) \xrightarrow{\cup} \wp(C_2), \dot{\subseteq}\rangle$ mapping relations to join-preserving transformers with $\alpha^{\rightsquigarrow}(R) \triangleq \boldsymbol{\lambda} X \bullet \{y \mid \exists x \in X : \langle x, y\rangle \in R\}$, $\gamma^{\rightsquigarrow}(g) \triangleq \{\langle x, y\rangle \mid y \in g(\{x\})\}$, the *function abstraction* $\mathcal{S}[\![\mapsto[C_1, C_2]]\!] \triangleq \langle \wp(C_1 \mapsto C_2), \subseteq\rangle \xrightleftharpoons[\alpha^{\mapsto}]{\gamma^{\mapsto}} \langle \wp(C_1) \mapsto \wp(C_2), \dot{\subseteq}\rangle$ with $\alpha^{\mapsto}(P) \triangleq \boldsymbol{\lambda} X \bullet \{f(x) \mid f \in P \wedge x \in X\}$, $\gamma^{\mapsto}(g) \triangleq \{f \in C_1 \mapsto C_2 \mid \forall X \in \wp(C_1) : \forall x \in X : f(x) \in g(X)\}$, the *cartesian abstraction* $\mathcal{S}[\![\times[I, C]]\!] \triangleq \langle \wp(I \mapsto C), \subseteq\rangle \xrightleftharpoons[\alpha^{\times}]{\gamma^{\times}} \langle I \mapsto \wp(C), \dot{\subseteq}\rangle$ with $\alpha^{\times}(X) \triangleq \boldsymbol{\lambda} i \in I \bullet \{x \in C \mid \exists f \in I \mapsto C : f[i \leftarrow x] \in X\}$, $\gamma^{\times}(Y) \triangleq \{f \mid \forall i \in I : f(i) \in Y(i)\}$, and the pointwise extension $\dot{\subseteq}$ of $\subseteq$ to I, *etc.*

4. Galois connector semantics *Galois connectors* build a *GC* from *GCs* provided as parameters. Unary Galois connectors include the *reduction connector* $\mathcal{S}[\![\mathsf{R}[\langle \mathcal{C}, \sqsubseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{A}, \preccurlyeq\rangle]]\!] \triangleq \langle \mathcal{C}, \sqsubseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle \{\alpha(P) \mid P \in \mathcal{C}\}, \preccurlyeq\rangle$ and the *pointwise connector* $\mathcal{S}[\![X \rightarrow \langle C, \sqsubseteq\rangle \xrightleftharpoons[\alpha]{\gamma} \langle A, \preccurlyeq\rangle]\!] \triangleq \langle X \mapsto C, \dot{\sqsubseteq}\rangle \xrightleftharpoons[\boldsymbol{\lambda} \rho \bullet \alpha \circ \rho]{\boldsymbol{\lambda} \overline{\rho} \bullet \gamma \circ \overline{\rho}} \langle X \mapsto A, \dot{\preccurlyeq}\rangle$ for the pointwise orderings $\dot{\sqsubseteq}$ and $\dot{\preccurlyeq}$. Binary Galois connectors include the *composition connector* $\mathcal{S}[\![\langle \mathcal{C}, \subseteq\rangle \xrightleftharpoons[\alpha_1]{\gamma_1} \langle \mathcal{A}_1, \Subset\rangle \fatsemi \langle \mathcal{A}_2, \sqsubseteq\rangle \xrightleftharpoons[\alpha_2]{\gamma_2} \langle \mathcal{A}_3, \preccurlyeq\rangle]\!] \triangleq (\!|\langle \mathcal{A}_1, \Subset\rangle = \langle \mathcal{A}_2, \sqsubseteq\rangle \mathrel{?} \langle \mathcal{C}, \subseteq\rangle \xrightleftharpoons[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \langle \mathcal{A}_3, \preccurlyeq\rangle \mathrel{:} \Omega|\!)$ (where Ω is a static error), the *prod-*

Program analysis in academia



A Galois Connection Calculus for Abstract Interpretation*

Patrick Cousot
CIMS**, NYU, USA

Radhia Cousot
rcousot@ens.fr

Abstract

Categories and Subject Descriptors

General Terms

Keywords

1. Galois connection ... Abstract Interpretation

3. Basic GC semantics

4. Galois connector semantics

Aims

- ✓ Introduce new program analysis techniques
- ✓ Focus on reverse engineering
- ✓ Example based

Aims

- ✓ Introduce new program analysis techniques
- ✓ Focus on reverse engineering
- ✓ Example based
- ✗ Limit the maths
- ✗ Limit the code

Aims

- ✓ Introduce new program analysis techniques
- ✓ Focus on reverse engineering
- ✓ Example based
- ✗ Limit the maths
- ✗ Limit the code
- ✗ Won't focus on specific ISA
- ✓ Perform analysis on an IR

Aims

- ✓ Introduce new program analysis techniques
- ✓ Focus on reverse engineering
- ✓ Example based
- ✗ Limit the maths
- ✗ Limit the code
- ✗ Won't focus on specific ISA
- ✓ Perform analysis on an IR
  - REIL

Reverse Engineering Intermediate Language (REIL)

- Developed by Zynamics (now Google)
- Used in Binnavi
- Simple, reduced instruction set
  - No implicit side effects
  - 17 instructions
  - All instructions take 3 operands (may be unused)

REIL syntax & semantics

```
0001: xor [DWORD r1, DWORD r1, DWORD r1]
0002: add [DWORD 10, DWORD r1, DWORD r1]
0003: str [DWORD 20, , DWORD r2]
0004: add [DWORD r1, DWORD r2, DWORD r1]
0005: stm [DWORD r1, , DWORD 0x12345678]
```

REIL syntax & semantics

```
0001: xor [DWORD r1, DWORD r1, DWORD r1]
0002: add [DWORD 10, DWORD r1, DWORD r1]
0003: str [DWORD 20, , DWORD r2]
0004: add [DWORD r1, DWORD r2, DWORD r1]
0005: stm [DWORD r1, , DWORD 0x12345678]
```

Important to differentiate between **syntax** and **semantics**

REIL syntax & semantics

```
0001: xor [DWORD r1, DWORD r1, DWORD r1]
0002: add [DWORD 10, DWORD r1, DWORD r1]
0003: str [DWORD 20, , DWORD r2]
0004: add [DWORD r1, DWORD r2, DWORD r1]
0005: stm [DWORD r1, , DWORD 0x12345678]
```

Important to differentiate between **syntax** and **semantics**

Syntax The words (*symbols*) that make up a sentence

Semantics The *meaning* behind the sentence

REIL syntax & semantics

```
0001: xor [DWORD r1, DWORD r1, DWORD r1]
0002: add [DWORD 10, DWORD r1, DWORD r1]
0003: str [DWORD 20, , DWORD r2]
0004: add [DWORD r1, DWORD r2, DWORD r1]
0005: stm [DWORD r1, , DWORD 0x12345678]
```

Important to differentiate between **syntax** and **semantics**

Syntax

Instructions	`xor`, `add`, `str`, ...
Operand sizes	`BYTE`, `WORD`, `DWORD`, ...
Registers	`r1`, `r2`, ...
Literals	`10`, `0x12345678`, ...

REIL syntax & semantics

```
0001: xor [DWORD r1, DWORD r1, DWORD r1]
0002: add [DWORD 10, DWORD r1, DWORD r1]
0003: str [DWORD 20, , DWORD r2]
0004: add [DWORD r1, DWORD r2, DWORD r1]
0005: stm [DWORD r1, , DWORD 0x12345678]
```

Important to differentiate between **syntax** and **semantics**

Semantics

`add [DWORD 10, DWORD r1, DWORD r1]`

1. Look up the value of register `r1`
2. Add the value 10 to the value from 1.
3. Store the result of 2. in register `r1`

Let's do some program analysis!

SMT solvers

Modelling code with maths

```
if ((x >= 3 && (y * 2 - x < 20) && !(y > 1 || y >= 10)) &&
        (x * y * z == -50)) {
    // ...
}
```

Modelling code with maths

```
if ((x >= 3 && (y * 2 - x < 20) && !(y > 1 || y >= 10)) &&
        (x * y * z == -50)) {
    // ...
}
```

Model this code with a **first-order logic** formula

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg (y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Modelling code with maths

```
if ((x >= 3 && (y * 2 - x < 20) && !(y > 1 || y >= 10)) &&
        (x * y * z == -50)) {
    // ...
}
```

Model this code with a **first-order logic** formula

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg (y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

conjunction ("and") negation ("not") disjunction ("or")

Modelling code with maths

We can

Modelling code with maths

We can

- **Assign** values to the formula's variables (x, y and z)

Modelling code with maths

We can

- **Assign** values to the formula's variables (x, y and z)
- Check if the formula is **satisfiable**

Modelling code with maths

We can

- **Assign** values to the formula's variables (x, y and z)
- Check if the formula is **satisfiable**
  - There exists a set of assignments that makes the formula true

Modelling code with maths

We can

- **Assign** values to the formula's variables (x, y and z)
- Check if the formula is **satisfiable**
  - There exists a set of assignments that makes the formula true
- Check if the formula is **valid**

Modelling code with maths

We can

- **Assign** values to the formula's variables (x, y and z)
- Check if the formula is **satisfiable**
  - There exists a set of assignments that makes the formula true
- Check if the formula is **valid**
  - The formula is true under **all** assignments

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$(5 \geq 3 \wedge (5 \times 2 - 5 < 20) \wedge \neg(5 > 1 \vee 5 \geq 10)) \wedge (5 \times 5 \times -2 = -50)$$

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$(\top \wedge (5 < 20) \wedge \neg(\top \vee \bot)) \wedge (-50 = -50)$$

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$(\top \land (5 < 20) \land \neg(\top \lor \bot)) \land (-50 = -50)$$

top ("true") bottom ("false")

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$(\top \wedge \top \wedge \neg\top) \wedge \top$$

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$(\top \wedge \top \wedge \bot) \wedge \top$$

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$\bot \land \top$$

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Assignment

$$x = 5, y = 5, z = -2$$

$$\bot$$

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Is the formula valid?

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Is the formula valid? **No**

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Is the formula satisfiable?

A simple example

$$(x \geq 3 \land (y \times 2 - x < 20) \land \neg(y > 1 \lor y \geq 10)) \land (x \times y \times z = -50)$$

Is the formula satisfiable? **Yes**

A simple example

$$(x \geq 3 \wedge (y \times 2 - x < 20) \wedge \neg(y > 1 \vee y \geq 10)) \wedge (x \times y \times z = -50)$$

Is the formula satisfiable? **Yes**

When

$$x = 5$$

$$y = -2$$

$$z = 5$$

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form
   - Each variable is assigned exactly once

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form
   - Each variable is assigned exactly once
   - Reassignments create a new version of that variable

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form
    - Each variable is assigned exactly once
    - Reassignments create a new version of that variable
2. Model each SSA instruction as a logical formula

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form
   - Each variable is assigned exactly once
   - Reassignments create a new version of that variable
2. Model each SSA instruction as a logical formula
3. Take the conjunction of all instructions from 2.

General approach

Automate process with a **Satisfiability Modulo Theories** (SMT) solver

1. Convert code to **static single assignment** (SSA) form
   - Each variable is assigned exactly once
   - Reassignments create a new version of that variable
2. Model each SSA instruction as a logical formula
3. Take the conjunction of all instructions from 2.
4. Query the resulting formula in an SMT solver

A more complex example

```
0001: xor [r1, r1, r1]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3]
0005: xor [r4, r4, r4]
0006: add [r4, r3, r4]
0007: bsh [r4, 1, r5]
0008: add [in, r1, r1]
0009: sub [in, r3, in]
000a: bsh [in, 1, in]
000b: div [r5, 16, r4]
000c: mul [r3, 2, r3]
000d: add [in, r3, r3]
000e: mul [r3, r3, r5]
000f: add [r2, 5, r2]
0010: div [r5, r2, r5]
0011: add [r5, r1, r1]
0012: mod [r1, 2, r3]
0013: jcc [r3, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Convert to SSA

Append _a, _b, _c, etc. to denote reassignments

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Model as logical formulas

To reach 0014

$$r1_a = r1 \oplus r1$$

$$r2 = -1$$

$$r3 = 234$$

$$r3_a = r2 \times r3$$

$$r4_a = r4 \oplus r4$$

$$r4_b = r4_a + r3_a$$

$$\ldots$$

$$r3_d = 0$$

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Model as logical formulas

To reach 0020

$$r1_a = r1 \oplus r1$$
$$r2 = -1$$
$$r3 = 234$$
$$r3_a = r2 \times r3$$
$$r4_a = r4 \oplus r4$$
$$r4_b = r4_a + r3_a$$
$$\ldots$$
$$r3_d \neq 0$$

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Take the conjunction

To reach 0014

$$
\begin{aligned}
r1_a &= r1 \oplus r1 \\
\wedge\ r2 &= -1 \\
\wedge\ r3 &= 234 \\
\wedge\ r3_a &= r2 \times r3 \\
\wedge\ r4_a &= r4 \oplus r4 \\
\wedge\ r4_b &= r4_a + r3_a \\
&\dots \\
\wedge\ r3_d &= 0
\end{aligned}
$$

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Take the conjunction

To reach 0020

$$
\begin{aligned}
r1_a &= r1 \oplus r1 \\
\wedge\ r2 &= -1 \\
\wedge\ r3 &= 234 \\
\wedge\ r3_a &= r2 \times r3 \\
\wedge\ r4_a &= r4 \oplus r4 \\
\wedge\ r4_b &= r4_a + r3_a \\
&\dots \\
\wedge\ r3_d &\neq 0
\end{aligned}
$$

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

Check for satisfiability

To reach 0014 ($r3_d = 0$)

$$in = 0$$

```
0014: ; ...
```

```
0020: ; ...
```

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Check for satisfiability

To reach 0014 ($r3_d = 0$)

$in = 0$

What other values for in can reach 0014?

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Check for satisfiability

To reach 0014 ($r3_d = 0$)

$in = 0$

What other values for `in` can reach 0014? Add additional constraint

$in \neq 0$

Recheck for satisfiability

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

Check for satisfiability

To reach 0020 ($r3_d \neq 0$)

```
0014: ; ...
```

```
0020: ; ...
```

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Check for satisfiability

To reach 0020 ($r3_d \neq 0$)

Unsatisfiable

A more complex example

```
0001: xor [r1, r1, r1_a]
0002: str [-1, , r2]
0003: str [234, , r3]
0004: mul [r2, r3, r3_a]
0005: xor [r4, r4, r4_a]
0006: add [r4_a, r3_a, r4_b]
0007: bsh [r4_b, 1, r5]
0008: add [in, r1_a, r1_b]
0009: sub [in, r3_a, in_a]
000a: bsh [in_a, 1, in_b]
000b: div [r5, 16, r4_c]
000c: mul [r3_a, 2, r3_b]
000d: add [in_b, r3_b, r3_c]
000e: mul [r3_c, r3_c, r5_a]
000f: add [r2, 5, r2_a]
0010: div [r5_a, r2_a, r5_b]
0011: add [r5_b, r1_b, r1_c]
0012: mod [r1_c, 2, r3_d]
0013: jcc [r3_d, , 0020]
```

```
0014: ; ...
```

```
0020: ; ...
```

Check for satisfiability

To reach 0020 ($r3_d \neq 0$)

Unsatisfiable

This is an **opaque predicate** – no need to RE this path

Summary

- Applications
  - Opaque predicate detection
  - Dead-code detection
  - Automatic exploit generation (AEG)
- Loops?
  - Typically unrolled

Summary

- Applications
  - Opaque predicate detection
  - Dead-code detection
  - Automatic exploit generation (AEG)
- Loops?
  - Typically unrolled

Challenges

- SMT solvers may not be able to solve complex formulas
- Unbounded/infinite loops
- Appropriate semantics

Symbolic execution

Introduction

Previously

Statically modelled code as first-order logic formulas

Introduction

Previously

Statically modelled code as first-order logic formulas

Now

Run program through interpreter that operates on **symbolic** values and generate logic formulas dynamically

Open-source tools

TRITON

Dynamic Binary Analysis

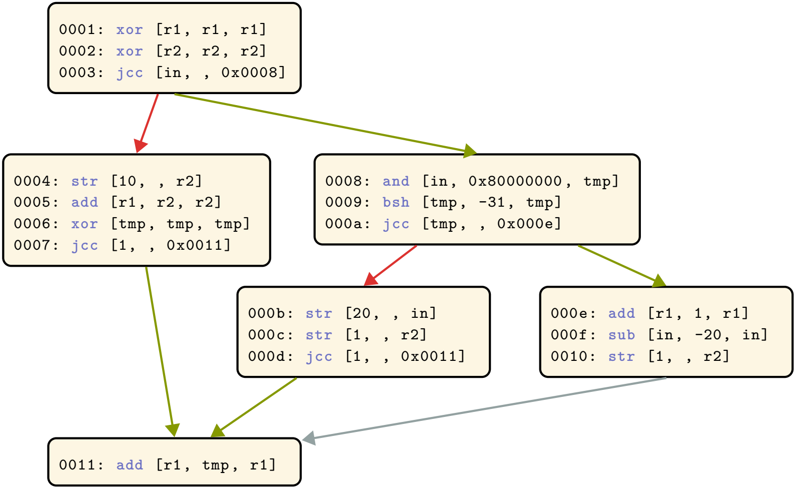
MANTICORE

S2E

General approach

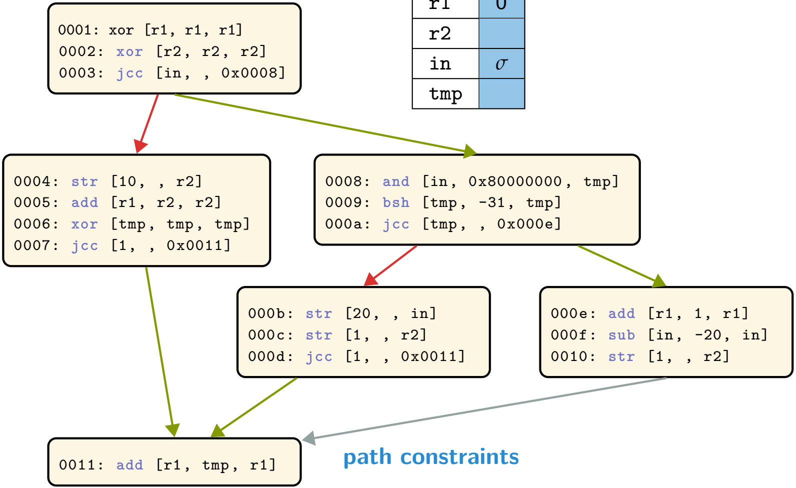
- Program input is provided as **symbolic** values (rather than **concrete**)
- Operations (e.g. addition, assignment, etc.) operate on these symbolic values to generate **symbolic expressions**
- Conditional statements (e.g. `jcc`) result in a **fork** – both paths are explored
- Invoke an SMT solver to find a solution to the symbolic expressions – this is a concrete input for the path explored

Example

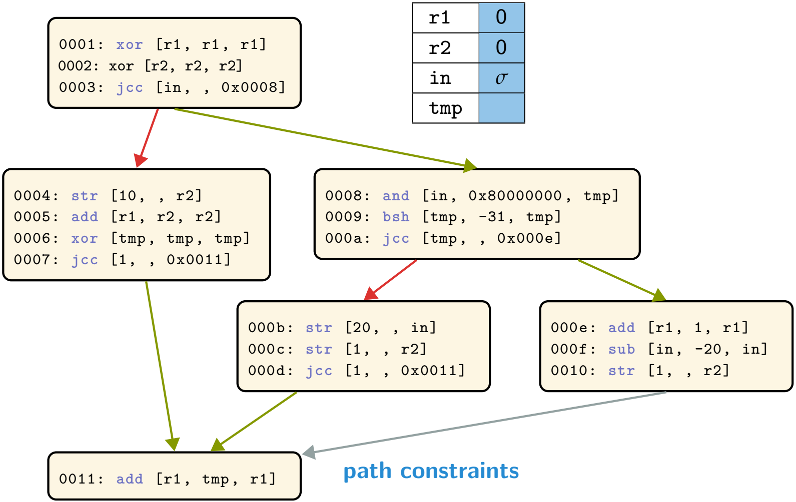

0001: xor [r1, r1, r1]
0002: xor [r2, r2, r2]
0003: jcc [in, , 0x0008]
0004: str [10, , r2]
0005: add [r1, r2, r2]
0006: xor [tmp, tmp, tmp]
0007: jcc [1, , 0x0011]
0008: and [in, 0x80000000, tmp]
0009: bsh [tmp, -31, tmp]
000a: jcc [tmp, , 0x000e]
000b: str [20, , in]
000c: str [1, , r2]
000d: jcc [1, , 0x0011]
000e: add [r1, 1, r1]
000f: sub [in, -20, in]
0010: str [1, , r2]
0011: add [r1, tmp, r1]

Example

r1	0
r2	
in	σ
tmp	

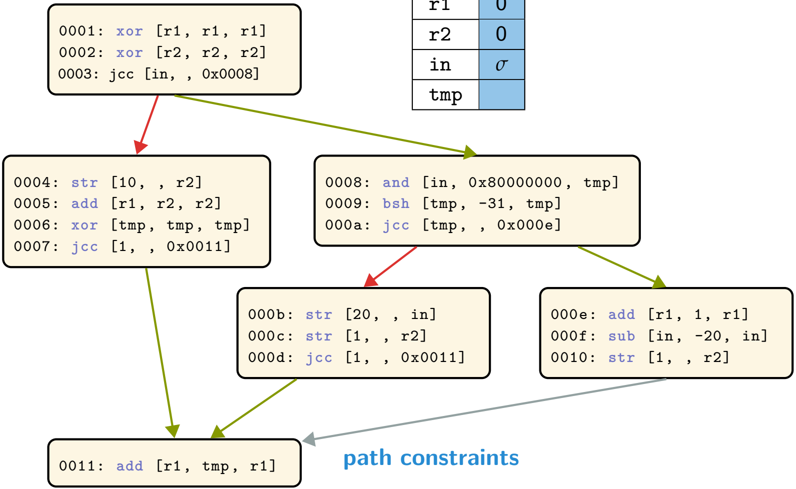


Example


0001: xor [r1, r1, r1]
0002: xor [r2, r2, r2]
0003: jcc [in, , 0x0008]
r1 0
r2 0
in σ
tmp
0004: str [10, , r2]
0005: add [r1, r2, r2]
0006: xor [tmp, tmp, tmp]
0007: jcc [1, , 0x0011]
0008: and [in, 0x80000000, tmp]
0009: bsh [tmp, -31, tmp]
000a: jcc [tmp, , 0x000e]
000b: str [20, , in]
000c: str [1, , r2]
000d: jcc [1, , 0x0011]
000e: add [r1, 1, r1]
000f: sub [in, -20, in]
0010: str [1, , r2]
0011: add [r1, tmp, r1]
path constraints

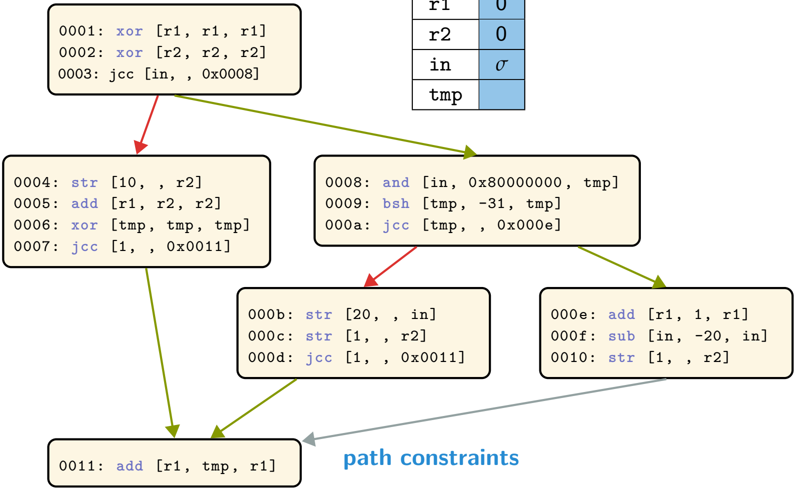
Example

r1	0
r2	0
in	σ
tmp	

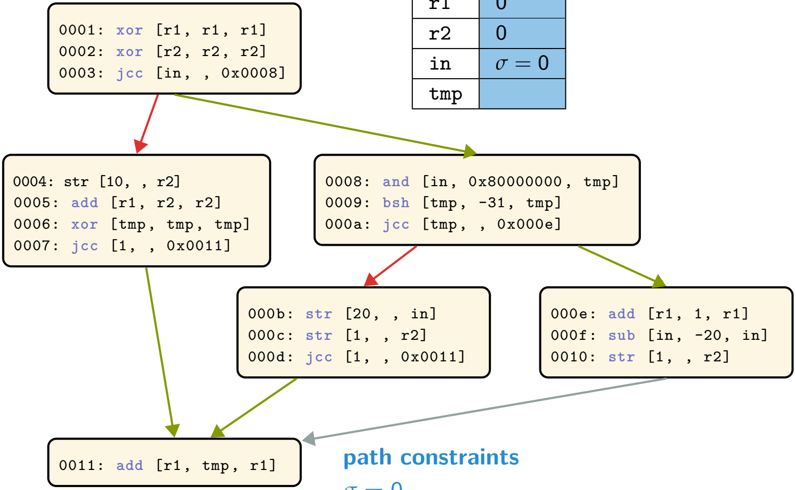


Example

r1	0
r2	0
in	σ
tmp	



Example



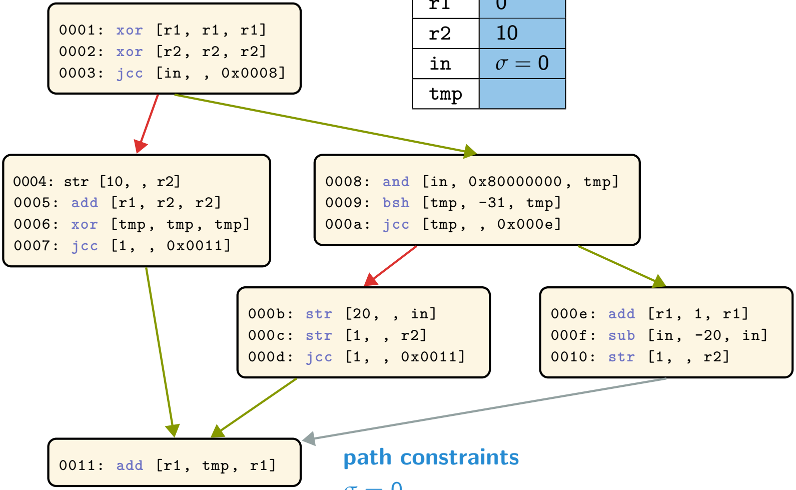
r1	0
r2	0
in	$\sigma = 0$
tmp	

path constraints

$\sigma = 0$

Example

r1	0
r2	10
in	$\sigma = 0$
tmp	

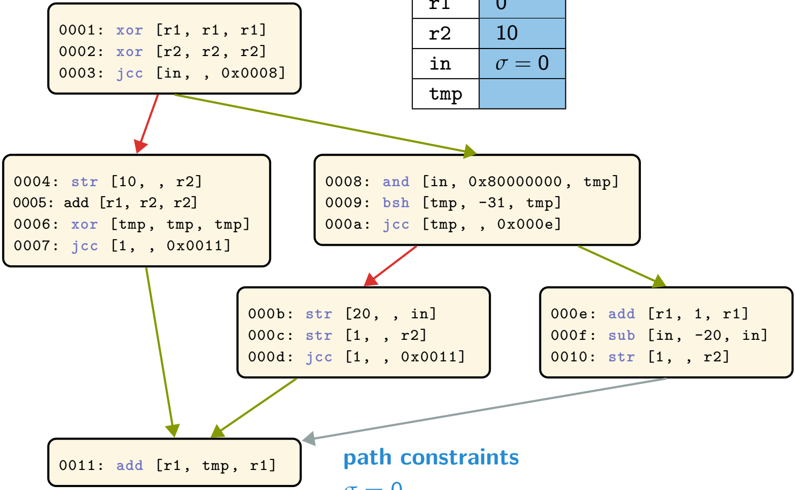


path constraints

$\sigma = 0$

Example

r1	0
r2	10
in	$\sigma = 0$
tmp	

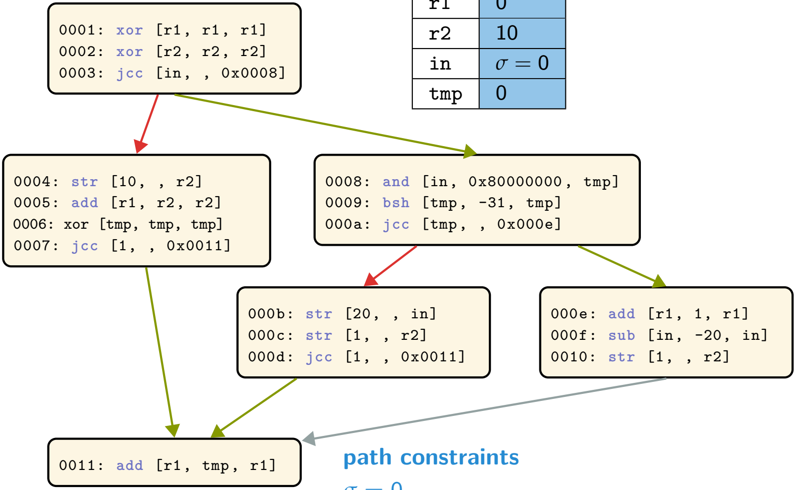


path constraints

$\sigma = 0$

Example

r1	0
r2	10
in	$\sigma = 0$
tmp	0

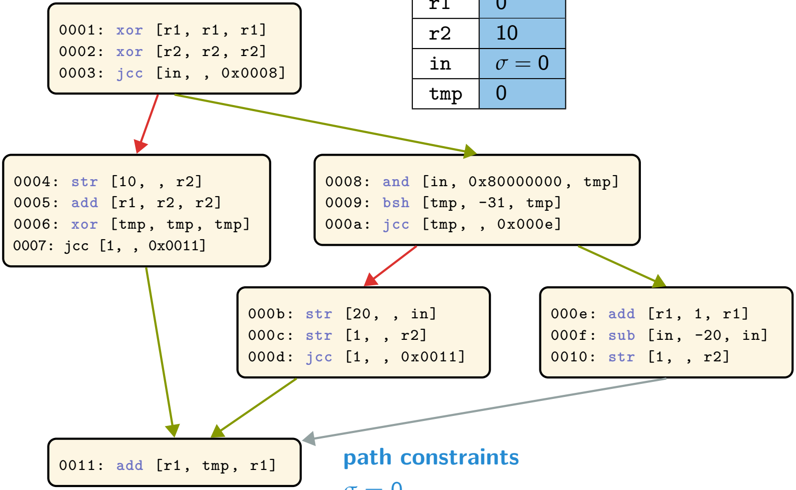


path constraints

$\sigma = 0$

Example

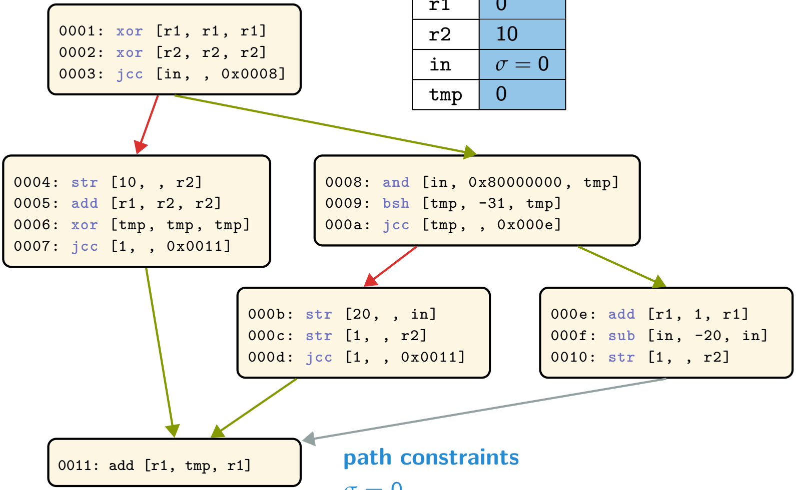
r1	0
r2	10
in	$\sigma = 0$
tmp	0



path constraints

$\sigma = 0$

Example



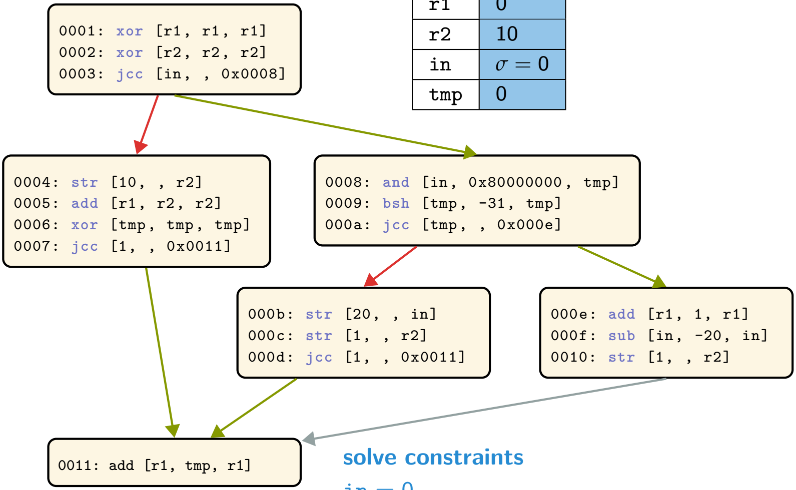
r1	0
r2	10
in	$\sigma = 0$
tmp	0

path constraints

$\sigma = 0$

Example

r1	0
r2	10
in	$\sigma = 0$
tmp	0

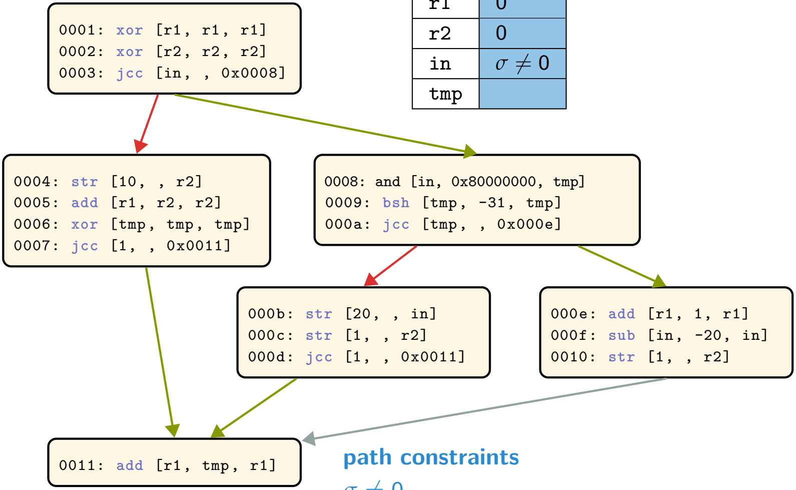


solve constraints

$\texttt{in} = 0$

Example

r1	0
r2	0
in	$\sigma \neq 0$
tmp	

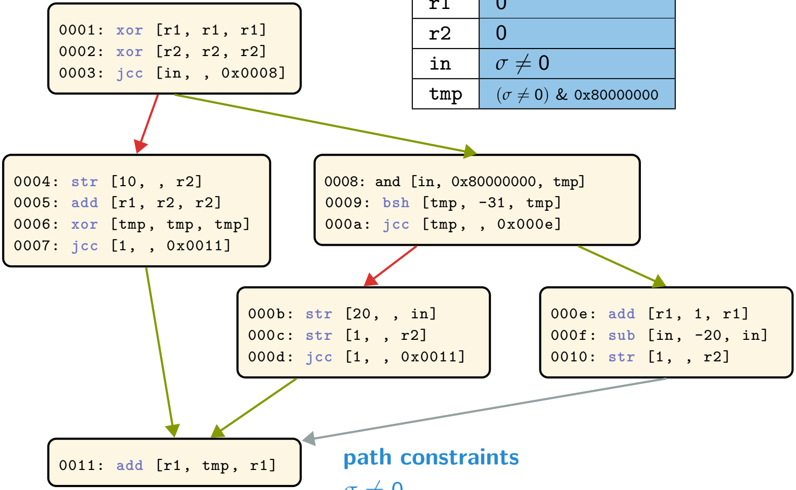


path constraints

$\sigma \neq 0$

Example

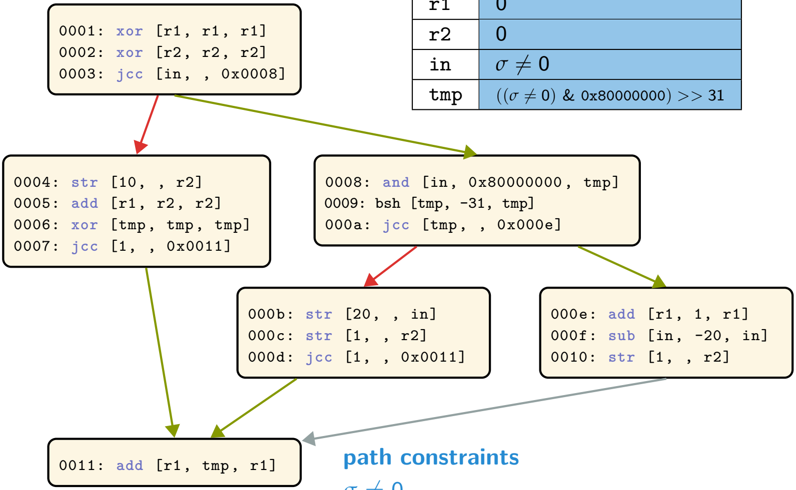
r1	0
r2	0
in	$\sigma \neq 0$
tmp	$(\sigma \neq 0)$ & 0x80000000



path constraints

$\sigma \neq 0$

Example

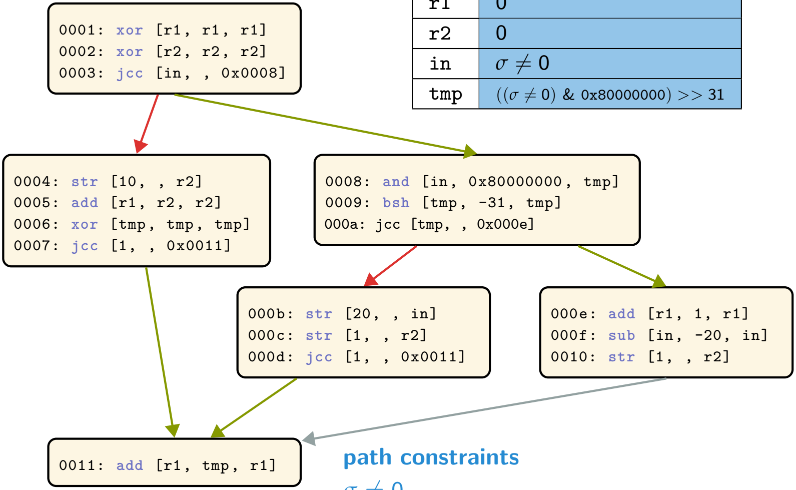


r1	0
r2	0
in	$\sigma \neq 0$
tmp	$((\sigma \neq 0)$ & 0x80000000$) >> 31$

path constraints

$\sigma \neq 0$

Example



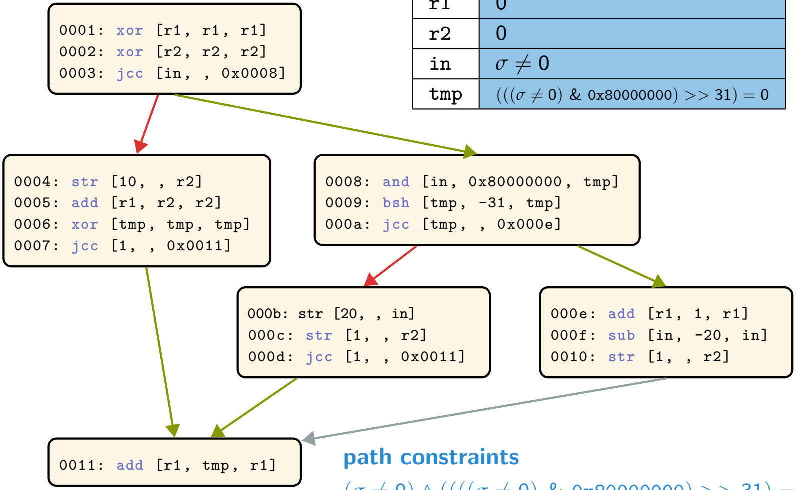
r1	0
r2	0
in	$\sigma \neq 0$
tmp	$((\sigma \neq 0)$ & 0x80000000) >> 31

path constraints

$\sigma \neq 0$

Example

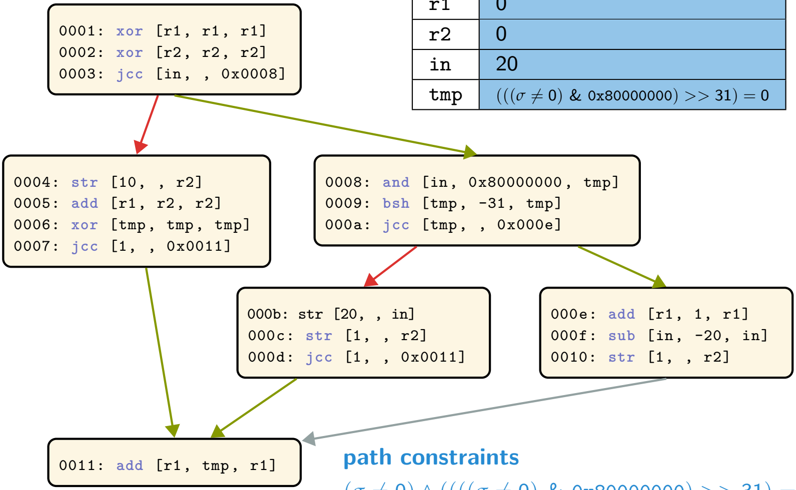
r1	0
r2	0
in	$\sigma \neq 0$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$



path constraints

$$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0)$$

Example



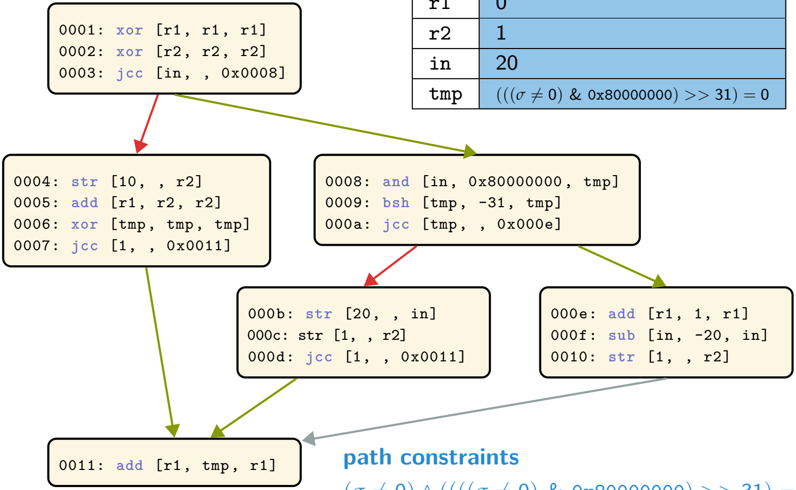
r1	0
r2	0
in	20
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$

path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0)$

Example

r1	0
r2	1
in	20
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$

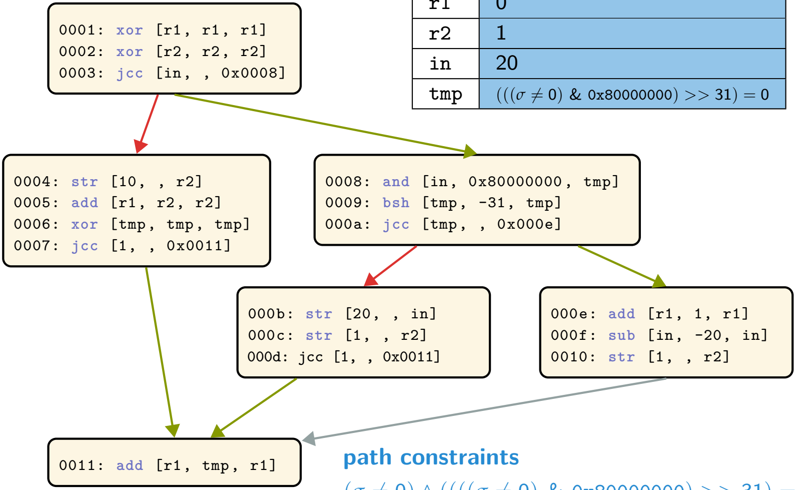


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0)$

Example

r1	0
r2	1
in	20
tmp	$(((\sigma \neq 0)\ \&\ \texttt{0x80000000}) >> 31) = 0$

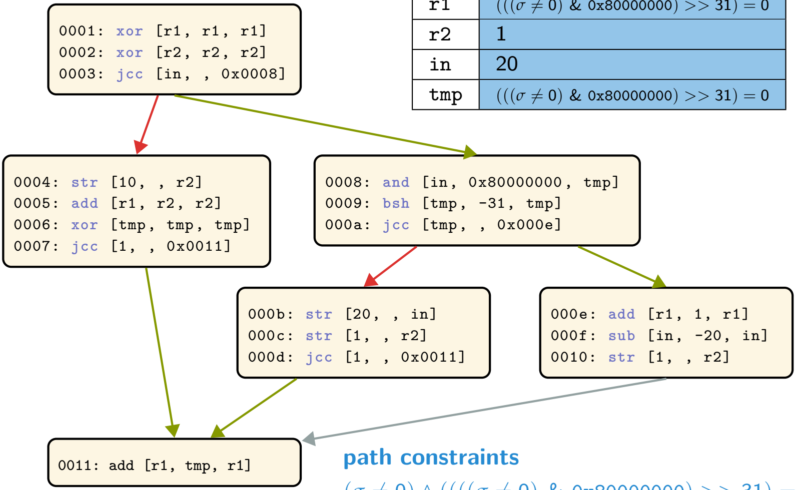


path constraints

$$(\sigma \neq 0) \land ((((\sigma \neq 0)\ \&\ \texttt{0x80000000}) >> 31) = 0)$$

Example

r1	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$
r2	1
in	20
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$

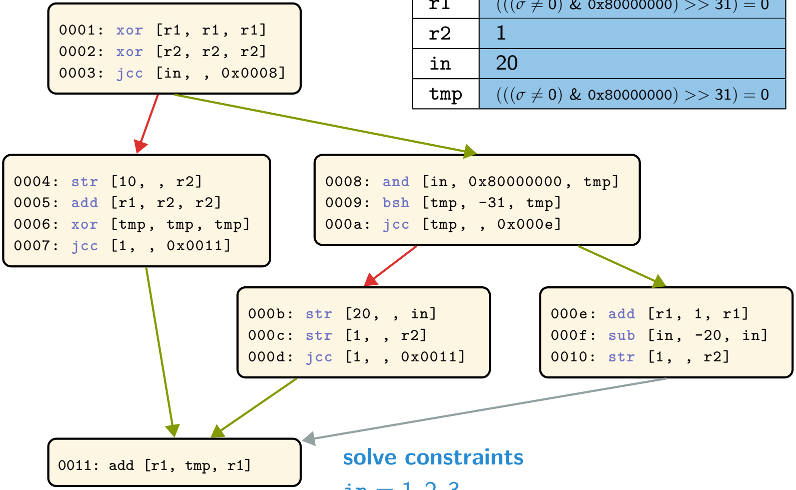


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0)$

Example

r1	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$
r2	1
in	20
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) = 0$

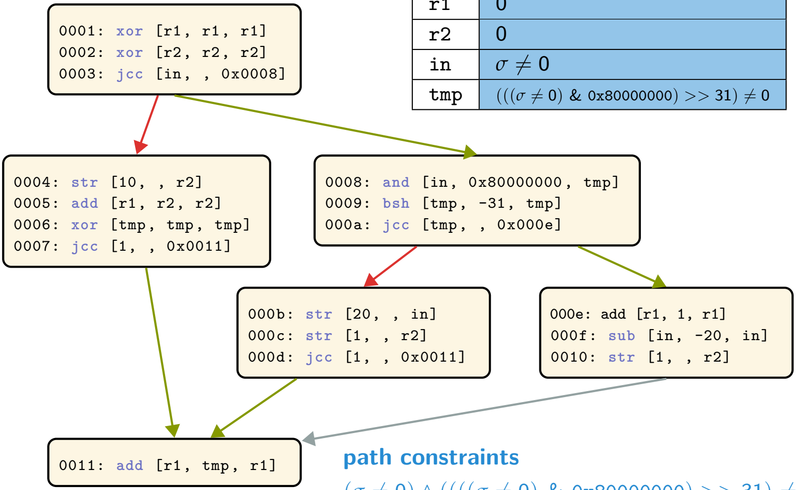


solve constraints

$\texttt{in} = 1, 2, 3, \ldots$

Example

r1	0
r2	0
in	$\sigma \neq 0$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$

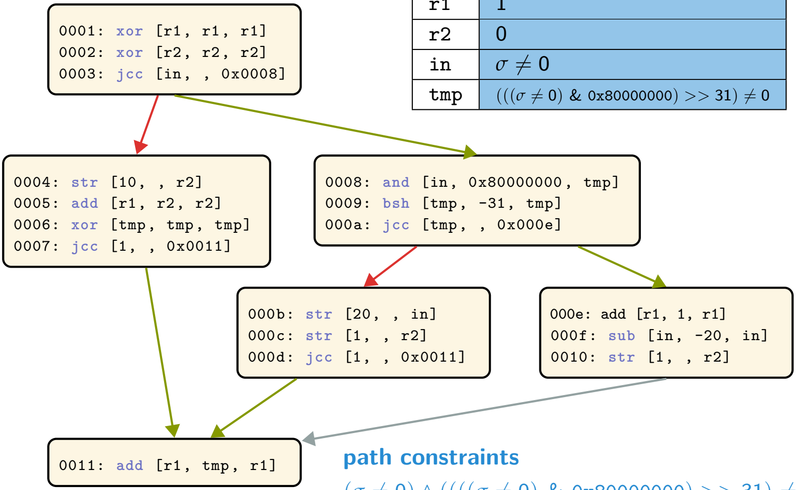


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$

Example

r1	1
r2	0
in	$\sigma \neq 0$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$

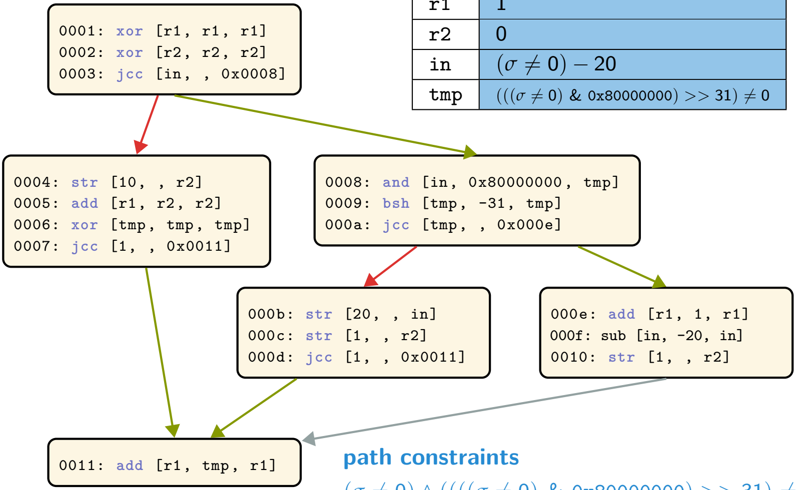


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$

Example

r1	1
r2	0
in	$(\sigma \neq 0) - 20$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$

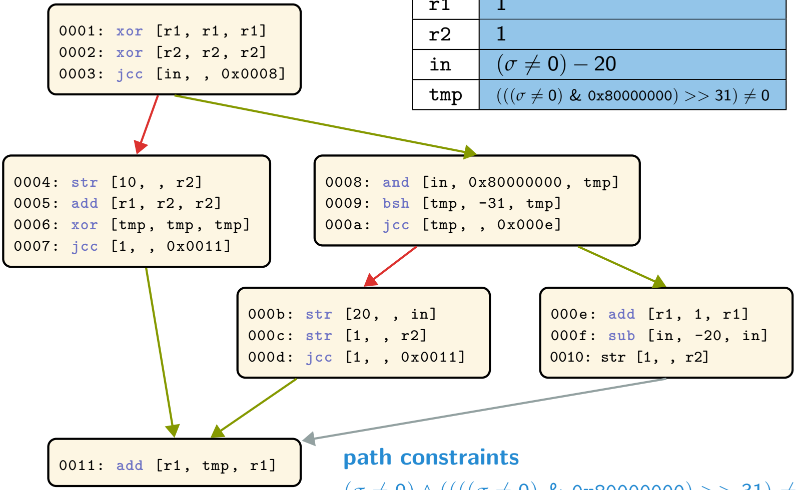


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$

Example

r1	1
r2	1
in	$(\sigma \neq 0) - 20$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$

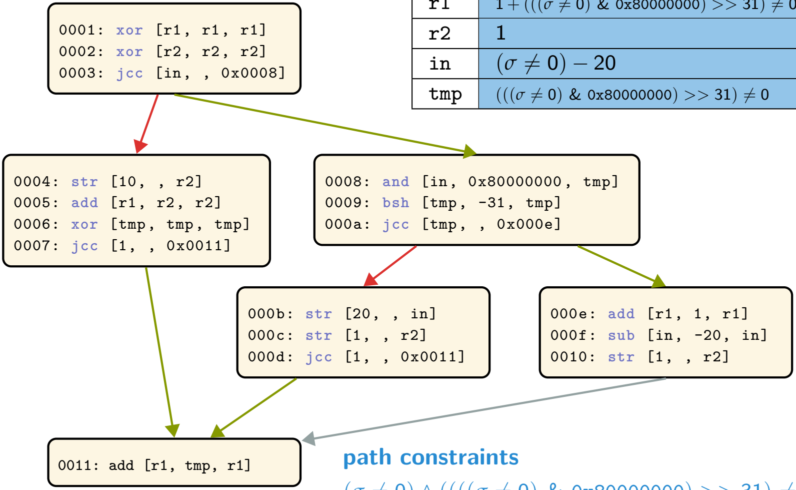


path constraints

$(\sigma \neq 0) \wedge ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$

Example

r1	$1 + (((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$
r2	1
in	$(\sigma \neq 0) - 20$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$

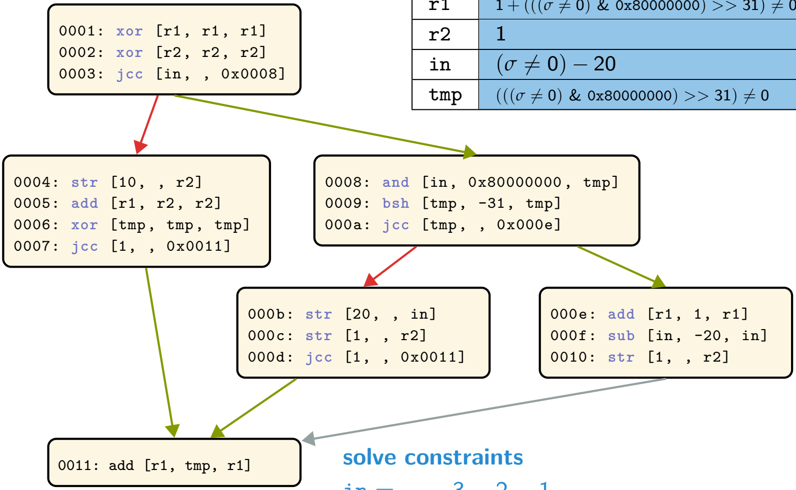


path constraints

$(\sigma \neq 0) \land ((((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$

Example

r1	$1 + (((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0)$
r2	1
in	$(\sigma \neq 0) - 20$
tmp	$(((\sigma \neq 0)\ \&\ 0x80000000) >> 31) \neq 0$



solve constraints

$\mathtt{in} = \ldots, -3, -2, -1$

Summary

- Applications
  - Automatic test-case generation
  - Vulnerability discovery (with fuzzing)
  - Malware analysis (explore "trigger sources")
- Loops?
  - Quickly result in **state-space explosion**
  - Possibly **model** common library functions
    - E.g. `strlen`, `strcpy`, `memcpy`, etc.

Summary

- Applications
  - Automatic test-case generation
  - Vulnerability discovery (with fuzzing)
  - Malware analysis (explore "trigger sources")
- Loops?
  - Quickly result in **state-space explosion**
  - Possibly **model** common library functions
    - E.g. `strlen`, `strcpy`, `memcpy`, etc.

Challenges

- State-space explosion
- Path (state) selection/prioritisation
- Environment modelling

Abstract interpretation

Why abstract interpretation?

Rice's Theorem

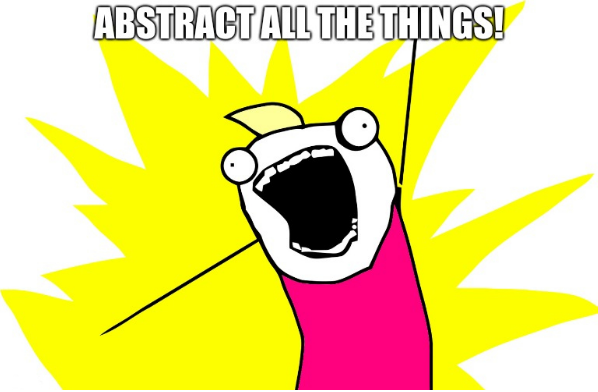
Any *non-trivial* property of program behaviour is undecidable

Why abstract interpretation?

Rice's Theorem

Any *non-trivial* property of program behaviour is undecidable

Solution?

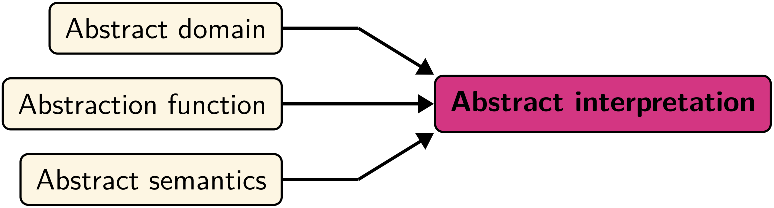


What is abstract interpretation?

Abstract the semantics of our program to make analysis possible

What is abstract interpretation?

Abstract the semantics of our program to make analysis possible



Abstract domain

Instead of operating on an (infinitely large) set of concrete values, operate on a smaller set of values that **approximates** the concrete values.

Abstract domain

Instead of operating on an (infinitely large) set of concrete values, operate on a smaller set of values that **approximates** the concrete values.

Domain	Values
Sign	$-$, 0, $+$
Interval	$[l, u]$, where l and u are integers and $l \leq u$

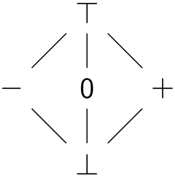
Abstract domain

Abstract values must form a lattice. The lattice is used when states are **joined** (merged) during execution.

Abstract domain

Abstract values must form a lattice. The lattice is used when states are **joined** (merged) during execution.

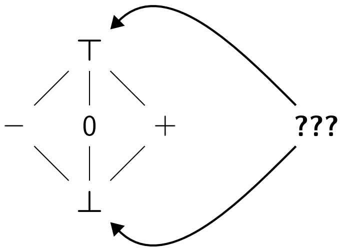
Sign domain



Abstract domain

Abstract values must form a lattice. The lattice is used when states are **joined** (merged) during execution.

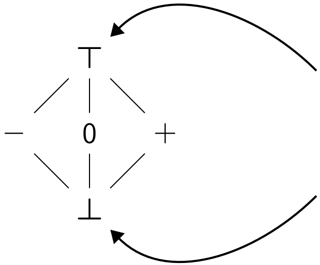
Sign domain



Abstract domain

Abstract values must form a lattice. The lattice is used when states are **joined** (merged) during execution.

Sign domain



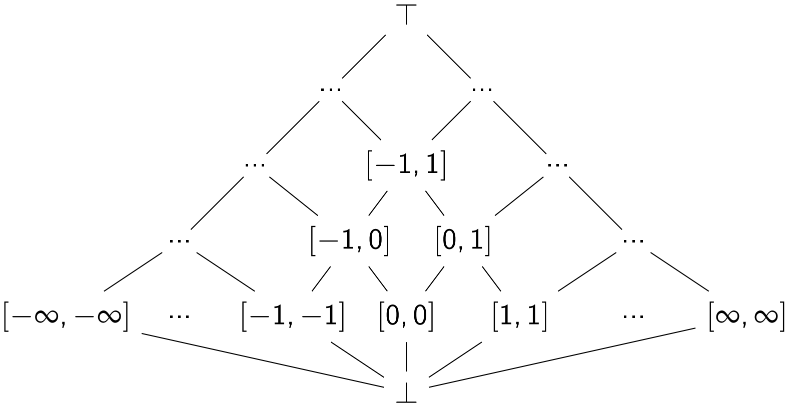
Top: don't know/any value

Bottom: uninitialised/empty set

Abstract domain

Abstract values must form a lattice. The lattice is used when states are **merged** during execution.

Interval domain



Abstraction function

Go from concrete $\rightarrow$ abstract

Abstraction function

Go from concrete $\rightarrow$ abstract

Sign domain

Concrete	Abstract
$\{\}$	$\bot$

Abstraction function

Go from concrete $\rightarrow$ abstract

Sign domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$+$

Abstraction function

Go from concrete $\rightarrow$ abstract

Sign domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$+$
$\{10, 5\}$	$+$

Abstraction function

Go from concrete $\rightarrow$ abstract

Sign domain

Concrete	Abstract
$\{\}$	$\perp$
$\{10\}$	$+$
$\{10, 5\}$	$+$
$\{-10, -5, -33\}$	$-$

Abstraction function

Go from concrete $\rightarrow$ abstract

Sign domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$+$
$\{10, 5\}$	$+$
$\{-10, -5, -33\}$	$-$
$\{10, 1, -5\}$	$\top$

Abstraction function

Go from concrete $\rightarrow$ abstract

Interval domain

Concrete	Abstract
$\{\}$	$\bot$

Abstraction function

Go from concrete $\rightarrow$ abstract

Interval domain

Concrete	Abstract
$\{\}$	$\perp$
$\{10\}$	$[10, 10]$

Abstraction function

Go from concrete $\rightarrow$ abstract

Interval domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$[10, 10]$
$\{10, 5\}$	$[5, 10]$

Abstraction function

Go from concrete $\rightarrow$ abstract

Interval domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$[10, 10]$
$\{10, 5\}$	$[5, 10]$
$\{-10, -5, -33\}$	$[-33, -5]$

Abstraction function

Go from concrete $\rightarrow$ abstract

Interval domain

Concrete	Abstract
$\{\}$	$\bot$
$\{10\}$	$[10, 10]$
$\{10, 5\}$	$[5, 10]$
$\{-10, -5, -33\}$	$[-33, -5]$
$\{10, 1, -5\}$	$[-5, 10]$

Abstract semantics

Give **meaning** to our program in the abstract domain

Abstract semantics

Give **meaning** to our program in the abstract domain

Sign domain

```
add [DWORD r1, DWORD r2, DWORD r3]
```

		r2		
		$-$	0	$+$
	$-$	$-$	$-$	$\top$
r1	0	$-$	0	$+$
	$+$	$\top$	$+$	$+$

Abstract semantics

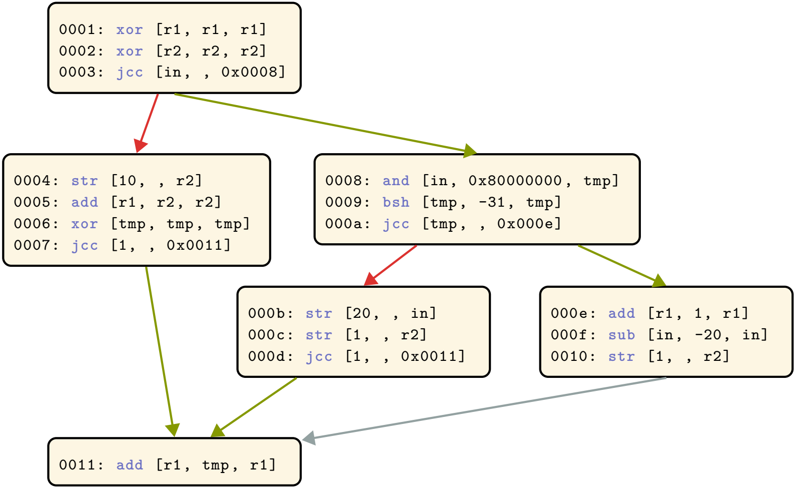
Give **meaning** to our program in the abstract domain

Interval domain

```
add [DWORD r1, DWORD r2, DWORD r3]
```

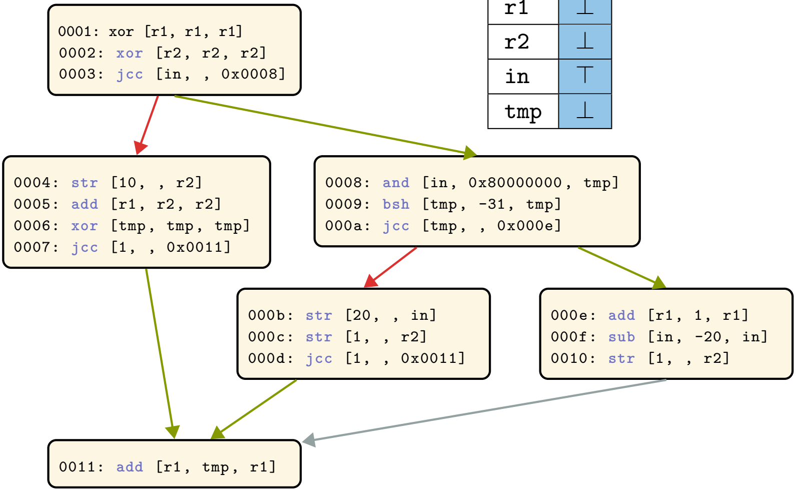
		r2
		$[x, y]$
r1	$[a, b]$	$[a+x, b+y]$

Example – sign analysis



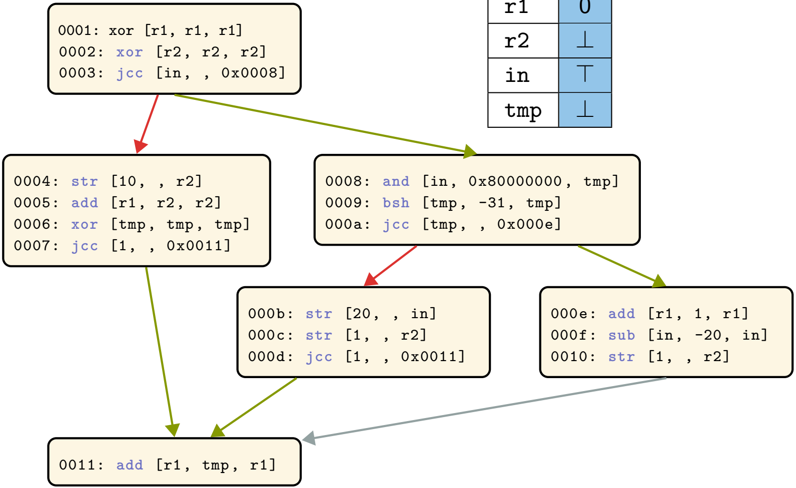
Example – sign analysis

r1	⊥
r2	⊥
in	⊤
tmp	⊥



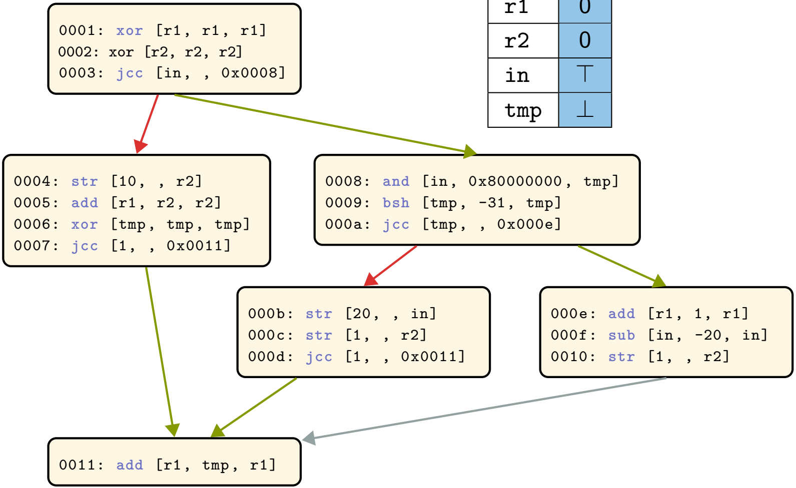
Example – sign analysis

r1	0
r2	⊥
in	⊤
tmp	⊥



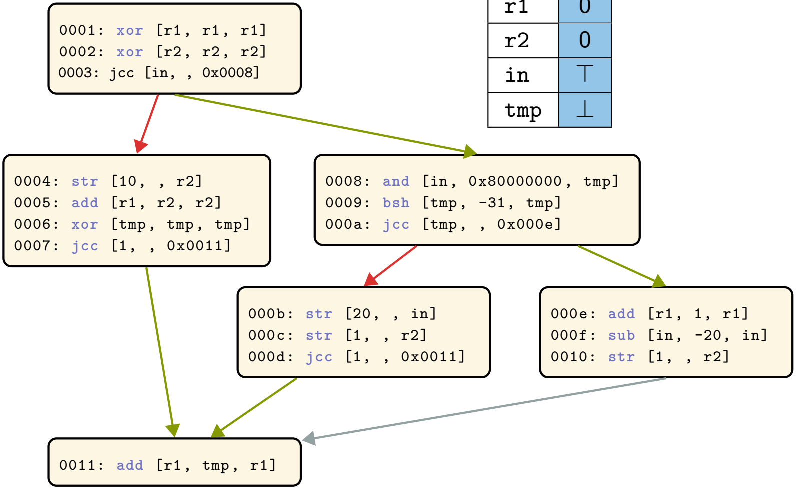
Example – sign analysis

r1	0
r2	0
in	⊤
tmp	⊥



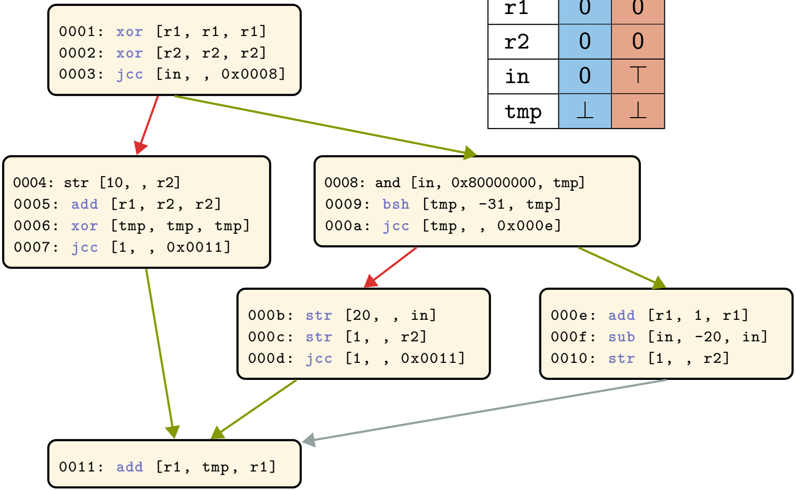
Example – sign analysis

r1	0
r2	0
in	⊤
tmp	⊥

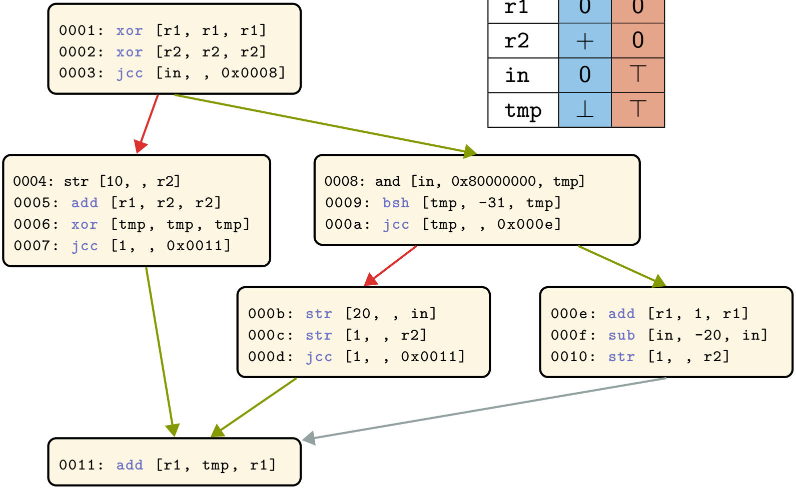


Example – sign analysis

r1	0	0
r2	0	0
in	0	⊤
tmp	⊥	⊥



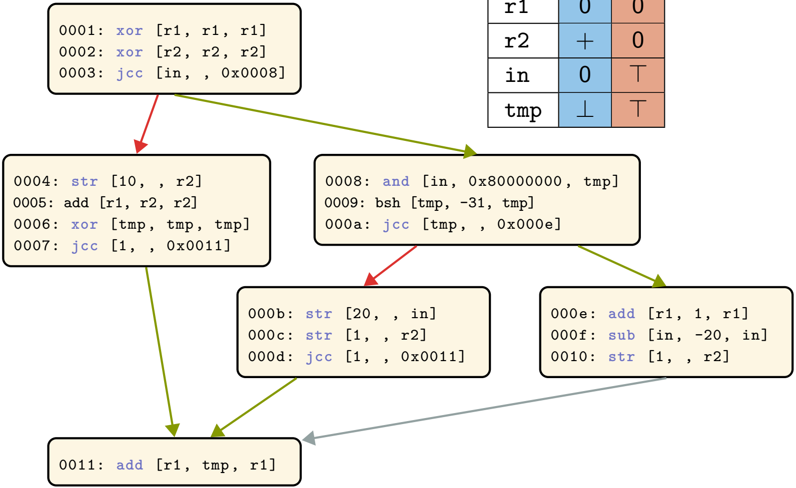
Example – sign analysis



r1	0	0
r2	+	0
in	0	⊤
tmp	⊥	⊤

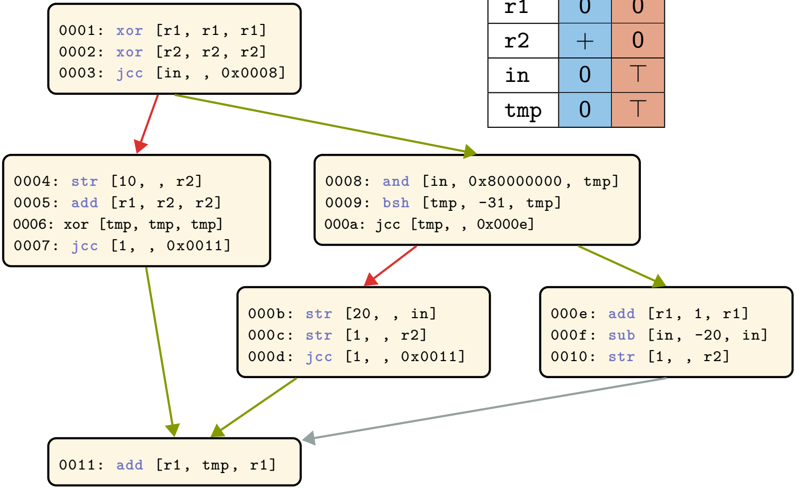
Example – sign analysis

r1	0	0
r2	+	0
in	0	⊤
tmp	⊥	⊤



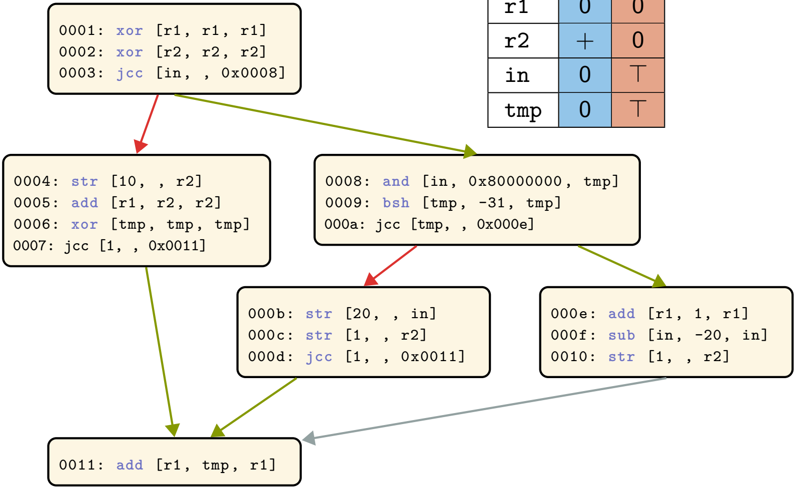
Example – sign analysis

r1	0	0
r2	+	0
in	0	⊤
tmp	0	⊤

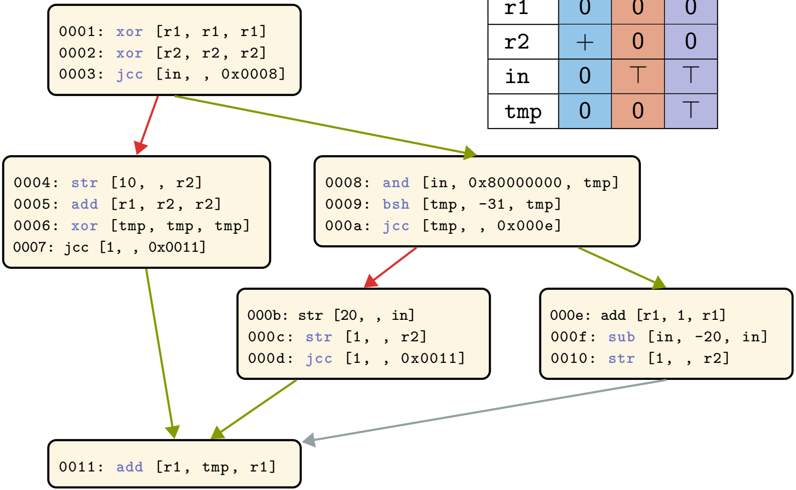


Example – sign analysis

r1	0	0
r2	+	0
in	0	$\top$
tmp	0	$\top$



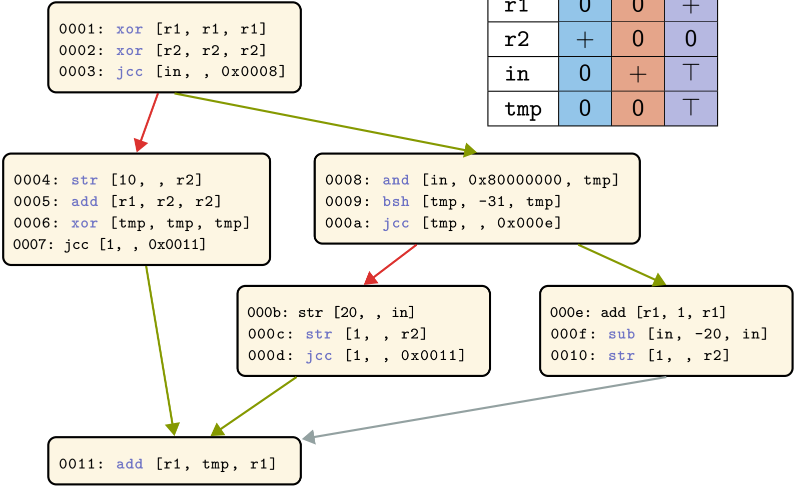
Example – sign analysis



r1	0	0	0
r2	+	0	0
in	0	⊤	⊤
tmp	0	0	⊤

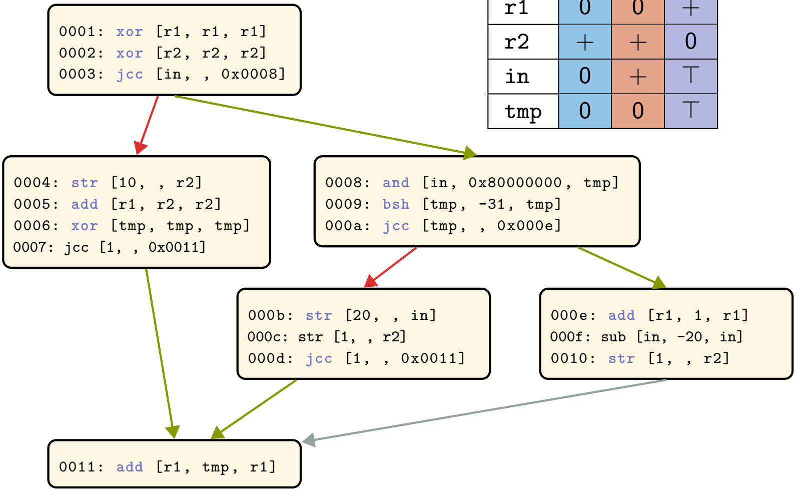
Example – sign analysis

r1	0	0	+
r2	+	0	0
in	0	+	$\top$
tmp	0	0	$\top$



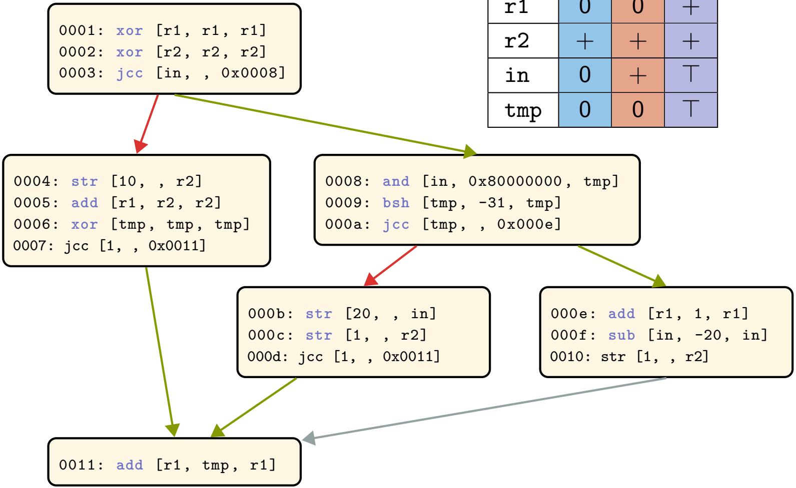
Example – sign analysis

r1	0	0	+
r2	+	+	0
in	0	+	$\top$
tmp	0	0	$\top$



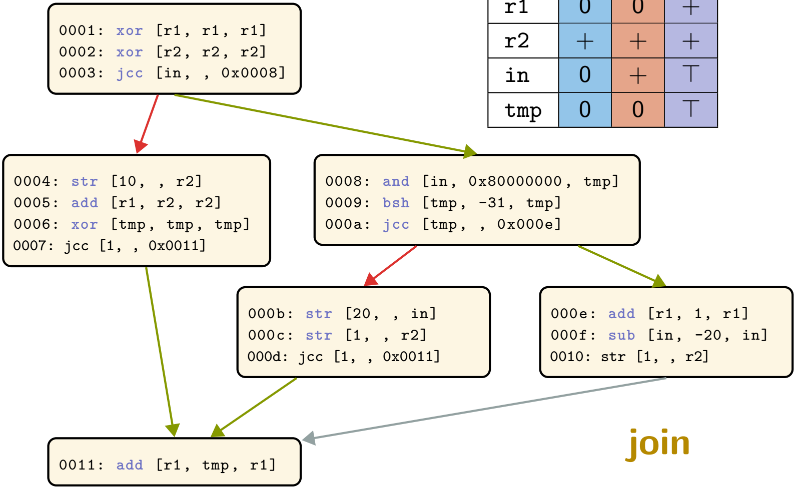
Example – sign analysis

r1	0	0	+
r2	+	+	+
in	0	+	$\top$
tmp	0	0	$\top$

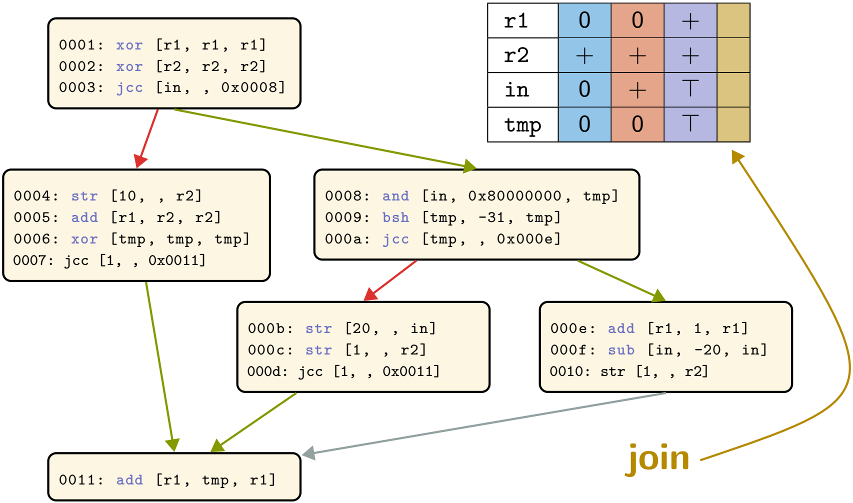


Example – sign analysis

r1	0	0	+
r2	+	+	+
in	0	+	⊤
tmp	0	0	⊤



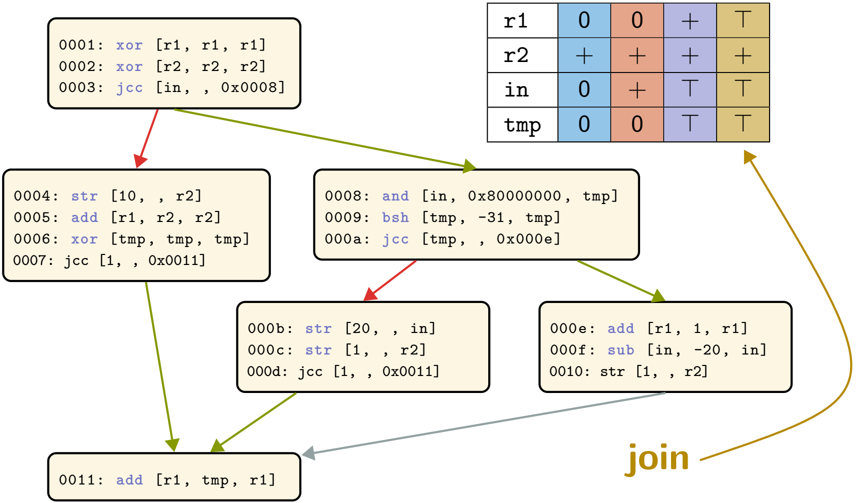
Example – sign analysis



r1	0	0	+	
r2	+	+	+	
in	0	+	$\top$	
tmp	0	0	$\top$	

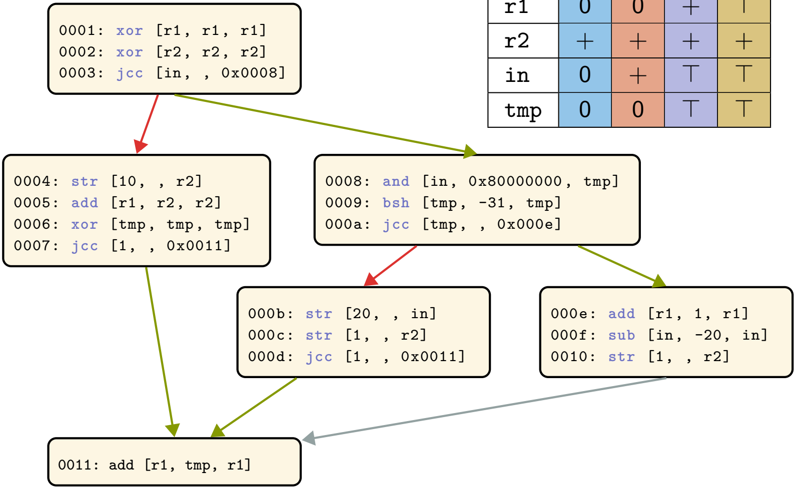
Example – sign analysis

r1	0	0	+	⊤
r2	+	+	+	+
in	0	+	⊤	⊤
tmp	0	0	⊤	⊤



Example – sign analysis

r1	0	0	$+$	$\top$
r2	$+$	$+$	$+$	$+$
in	0	$+$	$\top$	$\top$
tmp	0	0	$\top$	$\top$



Summary

- Abstract interpretation is hard on binary code
  - Could we have done better with a more complex abstract domain (e.g. intervals)?
- Accuracy vs. cost
- Loops?
  - Employ **widening** operator

Summary

- Abstract interpretation is hard on binary code
  - Could we have done better with a more complex abstract domain (e.g. intervals)?
- Accuracy vs. cost
- Loops?
  - Employ **widening** operator

Challenges

- How do we design a suitable abstract domain?
- How do we accurately represent the semantics of our instruction set?

Conclusion

Summary

- Given you a taste of what techniques exist

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance
  - Thanks to DARPA Cyber Grand Challenge

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance
  - Thanks to DARPA Cyber Grand Challenge
- Still a lot of work to go

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance
  - Thanks to DARPA Cyber Grand Challenge
- Still a lot of work to go
  - How do we deal with state-space explosion?

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance
  - Thanks to DARPA Cyber Grand Challenge
- Still a lot of work to go
  - How do we deal with state-space explosion?
  - How do we scale these techniques?

Summary

- Given you a taste of what techniques exist
- Binary program analysis is undergoing a renaissance
  - Thanks to DARPA Cyber Grand Challenge
- Still a lot of work to go
  - How do we deal with state-space explosion?
  - How do we scale these techniques?
  - Not many open-source tools (symbolic execution is the exception)


I wrote a vulnerability scanner that abstracts all the predicates in a binary, traverses the callgraph and generates phormulaes to run then with a SMT solver.
I found 1 vuln in 3 days with this tool.
He wrote a dumb ass fuzzer and found 5 vulns in 1 day.
Good thing I'm not a n00b like that guy.

Thank you!