SEAN HEELAN

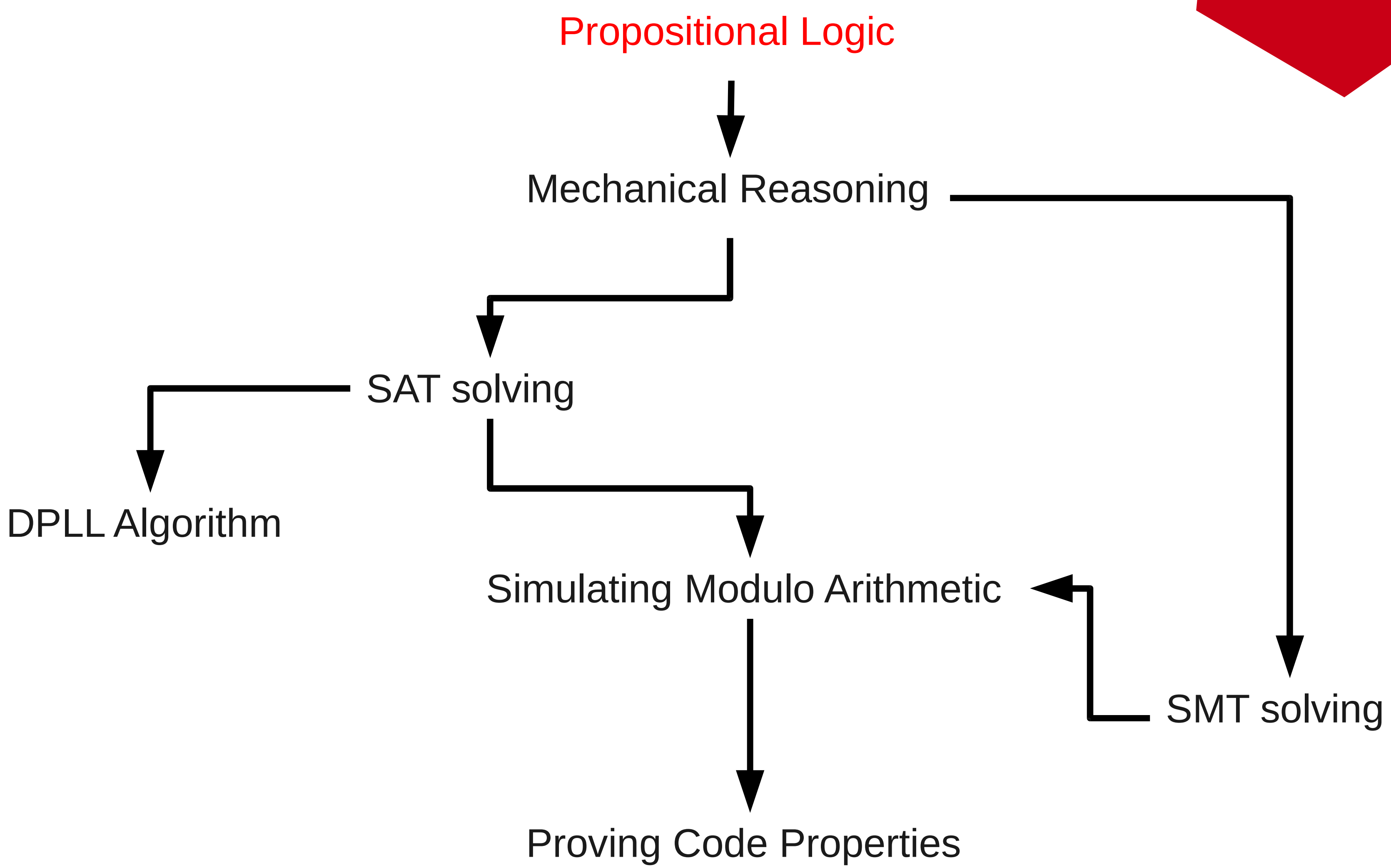
THE (IN)COMPLETE GUIDE TO CODE ANALYSIS CARPENTRY

Or how to avoid braining yourself when handed an SMT solving hammer

IMMUNITY INC.

```
void* ArrayBuffer::tryAllocate(unsigned numElements,
                               unsigned elementByteSize)
{
    void* result;
    // Do not allow 32-bit overflow of the total size
    if (numElements) {
    unsigned totalSize = numElements * elementByteSize;
    if (totalSize / numElements != elementByteSize)
        return 0;
    }
    if (WTF::tryFastCalloc(numElements,
                           elementByteSize).getValue(result))
        return result;
    return 0;
}
```

PART I: DOWN THE RABBIT HOLE



PROPOSITIONAL LOGIC

Propositional variables p, q, r, s ...

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->
Formation rules (grammar)	~(a ^ b) -> c, b ~ c

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->
Formation rules (grammar)	~(a ^ b) -> c, b ~ c
Truth Functional	“True”/”False”

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->
Formation rules (grammar)	~(a ^ b) -> c, b ~ c
Truth Functional	“True”/”False”
Axioms/Rules of Inference	

PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->
Formation rules (grammar)	~(a ^ b) -> c, b ~ c
Truth Functional	“True”/”False”
Axioms/Rules of Inference	

Mechanical “Reasoning”

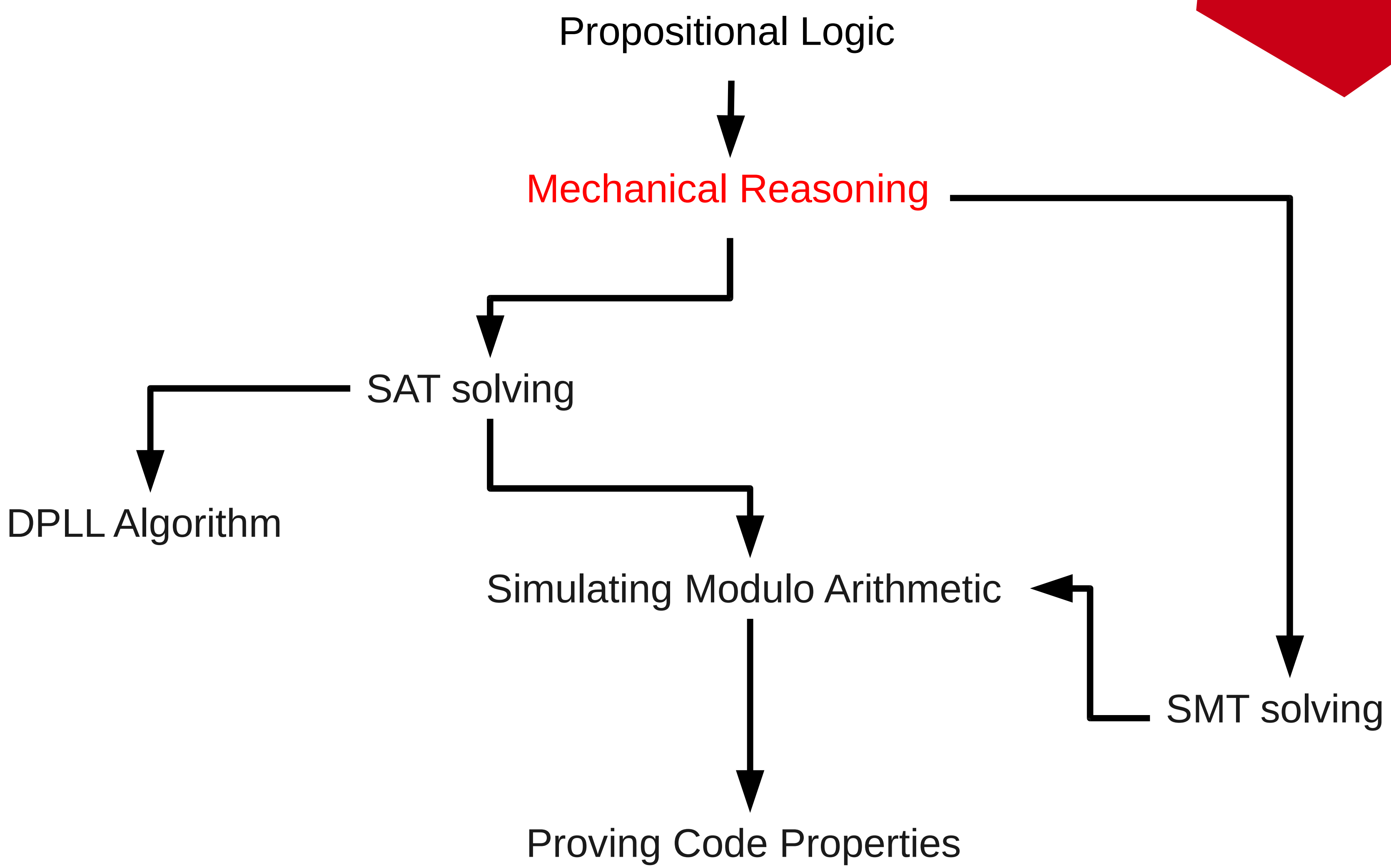
PROPOSITIONAL LOGIC

Propositional variables	p, q, r, s ...
Punctuation	(,)
Logical connectives	^, v, ~, ->
Formation rules (grammar)	~(a ^ b) -> c, b ~ c
Truth Functional	“True”/”False”
Axioms/Rules of Inference	

Mechanical “Reasoning”

[1] This is incomplete but sufficient for our purposes. Wikipedia does a relatively good job for a lot of what we'll cover. Otherwise see http://www.unprotectedhex.com/psv/ for books.

Part 1: Down the Rabbit Hole



MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

^, v, ~, -> are named functions over this domain.

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

^, v, ~, -> are named functions over this domain.

These functions are simply the functions we thought worth naming.

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

^, v, ~, -> are named functions over this domain.

These functions are simply the functions we thought worth naming.

p	f^1	f^2	f^3	f^4
T	T	T	F	F
F	T	F	T	F

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

^, v, ~, -> are named functions over this domain.

These functions are simply the functions we thought worth naming.

p	f^1	f^2	f^3	f^4
T	T	T	F	F
F	T	F	T	F

MECHANICAL REASONING

Truth functional -> the values for propositional variables selected from the boolean domain T/F.

True/False are arbitrary constants – no concept of philosophical truth or falsity. Could be X/Y.

^, v, ~, -> are named functions over this domain.

These functions are simply the functions we thought worth naming.

(Truth Table)

p	~p
T	F
F	T

p	f^1	f^2	f^3	f^4
T	T	T	F	F
F	T	F	T	F

Mechanical Reasoning (cont)

A 'rewriting' system for any arbitrary propositional formula

MECHANICAL REASONING (CONT)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b)) under a=T, b=F

Mechanical Reasoning (cont)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b) under a=T, b=F

a=T, b=F

(a v b) -> (a ^ b)

MECHANICAL REASONING (CONT)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b) under a=T, b=F

a=T, b=F

(a v b) -> (a ^ b)
(T v F) -> (T ^ F)

MECHANICAL REASONING (CONT)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b)) under a=T, b=F

a=T, b=F

(a v b) -> (a ^ b)
(T v F) -> (T ^ F)
T -> F

MECHANICAL REASONING (CONT)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b) under a=T, b=F

a=T, b=F

(a v b) -> (a ^ b)
(T v F) -> (T ^ F)
T -> F
F

Mechanical Reasoning (cont)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b) under a=T, b=F

X is satisfied if it reduces to T under any model

X is valid if it reduces to T under all models

a=T, b=F

(a v b) -> (a ^ b)
(T v F) -> (T ^ F)
T -> F
F

MECHANICAL REASONING (CONT)

A 'rewriting' system for any arbitrary propositional formula

Evaluate X:((a v b) -> (a ^ b) under a=T, b=F

a=T, b=F

(a v b) -> (a ^ b)
(T v F) -> (T ^ F)
T -> F
F

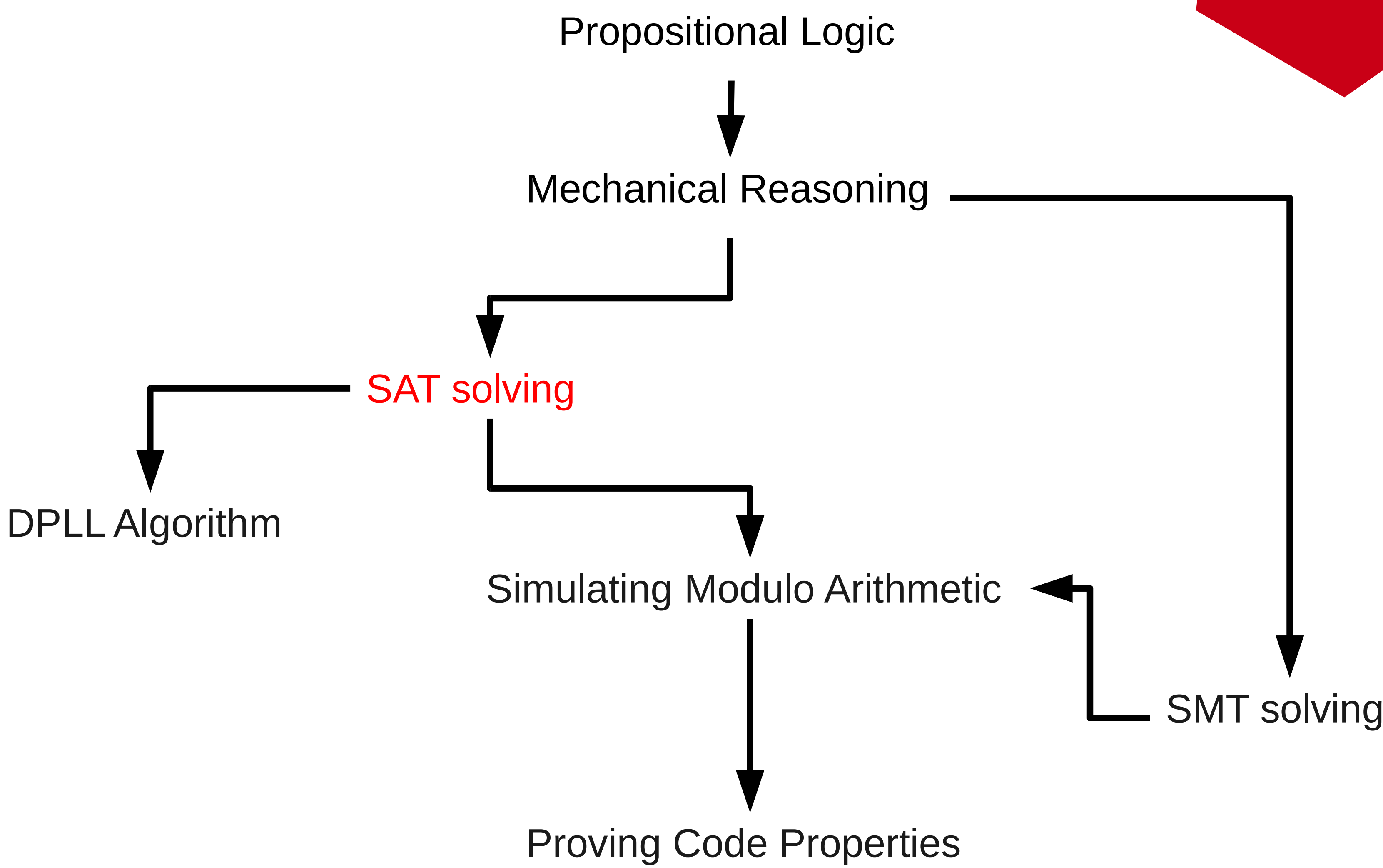
X is satisfied if it reduces to T under any model

X is valid if it reduces to T under all models

Boolean satisfiability problem – *Find such an assignment or prove that none exists*

→ SAT solving

PART I: DOWN THE RABBIT HOLE



SAT SOLVING

SAT is a decidable problem

SAT SOLVING

SAT is a decidable problem

The solution set is recursive (algorithm that terminates in finite time)

SAT SOLVING

SAT is a decidable problem

The solution set is recursive (algorithm that terminates in finite time)

Algorithm? Dumb solution – treat X as a n-ary function, build a truth table

SAT SOLVING

SAT is a decidable problem

The solution set is recursive (algorithm that terminates in finite time)

Algorithm? Dumb solution – treat X as a n-ary function, build a truth table

a	b	a v b	a ^ b	(a v b) -> (a ^ b)
T	T	T	T	T
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T

SAT SOLVING

SAT is a decidable problem → The solution set is recursive (algorithm that terminates in finite time)

Algorithm? Dumb solution – treat X as a n-ary function, build a truth table

a	b	a v b	a ^ b	(a v b) -> (a ^ b)
T	T	T	T	T
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T

Quite obviously won't scale, table size of 2^n but demonstrates an entirely mechanical process that is guaranteed to terminate.

SAT SOLVING

SAT is a decidable problem

The solution set is recursive (algorithm that terminates in finite time)

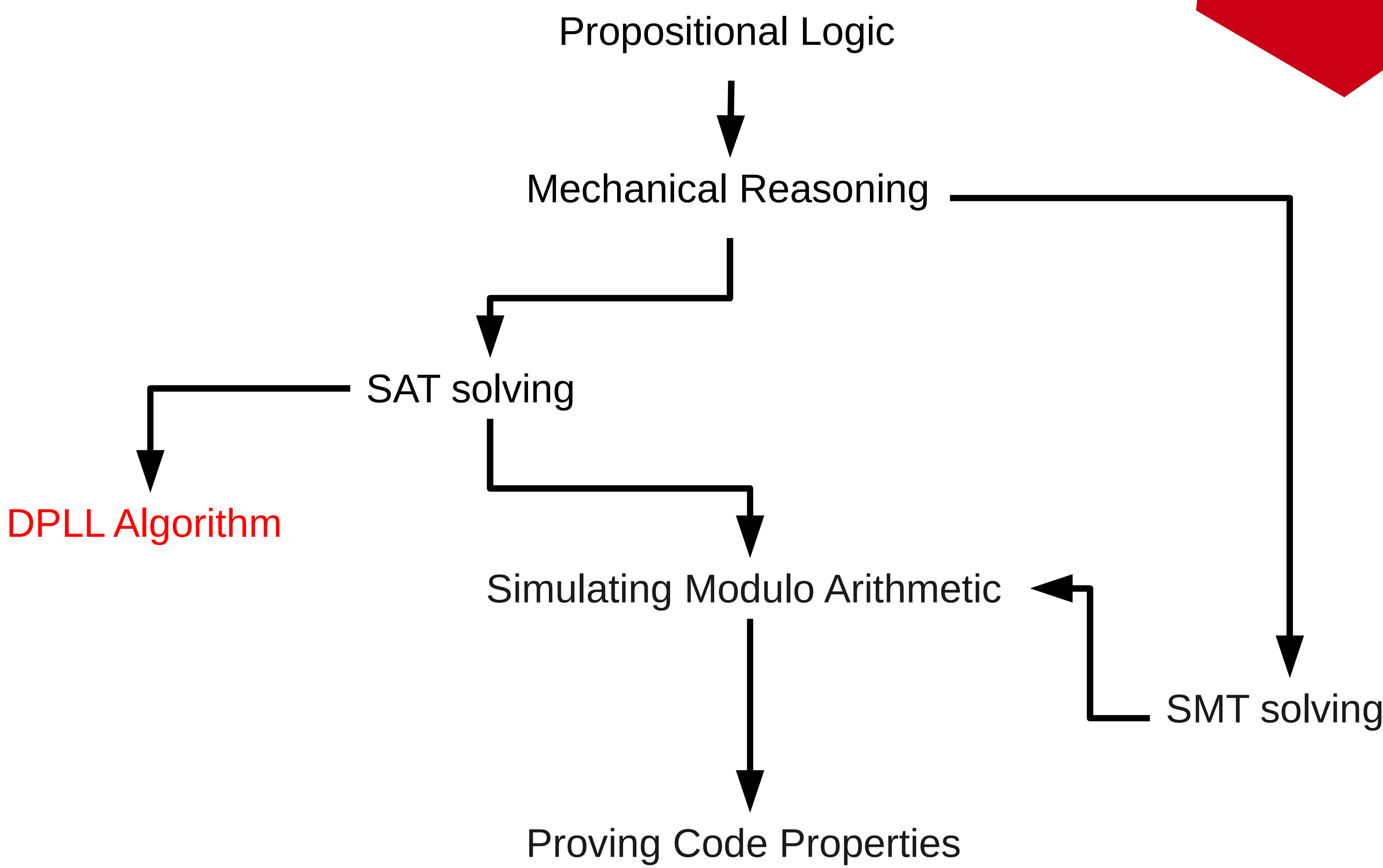
Algorithm? Dumb solution – treat X as a n-ary function, build a truth table

a	b	a v b	a ^ b	(a v b) -> (a ^ b)
T	T	T	T	T
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T

Quite obviously won't scale, table size of 2^n but demonstrates an entirely mechanical process that is guaranteed to terminate.

DPLL Algorithm

Part 1: Down the Rabbit Hole



DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

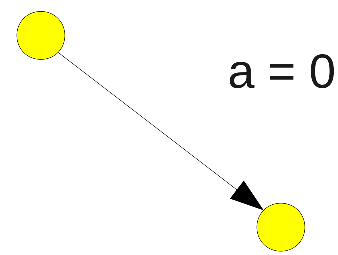
We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

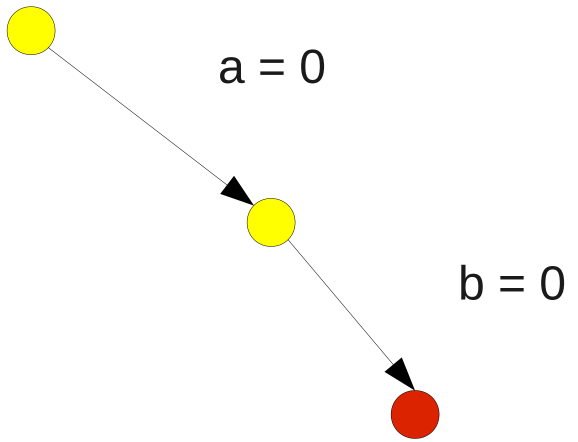


DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

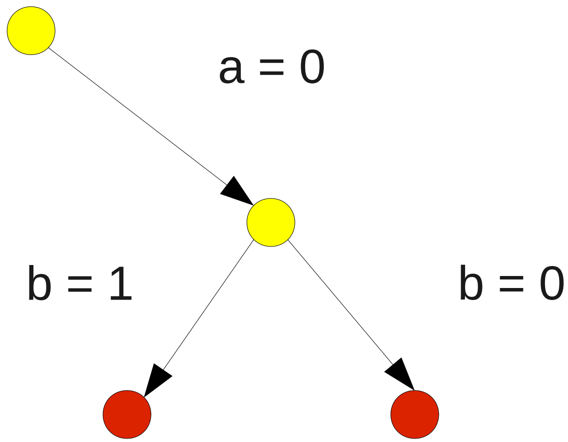


DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

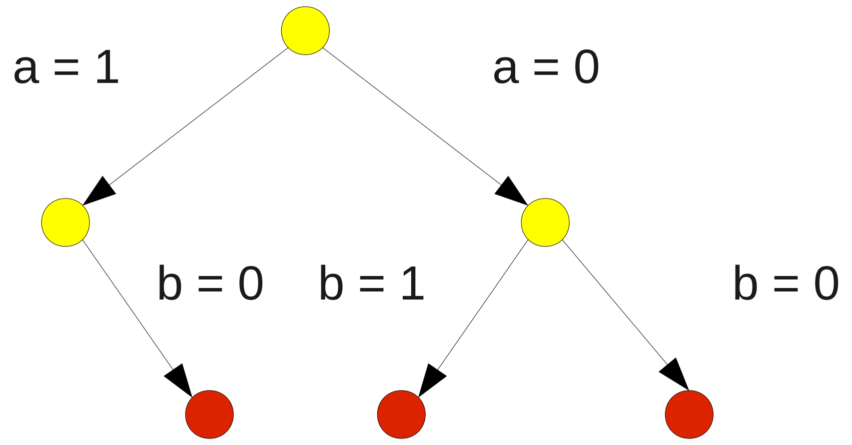


DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

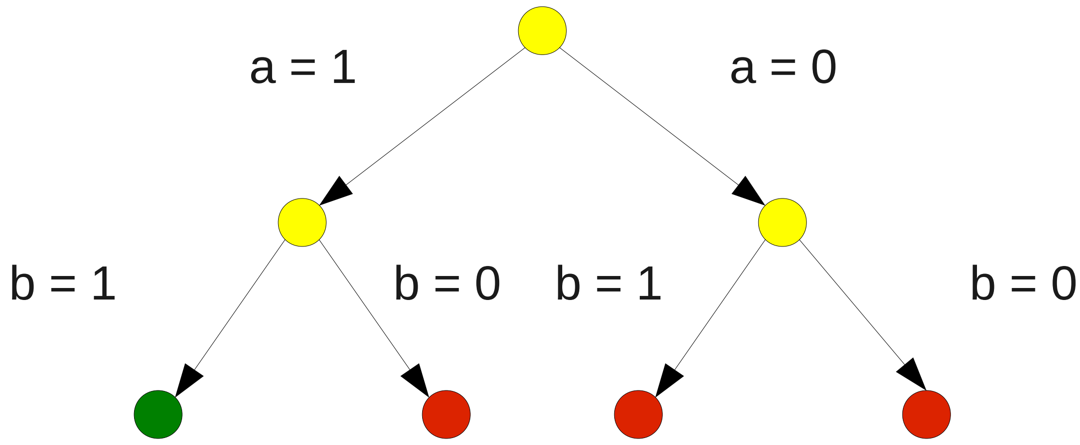


DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)

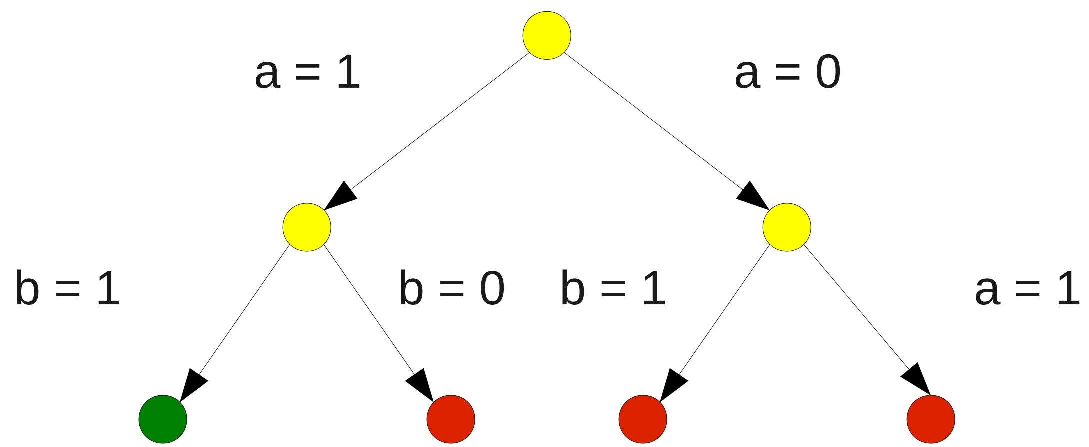


DPLL ALGORITHM

SAT is in the NP-Complete complexity class so our worst case is always going to be exponential in the size of our input.

DPLL Algorithm – The basis for modern SAT solvers

We are essentially doing a state space search over a binary tree (each vertex represents an assignment of a value to a variable) e.g for (a ^ b)



DPLL and other optimisations work by attempting to prune the tree and limit the cases we need to consider

DPLL ALGORITHM (CONT)

Convert formula to CNF first e.g (a v b) ^ (c v d) ^ (~d v a)

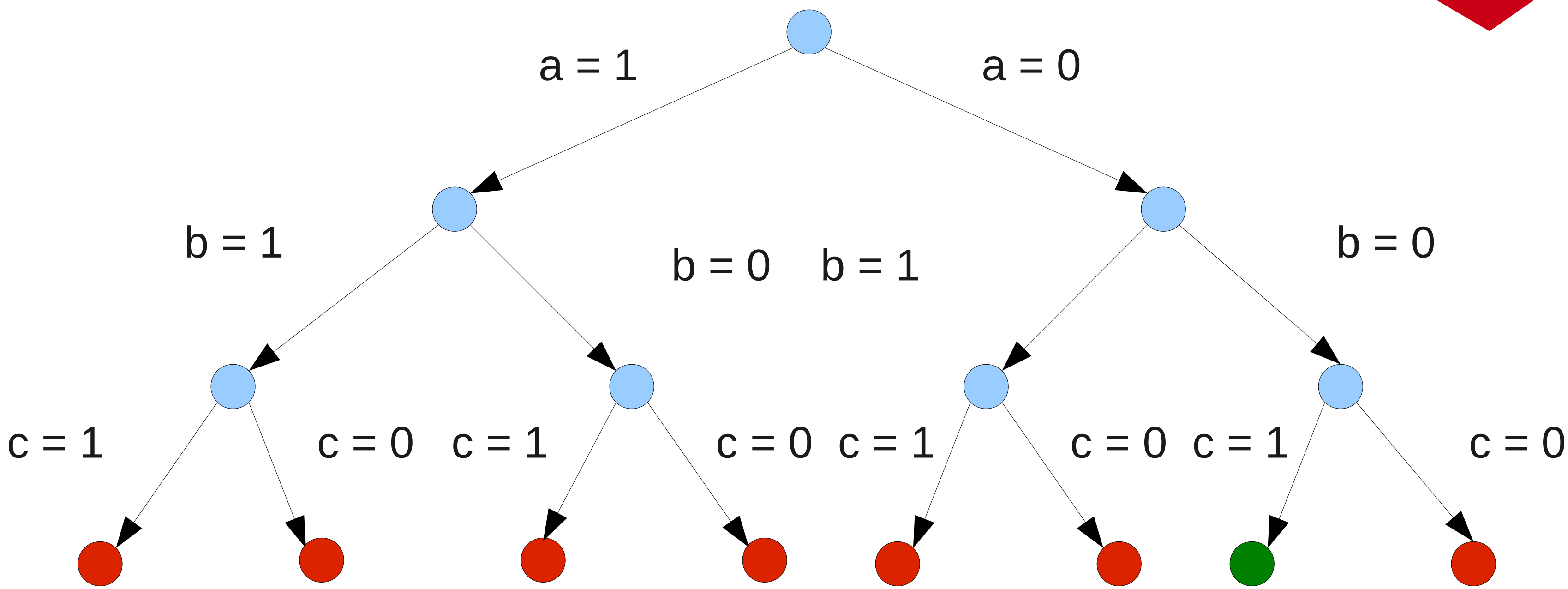
DPLL Algorithm (cont)

Convert formula to CNF first e.g (a v b) ^ (c v d) ^ (~d v a)

Deduction rule – For each clause if there is one unassigned variable *p* and all others are False then *p* must be True (if this isn't possible then the formula is UNSAT)

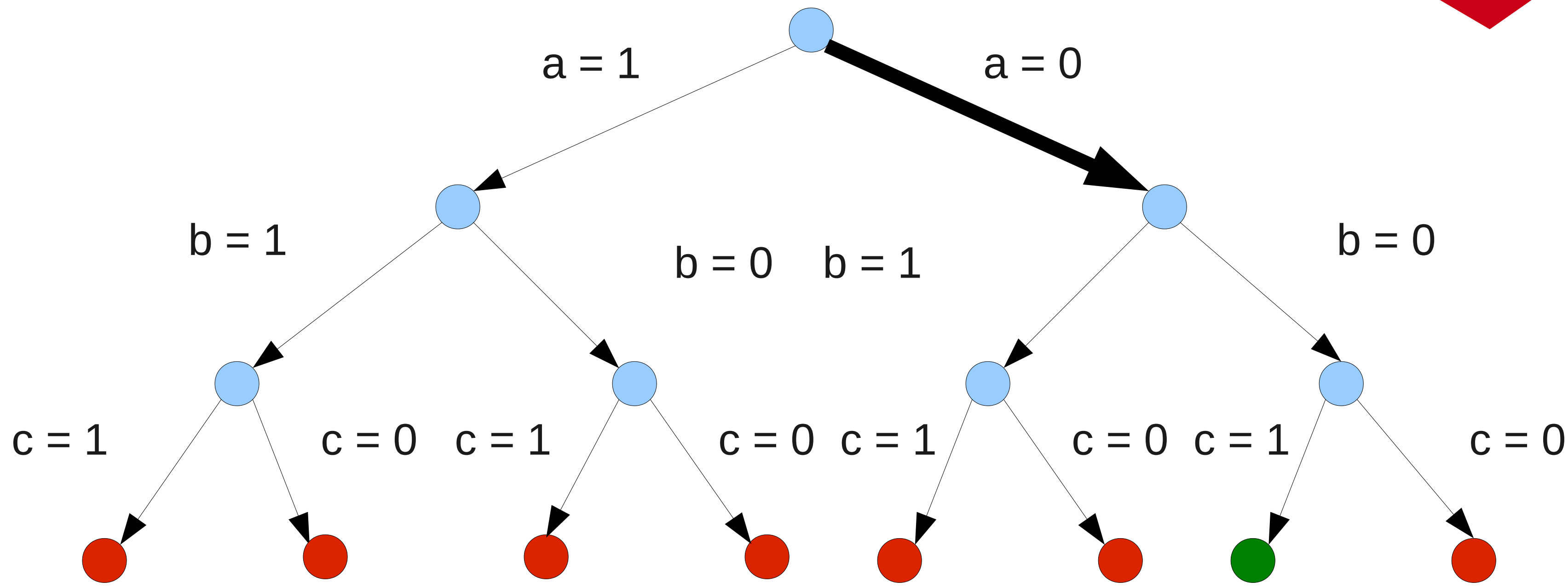
DPLL ALGORITHM (CONT)

Unit clause deduction on (~a) v (~b) v (a v b v c)



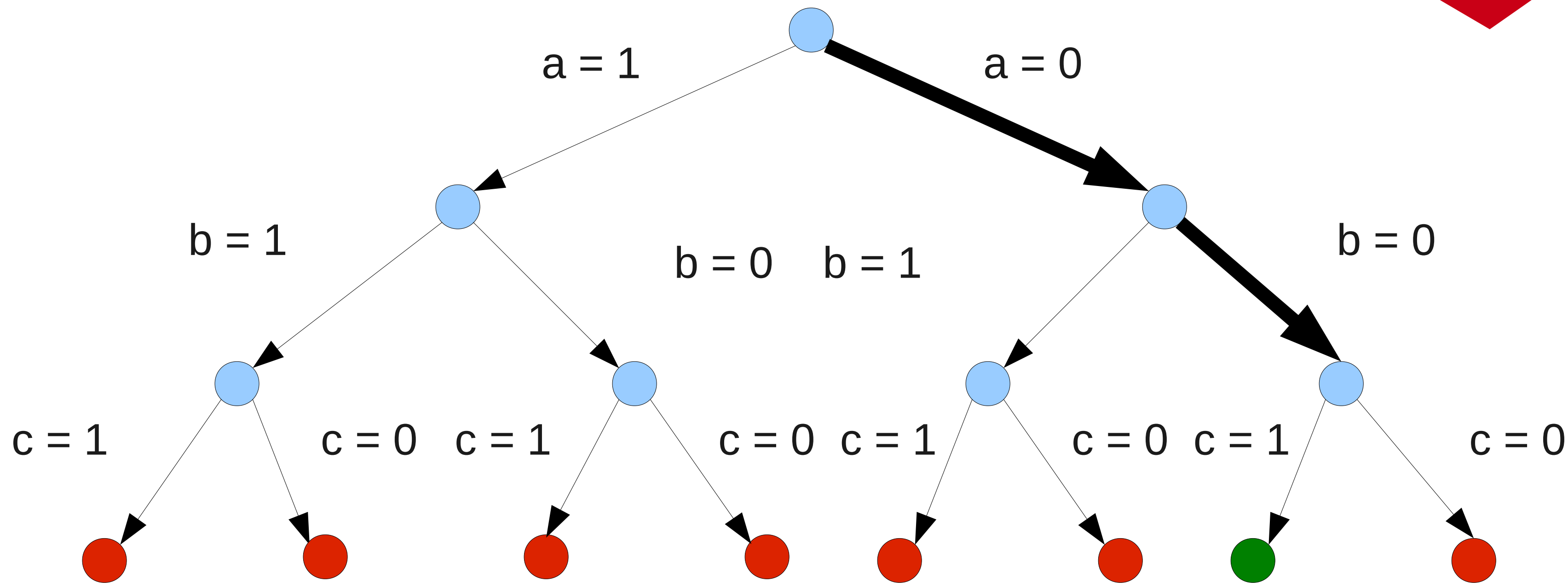
DPLL Algorithm (cont)

Unit clause deduction on (~a) v (~b) v (a v b v c)



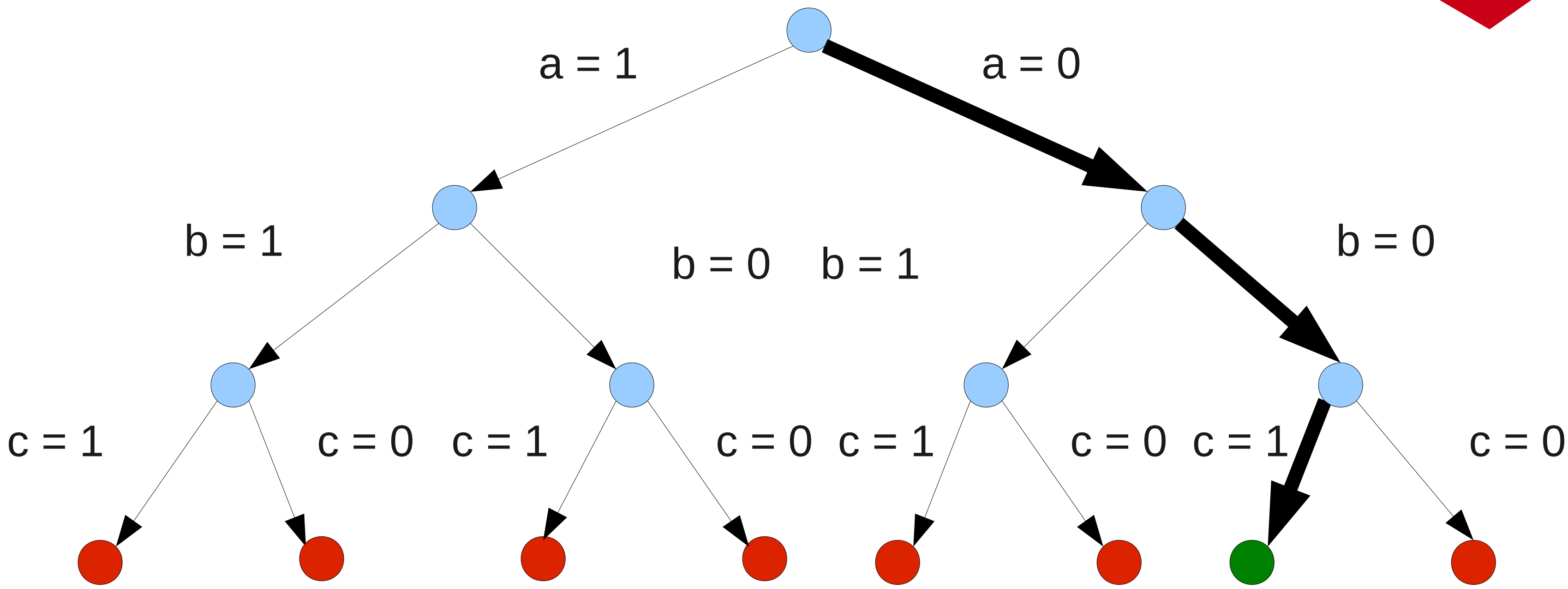
DPLL ALGORITHM (CONT)

Unit clause deduction on (~a) v (~b) v (a v b v c)



DPLL Algorithm (cont)

Unit clause deduction on (~a) v (~b) v (a v b v c)



No search required in this case

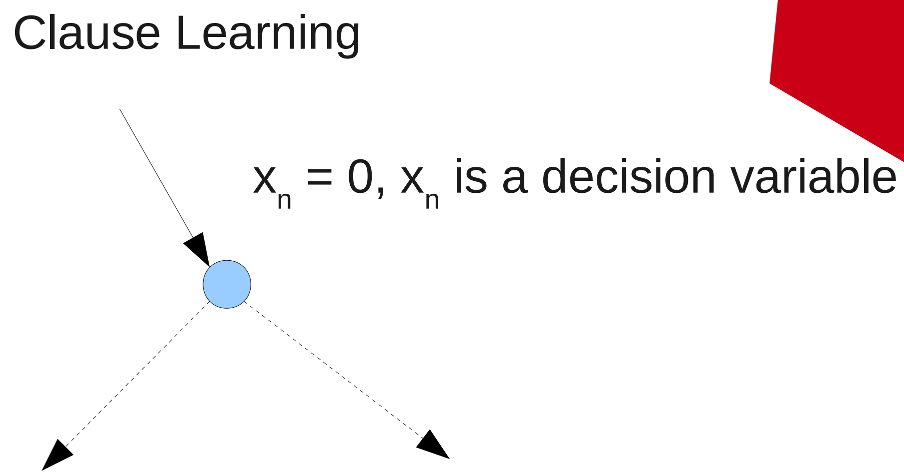
DPLL ALGORITHM (CONT)

Convert formula to CNF first e.g (a v b) ^ (c v d) ^ (~d v a)

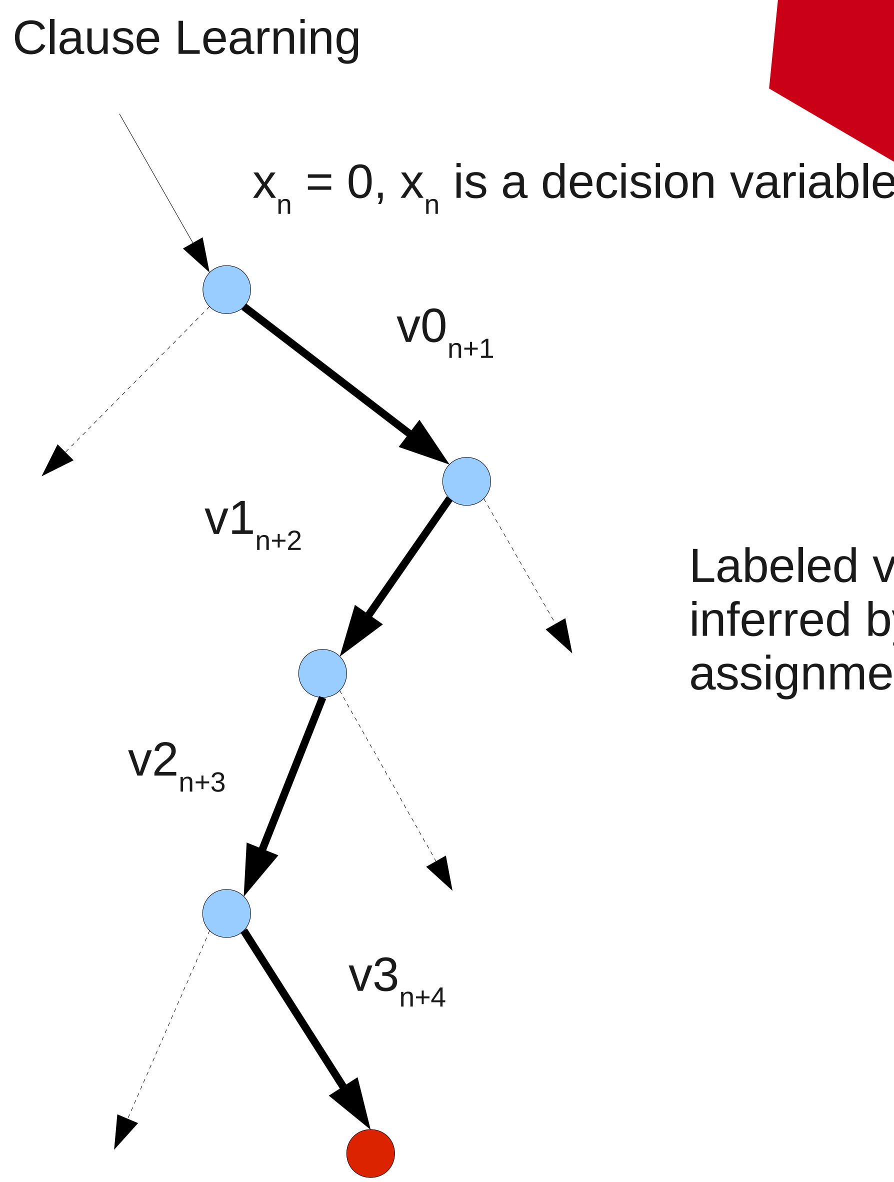
Deduction rule – For each clause if there is one unassigned variable *p* and all others are False then *p* must be True (if this isn't possible then the formula is UNSAT)

Clause learning – If a clause evaluates to False backtrack to the assignment (before any deductive assignments) that is the cause and add its negation to the formula to avoid it in the future

DPLL Algorithm (cont)



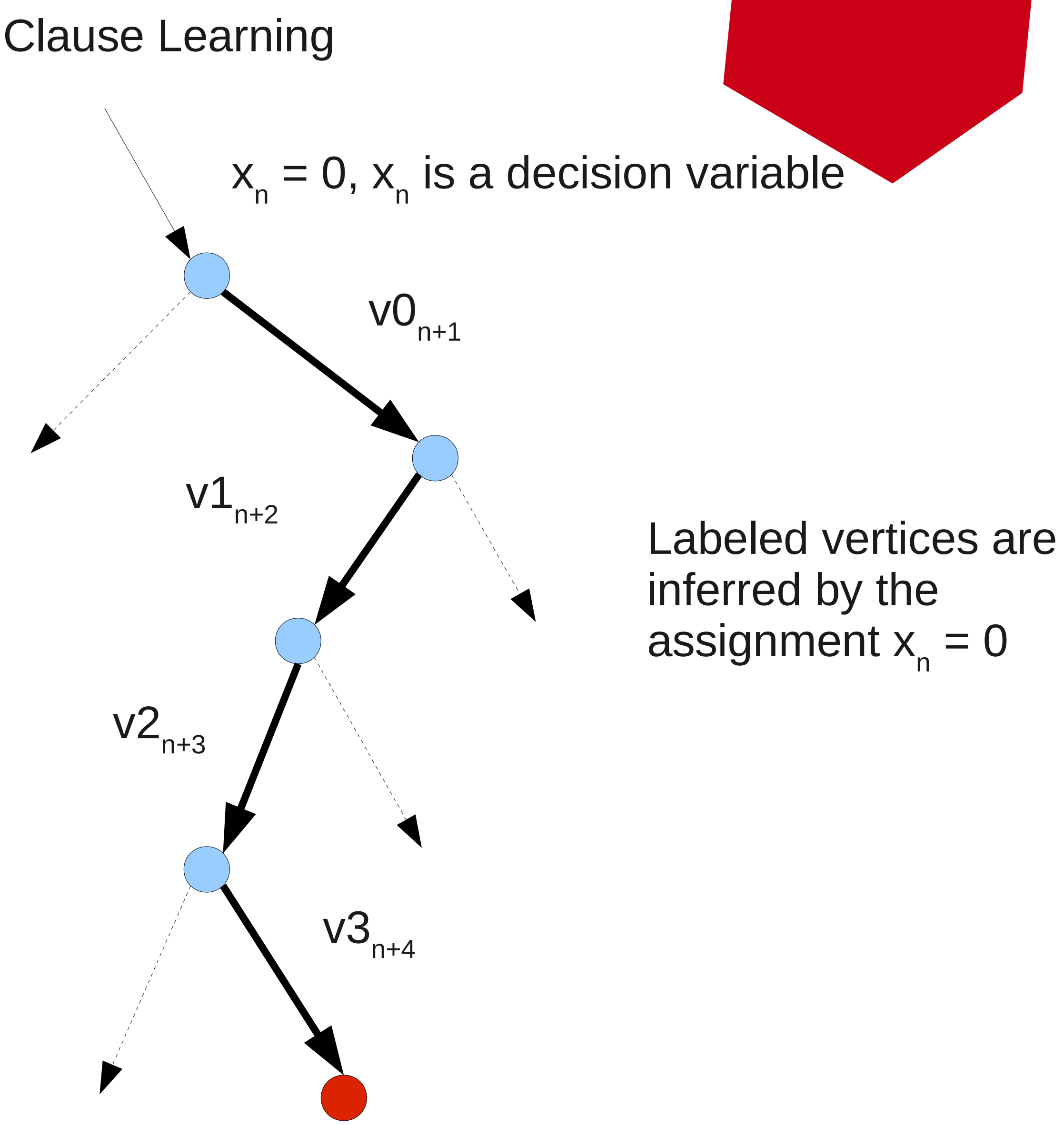
DPLL Algorithm (cont)



Labeled vertices are inferred by the assignment $x_n = 0$

DPLL Algorithm (cont)

The learned clause is (x)

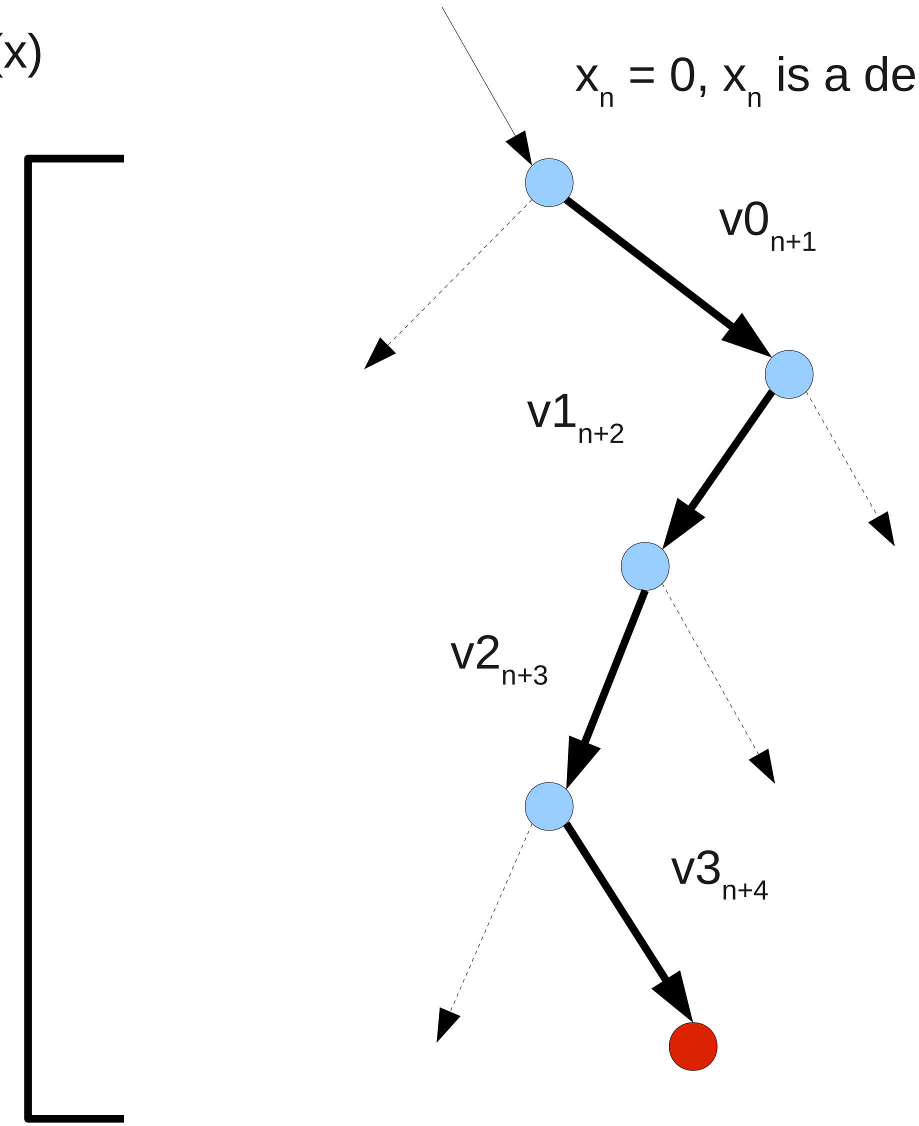


DPLL ALGORITHM (CONT)

Clause Learning

The learned clause is (x)

$x_n = 0$, x_n is a decision variable



This entire sub-tree is effectively pruned

Labeled vertices are inferred by the assignment $x_n = 0$

DPLL ALGORITHM (CONT)

Convert formula to CNF first e.g (a v b) ^ (c v d) ^ (~d v a)

Deduction rule – For each clause if there is one unassigned variable *p* and all others are False then *p* must be True (if this isn't possible then the formula is UNSAT)

Clause learning – If a clause evaluates to False backtrack to the assignment (before any deductive assignments) that is the cause and add its negation to the formula to avoid it in the future

Branch selection heuristics – Selecting the variable to assign

DPLL ALGORITHM (CONT)[1]

Convert formula to CNF first e.g (a v b) ^ (c v d) ^ (~d v a)

Deduction rule – For each clause if there is one unassigned variable *p* and all others are False then *p* must be True (if this isn't possible then the formula is UNSAT)

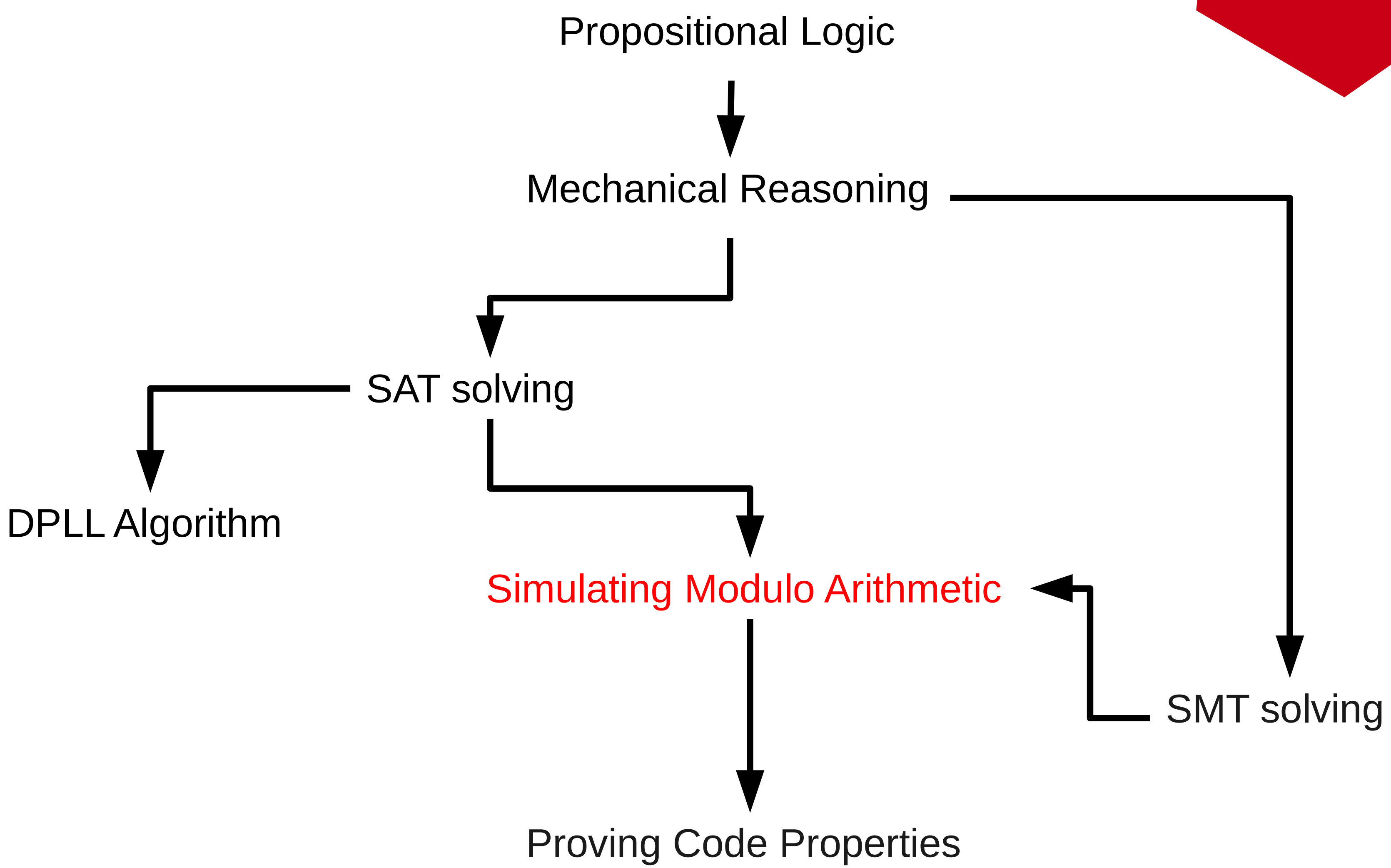
Clause learning – If a clause evaluates to False backtrack to the assignment (before any deductive assignments) that is the cause and add its negation to the formula to avoid it in the future

Branch selection heuristics – Selecting the variable to assign

Restarts after a given timeout to reduce sensitivity to unfortunate branch selection

[1] See Leonardo De Moura's MSR page for an excellent collection of papers on building real world solvers

PART I: DOWN THE RABBIT HOLE



Modulo Arithmetic & SAT

We can encode +, -, *, /, >>, << etc as operations over *bit-vectors*

MODULO ARITHMETIC & SAT

We can encode +, -, *, /, >>, << etc as operations over *bit-vectors*

1 byte, 8 bits, 8 propositional variables $<p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0>$

Modulo Arithmetic & SAT

We can encode +, -, *, /, >>, << etc as operations over *bit-vectors*

1 byte, 8 bits, 8 propositional variables $<p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0>$

Encoding of a full adder (where X is XOR)

cin_n = 0 if n == 0 or $carry_{n-1}$ otherwise

$sum(a_n, b_n, cin) = ((a_n \text{ X } b_n) \text{ X } cin_n)$

$carry(a_n, b_n, cin_n) = ((a_n \wedge b_n) \vee ((a_n \text{ X } b_n) \wedge cin_n)))$

MODULO ARITHMETIC & SAT

We can encode +, -, *, /, >>, << etc as operations over *bit-vectors*

1 byte, 8 bits, 8 propositional variables $<p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0>$

Encoding of a full adder (where X is XOR)

$cin_n = 0$ if $n == 0$ or $carry_{n-1}$ otherwise
$sum(a_n, b_n, cin) = ((a_n \text{ X } b_n) \text{ X } cin_n)$
$carry(a_n, b_n, cin_n) = ((a_n \wedge b_n) \vee ((a_n \text{ X } b_n) \wedge cin_n)))$

Can be extended to bitvectors of arbitrary lengths and used to represent subtraction

Modulo Arithmetic & SAT

Why do we care about DPLL
and optimising our search?

MODULO ARITHMETIC & SAT

Why do we care about DPLL and optimising our search?

Single multiplier – 8 bits[1]

n	# Vars	# Clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

[1] http://www.decision-procedures.org

MODULO ARITHMETIC & SAT

Why do we care about DPLL and optimising our search?

Single multiplier – 8 bits[1]

n	# Vars	# Clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Modern solvers can handle > 10^6 variables so we can reason about our initial problem but we're quickly going to hit problems using bit-flattening on other instances

[1] http://www.decision-procedures.org

MODULO ARITHMETIC & SAT

Why do we care about DPLL and optimising our search?

Single multiplier – 8 bits[1]

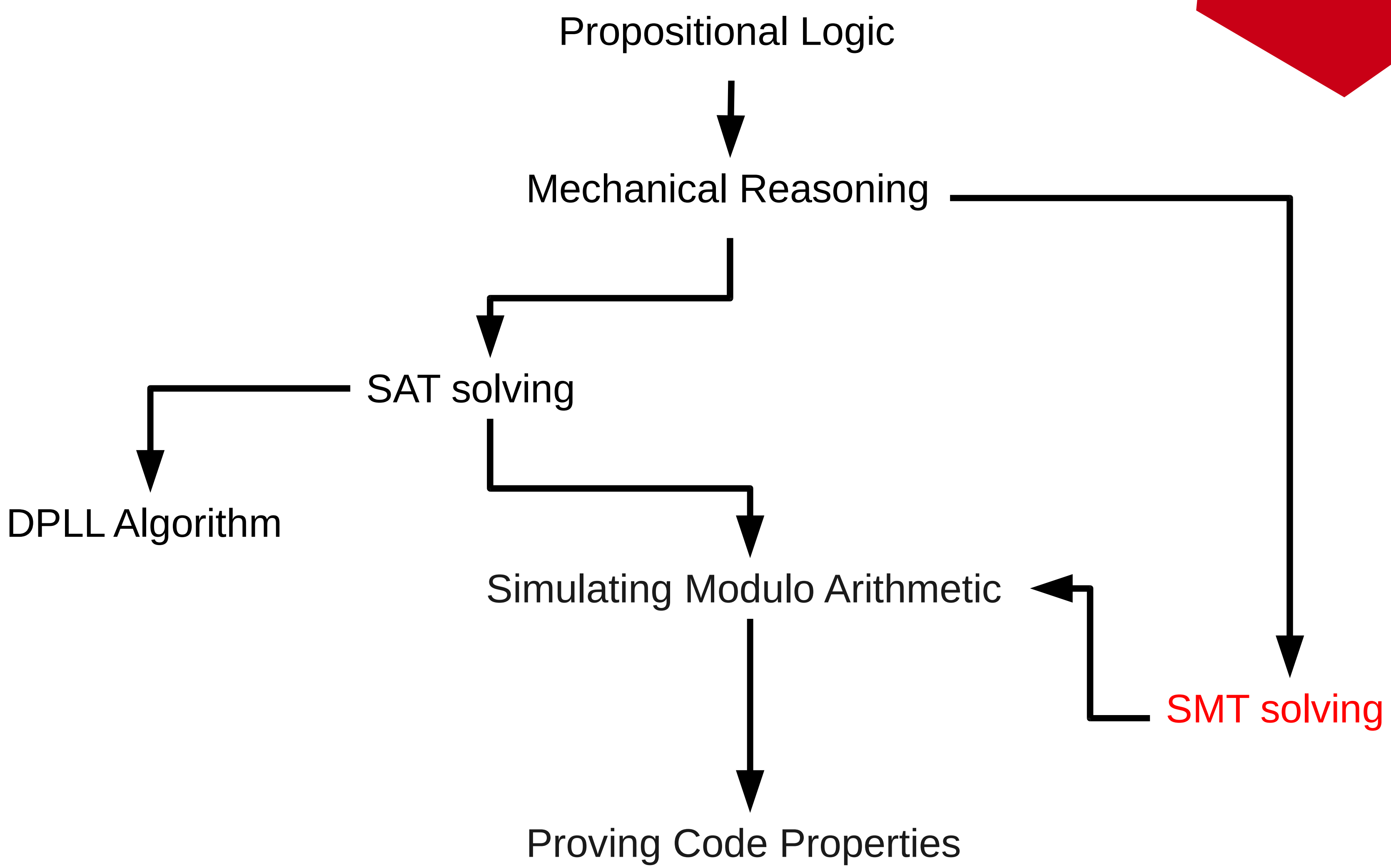
n	# Vars	# Clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Modern solvers can handle > 10^6 variables so we can reason about our initial problem but we're quickly going to hit problems using bit-flattening on other instances

SMT solvers

[1] http://www.decision-procedures.org

PART 1: DOWN THE RABBIT HOLE



SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

Satisfiability Modulo Theories – Determining the satisfiability of a formula accounting for an underlying theory e.g. the theory of linear arithmetic

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

Satisfiability Modulo Theories – Determining the satisfiability of a formula accounting for an underlying theory e.g. the theory of linear arithmetic

$$(a < b) \wedge (b < 10) \wedge (c = a + 1) \wedge (c = 10)$$

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

Satisfiability Modulo Theories – Determining the satisfiability of a formula accounting for an underlying theory e.g. the theory of linear arithmetic

$$(a < b) \wedge (b < 10) \wedge (c = a + 1) \wedge (c = 10)$$

Encode skeleton as a boolean formula $x_0 \wedge x_1 \wedge x_2 \wedge x_3$

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

Satisfiability Modulo Theories – Determining the satisfiability of a formula accounting for an underlying theory e.g. the theory of linear arithmetic

$$(a < b) \wedge (b < 10) \wedge (c = a + 1) \wedge (c = 10)$$

Encode skeleton as a boolean formula $x_0 \wedge x_1 \wedge x_2 \wedge x_3$

Find a satisfying assignment $x_0 = x_1 = x_2 = x_3 = T$

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

There are many algorithms for handling equality logic, linear/non-linear arithmetic and so forth[1] that don't rely on flattening out the equations to bit-vector operations e.g. simplex over linear inequalities

Satisfiability Modulo Theories – Determining the satisfiability of a formula accounting for an underlying theory e.g. the theory of linear arithmetic

$$(a < b) \wedge (b < 10) \wedge (c = a + 1) \wedge (c = 10)$$

Encode skeleton as a boolean formula $x_0 \wedge x_1 \wedge x_2 \wedge x_3$

Find a satisfying assignment $x_0 = x_1 = x_2 = x_3 = T$

Use a theory specific solver to check if this assignment is feasible $(a < b) \wedge (b < 10) \wedge (c = a + 1) \wedge (c = 10)$

[1] For sanity I've avoided a discussion of first order logic, decidability and combining theories that is relatively important. See Wikipedia or http://unprotectedhex.com/psv

SMT SOLVERS

Provide a much more intuitive encoding of programming language semantics

SMT SOLVERS

Provide a much more intuitive encoding of programming language semantics

Theories for arrays, lists, sets etc

SMT Solvers

Provide a much more intuitive encoding of programming language semantics

SMT-LIB specification describes a number of base theories and provides functions over them

Theories for arrays, lists, sets etc

SMT SOLVERS

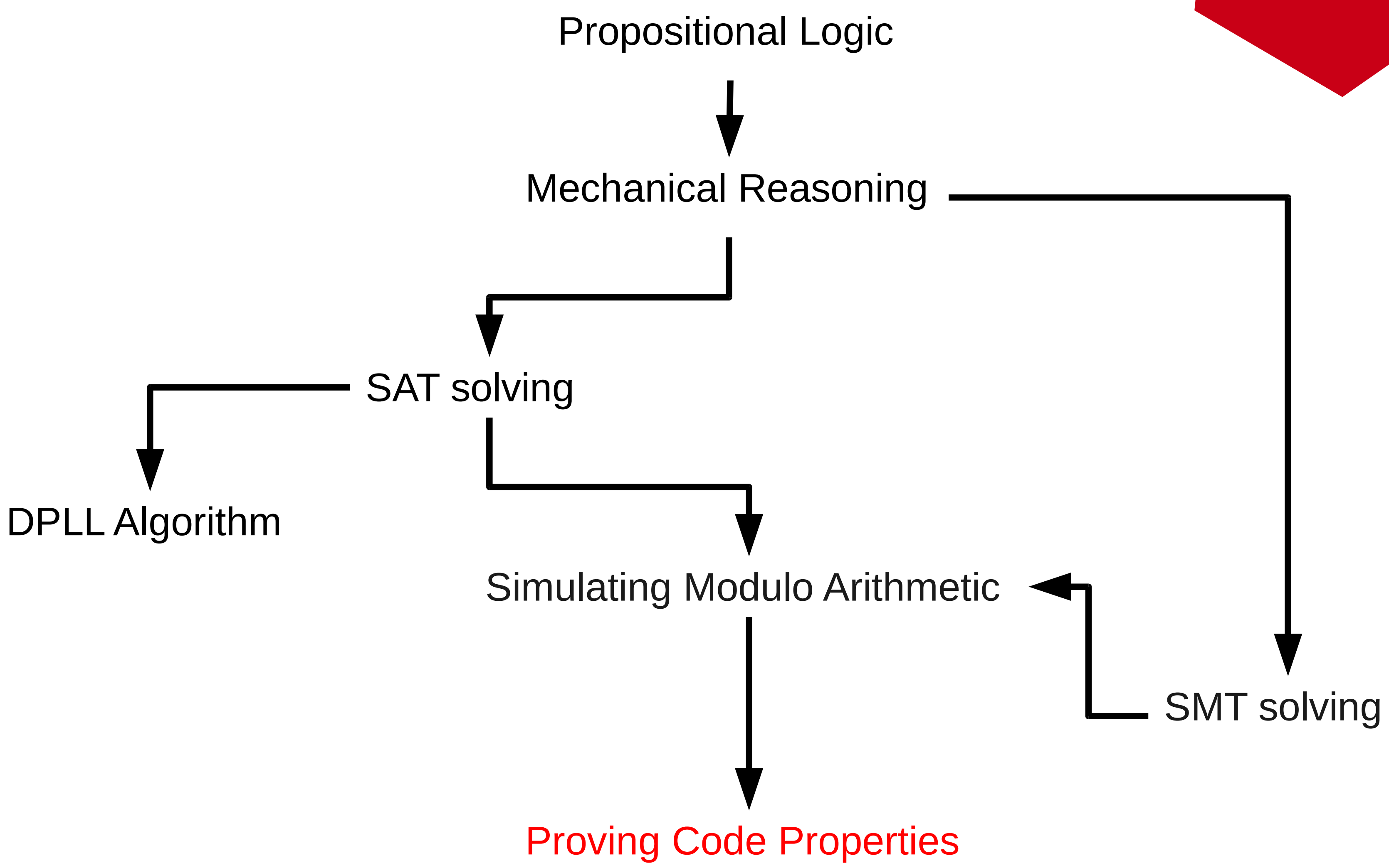
Provide a much more intuitive encoding of programming language semantics

Theories for arrays, lists, sets etc

SMT-LIB specification describes a number of base theories and provides functions over them

bvadd, bvmul, bvand, bvor, bvult, bvugt etc

PART 1: DOWN THE RABBIT HOLE



WebKit Code

```
void* ArrayBuffer::tryAllocate(unsigned numElements,
                unsigned elementByteSize)
{
    void* result;
    // Do not allow 32-bit overflow of the total size
    if (numElements) {
    unsigned totalSize = numElements * elementByteSize;
    if (totalSize / numElements != elementByteSize)
        return 0;
    }
    if (WTF::tryFastCalloc(numElements,
                            elementByteSize).getValue(result))
        return result;
    return 0;
}
```

WebKit Code

```
void* ArrayBuffer::tryAllocate(unsigned numElements,
                unsigned elementByteSize)
{
    void* result;
    // Do not allow 32-bit overflow of the total size
    if (numElements) {
    unsigned totalSize = numElements * elementByteSize;
    if (totalSize / numElements != elementByteSize)
        return 0;
    }
    if (WTF::tryFastCalloc(numElements,
                            elementByteSize).getValue(result))
        return result;
    return 0;
}
```

SMT-LIB model

```
(benchmark uint_ovf
:status unknown
:logic QF_BV

:extrafuns ((totalSize BitVec[32])(numElements BitVec[32])
        (elSize BitVec[32]))

; elSize != 0
; This assumption isn't actually accounted for in the code
:assumption (bvugt elSize bv0[32])
; if (numElements) {
:assumption (bvugt numElements bv0[32])

; unsigned totalSize = numElements * elementByteSize;
:assumption (= totalSize (bvmul numElements elSize))

; totalSize / numElements != elementByteSize
:assumption (= elSize (bvudiv totalSize numElements))

; Our requirement totalSize < numElements
:formula (bvult totalSize numElements)
)
```

WebKit Code

```
void* ArrayBuffer::tryAllocate(unsigned numElements,
                                unsigned elementByteSize)
{
    void* result;
    // Do not allow 32-bit overflow of the total size
    if (numElements) {
    unsigned totalSize = numElements * elementByteSize;
    if (totalSize / numElements != elementByteSize)
        return 0;
    }
    if (WTF::tryFastCalloc(numElements,
                               elementByteSize).getValue(result))
        return result;
    return 0;
}
```

SMT-LIB model

```
(benchmark uint_ovf
:status unknown
:logic QF_BV

:extrafuns ((totalSize BitVec[32])(numElements BitVec[32])
        (elSize BitVec[32]))

; elSize != 0
; This assumption isn't actually accounted for in the code
:assumption (bvugt elSize bv0[32])
; if (numElements) {
:assumption (bvugt numElements bv0[32])

; unsigned totalSize = numElements * elementByteSize;
:assumption (= totalSize (bvmul numElements elSize))

; totalSize / numElements != elementByteSize
:assumption (= elSize (bvudiv totalSize numElements))

; Our requirement totalSize < numElements
:formula (bvult totalSize numElements)
)
```

```
[sean@sean-laptop bin]$ time ./yices -f webkit.smt
unsat

real    0m0.220s
user    0m0.212s
sys     0m0.005s
```

Part II: One Hammer, Many Nails

ROP

↓

Path Filtering

↓

Finding variable bounds

↓

Finding integer overflows

↓

Loops and solvers

↓

Performance/Scaling

PART II: ONE HAMMER, MANY NAILS

ROP

↓

Path Filtering

↓

Finding variable bounds

↓

Finding integer overflows

↓

Loops and solvers

↓

Performance/Scaling

WHICH OF THESE EXPRESS EAX = 0

```
MOV EAX,DWORD PTR SS:[EBP-4]
POP EDI
POP ESI
POP EBX
LEAVE
RETN 4
```

```
MOV EAX,ESI
POP ESI
POP EBP
RETN 8
```

```
XOR EAX,EAX
INC EAX
POP EDI
POP ESI
POP EBX
LEAVE
RETN 8
```

```
ADC BYTE PTR DS:[ECX+C5E82404],CL
OR AL,BYTE PTR DS:[EAX]
ADD BYTE PTR DS:[EBX+658D14EC],AL
CLC
POP EBX
POP ESI
POP EBP
RETN 10
```

```
XOR EDI,EDI
MOV EAX,EDI
POP EDI
LEAVE
RETN 4
```

```
XOR EAX,EAX
POP ESI
LEAVE
RETN 4
```

```
MOV ESI,DWORD PTR SS:[EBP-8]
MOV EDI,DWORD PTR SS:[EBP-4]
SUB ESP,4
MOV ESP,EBP
POP EBP
RETN 10
```

```
SBB EAX,EAX
AND EAX,7FF6FD00
ADD EAX,80090300
POP EBP
RETN 8
```

```
ADD EAX,90302B8
ADD BYTE PTR SS:[EBP+8B0B7CC0],4D
ADC AL,8B
PUSH EBP
CLD
MOV DWORD PTR DS:[ECX+4],EDX
MOV DWORD PTR DS:[ECX],ESI
POP ESI
LEAVE
RETN 10
```

```
MOV AL,BYTE PTR SS:[EBP+B]
POP EDI
POP ESI
POP EBX
LEAVE
RETN 14
```

REASONING ABOUT X86 USING SMT SOLVERS

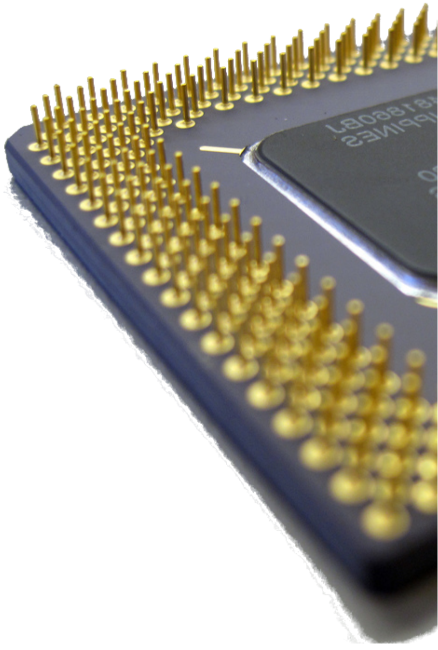
1) Convert x86 to SMT representation *F*

(We've seen it's possible to model code as formulae so we just need to automate that)

REASONING ABOUT X86 USING SMT SOLVERS

1) Convert x86 to SMT representation *F*
   (We've seen it's possible to model code as formulae so we just need to automate that)

2) Convert any requirements we have/conditions we want met into another SMT formula *C*

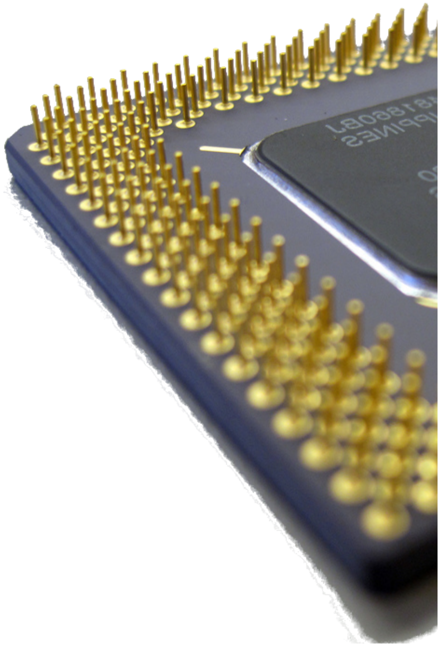


REASONING ABOUT X86 USING SMT SOLVERS

1) Convert x86 to SMT representation *F*
    (We've seen it's possible to model code as formulae so we just need to automate that)

2) Convert any requirements we have/conditions we want met into another SMT formula *C*

3) If *(F ^ C)* is SAT then we can retrieve a satisfying assignment from the solver that results in our conditions being satisfied



REASONING ABOUT X86 USING SMT SOLVERS

1) Convert x86 to SMT representation *F*
(We've seen it's possible to model code as formulae so we just need to automate that)

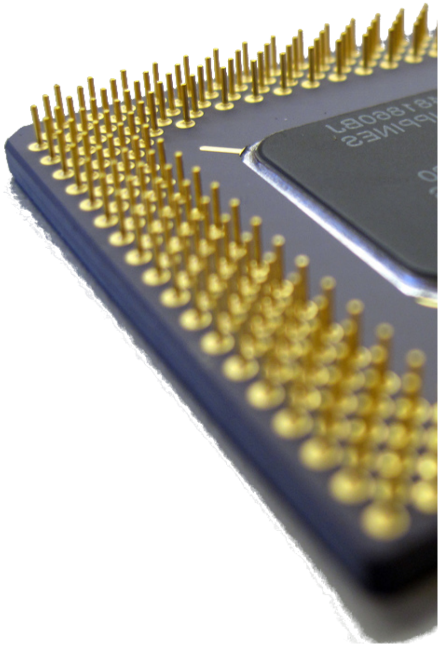
2) Convert any requirements we have/conditions we want met into another SMT formula *C*

3) If *(F ^ C)* is SAT then we can retrieve a satisfying assignment from the solver that results in our conditions being satisfied

Equivalence vs Implication
(Side effects)

EAX == 0

```
XOR EDI,EDI
MOV EAX,EDI
POP EDI
LEAVE
RETN 4
```



REASONING ABOUT X86 USING SMT SOLVERS

1) Convert x86 to SMT representation *F*
   (We've seen it's possible to model code as formulae so we just need to automate that)

2) Convert any requirements we have/conditions we want met into another SMT formula *C*

3) If *(F ^ C)* is SAT then we can retrieve a satisfying assignment from the solver that results in our conditions being satisfied

Equivalence vs Implication
(Side effects)

EAX == 0

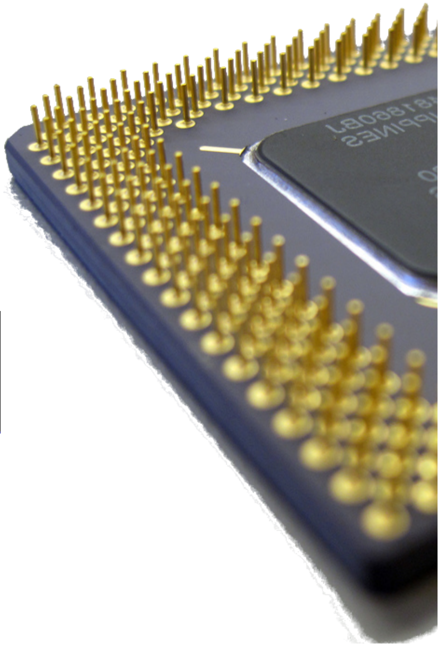
```
XOR EDI,EDI
MOV EAX,EDI
POP EDI
LEAVE
RETN 4
```

Validity versus Satisfiability
(Context sensitive gadgets)

```
XOR EAX,EAX
POP ESI
LEAVE
RETN 4
```

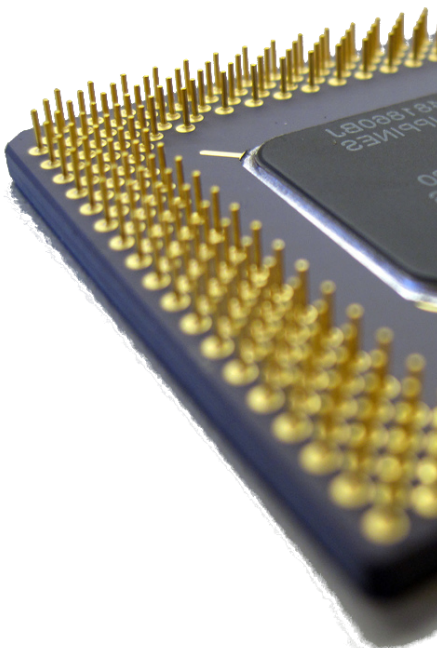
vs

```
MOV EAX,ESI
POP ESI
POP EBP
RETN 8
```



REASONING ABOUT X86 USING SMT SOLVERS

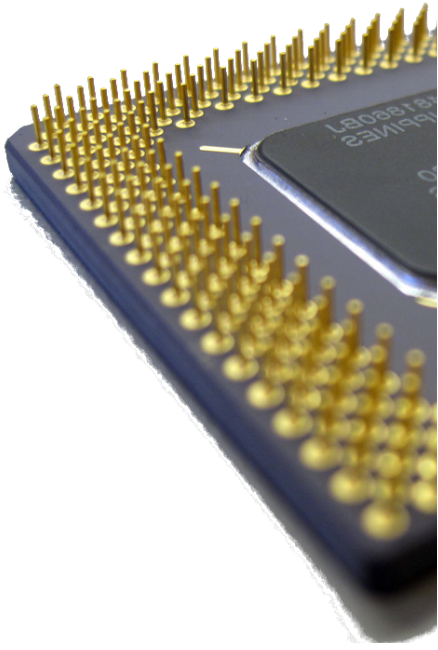
```
XOR EAX,EAX
INC EAX
POP EDI
POP ESI
POP EBX
LEAVE
RETN 8
```



REASONING ABOUT X86 USING SMT SOLVERS

```
XOR EAX,EAX
INC EAX
POP EDI
POP ESI
POP EBX
LEAVE
RETN 8
```

```python
def analyzeXOR(self, op):
    dst = self.buildState(op.operand[0]);
    src = self.buildState(op.operand[1]);
    (res,dstval,srcval)=self.solveArithmetic(self.state.solver.xorExpr,dst,src)
    self.updateFlags("LOGIC", res, dstval, srcval)

def analyzeINC(self, op):
    cf = self.state.flags['_CF']
    op.operand = (op.operand[0], op.constantOperand(1), op.emptyOperand())
    self.analyzeADD(op)
    self.state.flags['_CF'] = cf


def analyzeRETN(self, op):
    self.state.EIP = self.getMemoryStateFromSolverState(self.state.regs['ESP'], 32)
    self.state.RETNoffset = self.state.solver.constExpr(op.op1Constant())
    self.state.regs['ESP'] = self.state.solver.addExpr(self.state.regs['ESP'],
                                 self.state.solver.addExpr(
                                     self.state.solver.constExpr(4),
                                     self.state.RETNoffset))
```

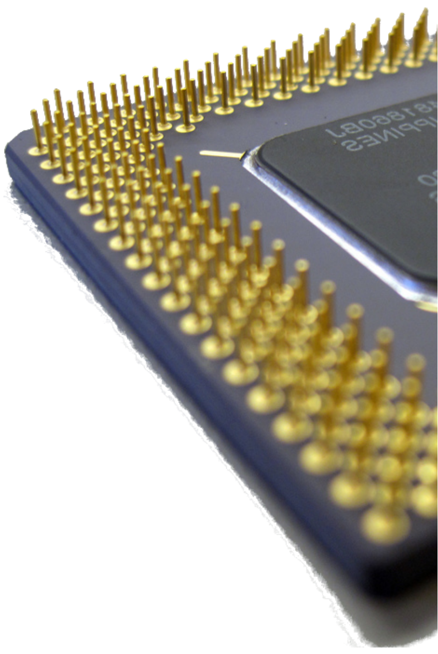


REASONING ABOUT X86 USING SMT SOLVERS

```
XOR EAX,EAX
INC EAX
POP EDI
POP ESI
POP EBX
LEAVE
RETN 8
```

```
def analyzeXOR(self, op):
    dst = self.buildState(op.operand[0]);
    src = self.buildState(op.operand[1]);
    (res,dstval,srcval)=self.solveArithmetic(self.state.solver.xorExpr,dst,src)
    self.updateFlags("LOGIC", res, dstval, srcval)

def analyzeINC(self, op):
    cf = self.state.flags['_CF']
    op.operand = (op.operand[0], op.constantOperand(1), op.emptyOperand())
    self.analyzeADD(op)
    self.state.flags['_CF'] = cf


def analyzeRETN(self, op):
    self.state.EIP = self.getMemoryStateFromSolverState(self.state.regs['ESP'], 32)
    self.state.RETNoffset = self.state.solver.constExpr(op.op1Constant())
    self.state.regs['ESP'] = self.state.solver.addExpr(self.state.regs['ESP'],
                                    self.state.solver.addExpr(
                                        self.state.solver.constExpr(4),
                                        self.state.RETNoffset))
```

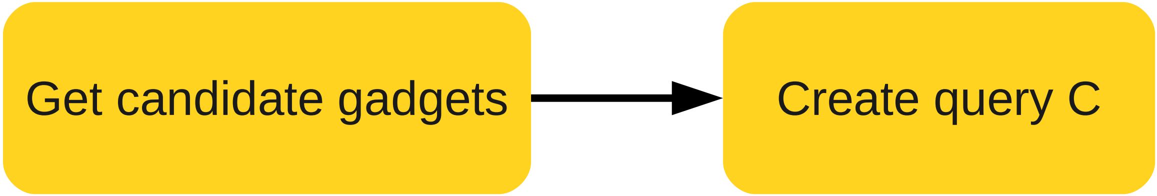
```
def xorExpr(self, exp1, exp2):
    return self.CVC.vc_bvXorExpr(self.vc, exp1, exp2) | self.get_error()

def negExpr(self, exp):
    return self.CVC.vc_bvUMinusExpr(self.vc, exp) | self.get_error()

def addExpr(self, exp1, exp2, bits=None):
    if not bits: bits=self.getBitSizeFromExpr(exp1)
    return self.CVC.vc_bvPlusExpr(self.vc, bits, exp1, exp2) | self.get_error()

def subExpr(self, exp1, exp2, bits=None):
    if not bits: bits=self.getBitSizeFromExpr(exp1)
    return self.CVC.vc_bvMinusExpr(self.vc, bits, exp1, exp2) | self.get_error()
```



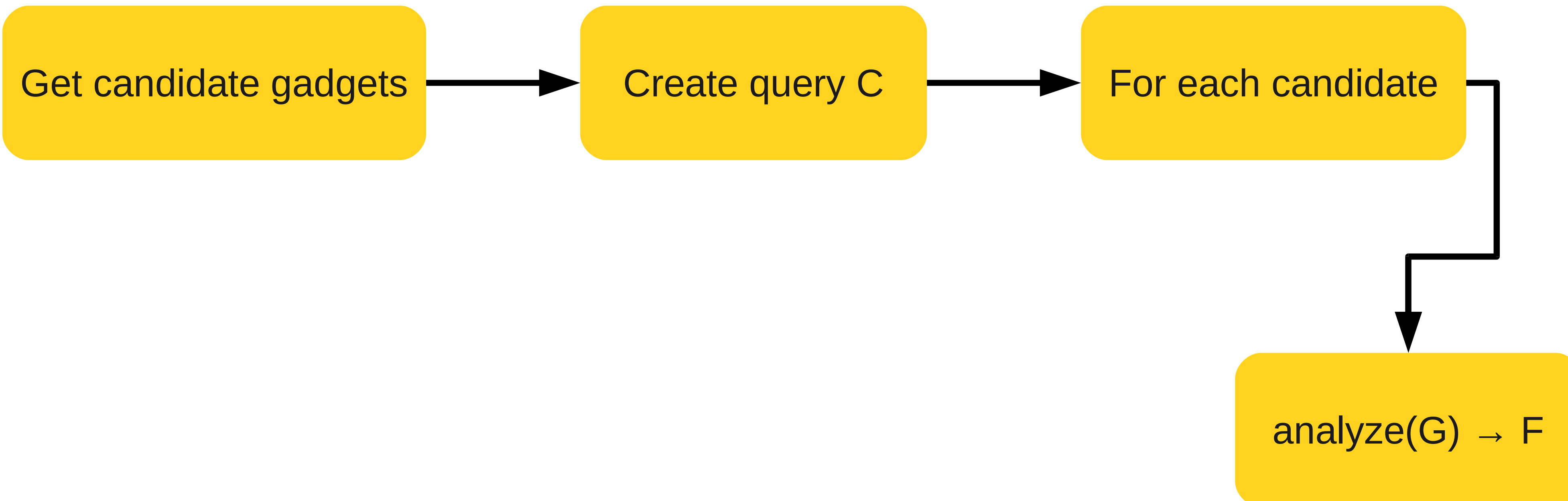
find_gadget.py # Finding gadgets with specific semantics

Get candidate gadgets

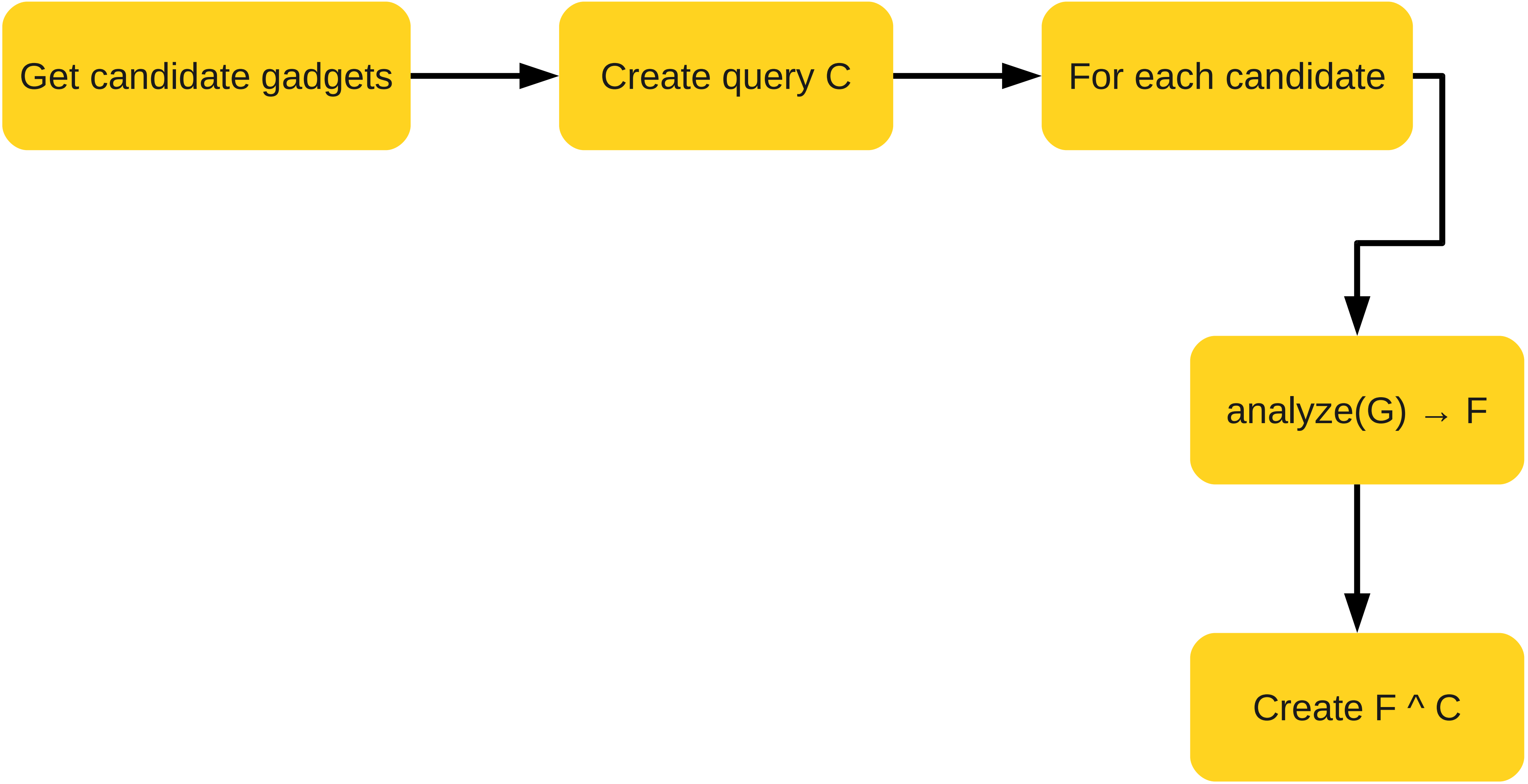
find_gadget.py # Finding gadgets with specific semantics



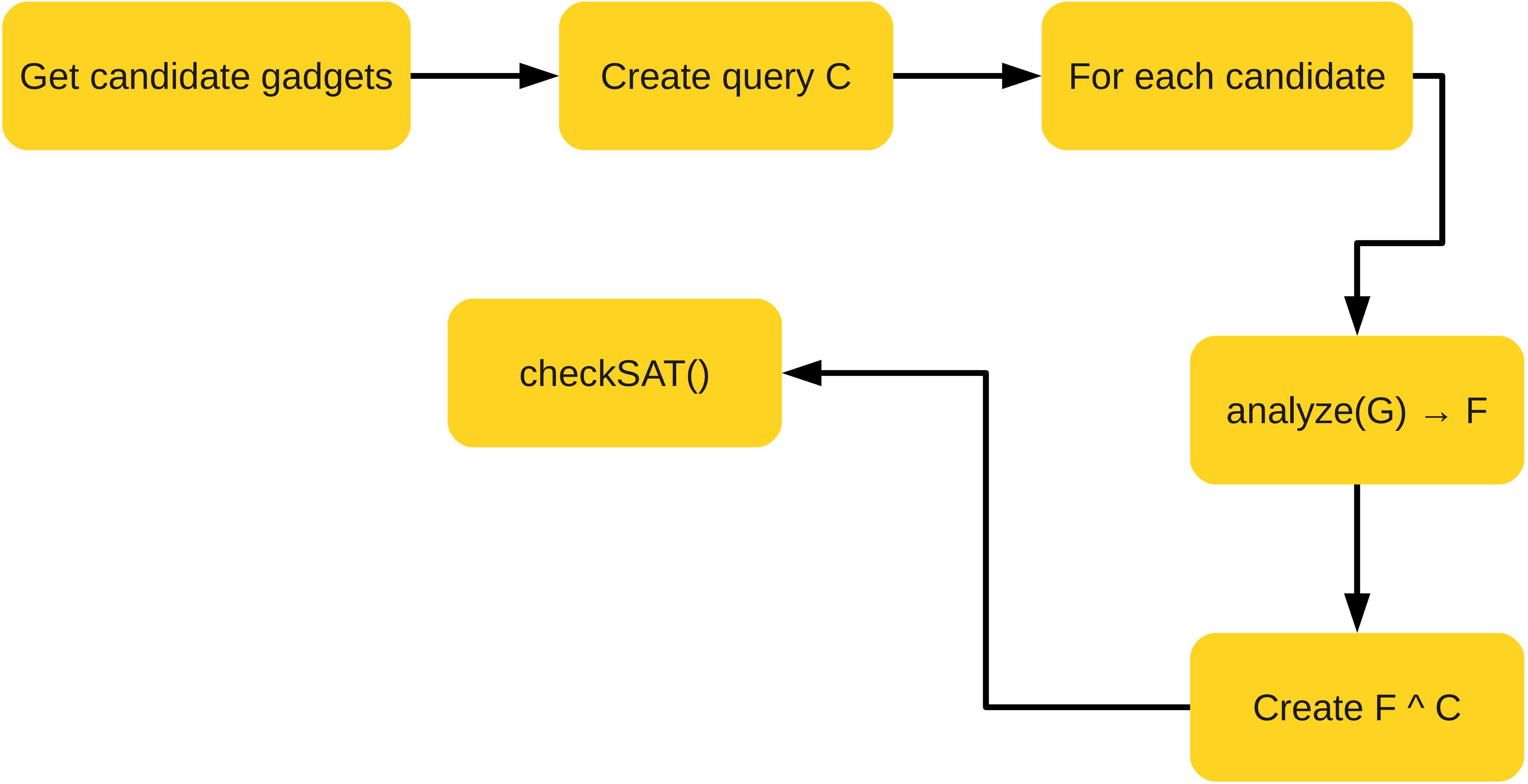
find_gadget.py # Finding gadgets with specific semantics



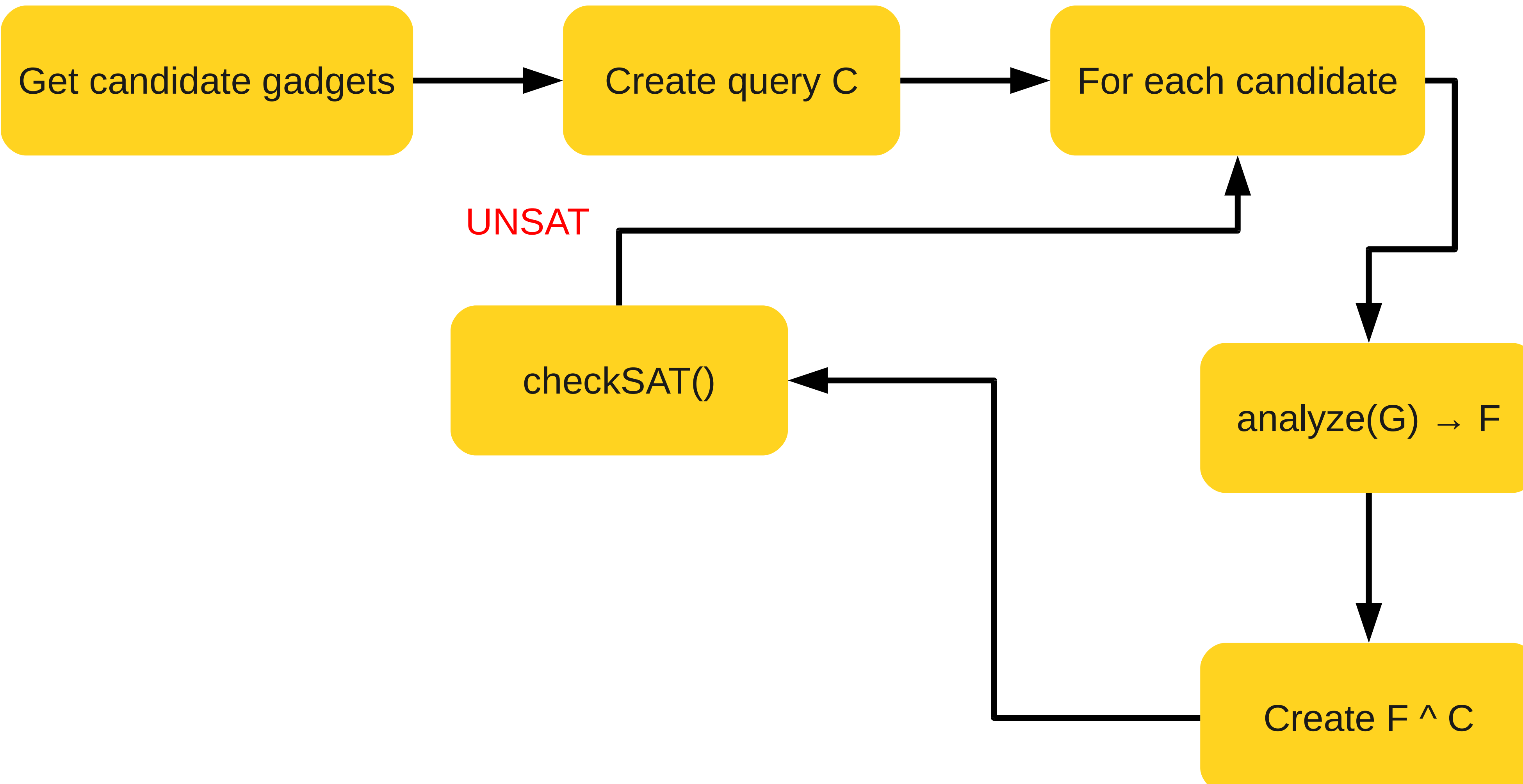
FIND_GADGET.PY # FINDING GADGETS WITH SPECIFIC SEMANTICS



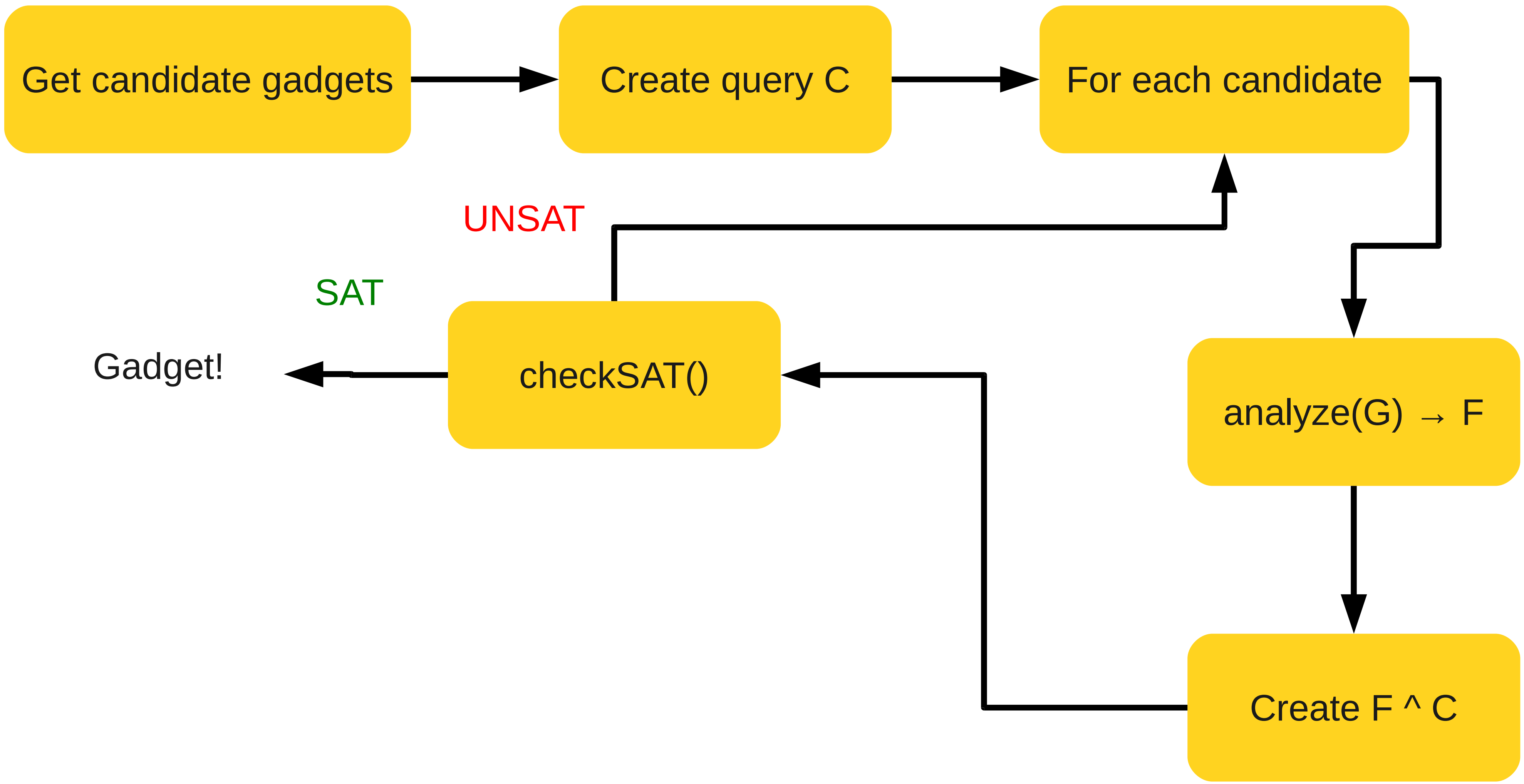
FIND_GADGET.PY # FINDING GADGETS WITH SPECIFIC SEMANTICS



FIND_GADGET.PY # FINDING GADGETS WITH SPECIFIC SEMANTICS



FIND_GADGET.PY # FINDING GADGETS WITH SPECIFIC SEMANTICS



find_gadget.py # finding gadgets with specific semantics

DEMO

INFRASTRUCTURE

SequenceAnalyzer – Models x86 as operations over a set of SMT primitives

Solver – Ctypes interface to the CVC3 SMT solver API. Supports a variety of theories including quantifier free, bit-vector arithmetic, linear arithmetic etc.

Part II: One Hammer, Many Nails

ROP

Path Filtering

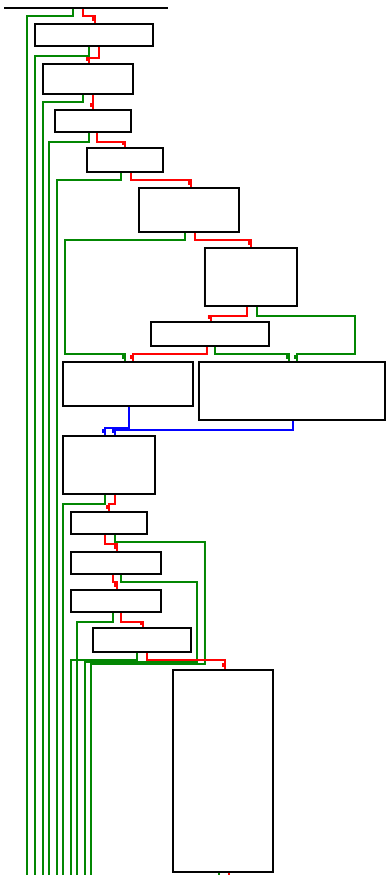
Finding variable bounds

Finding integer overflows

Loops and solvers

Performance/Scaling

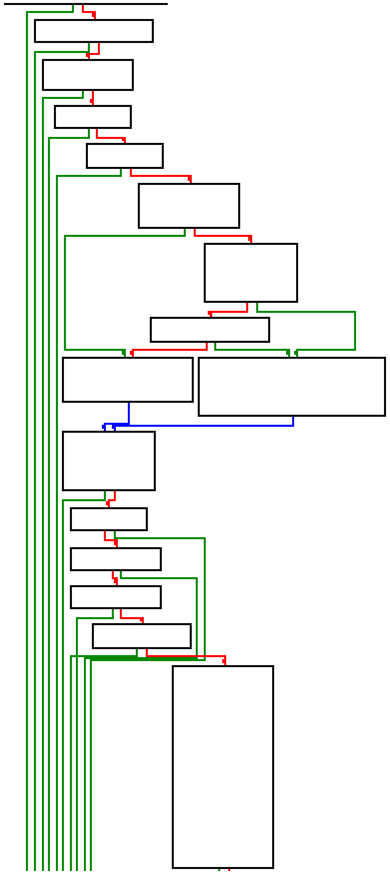
PATHOGEN.PY # FILTERING INVALID PATHS



PATHOGEN.PY # FILTERING INVALID PATHS

A problem for both manual analysis and automated tools

What paths in the CFG are actually possible given the constraints? Given a certain configuration of registers/memory?

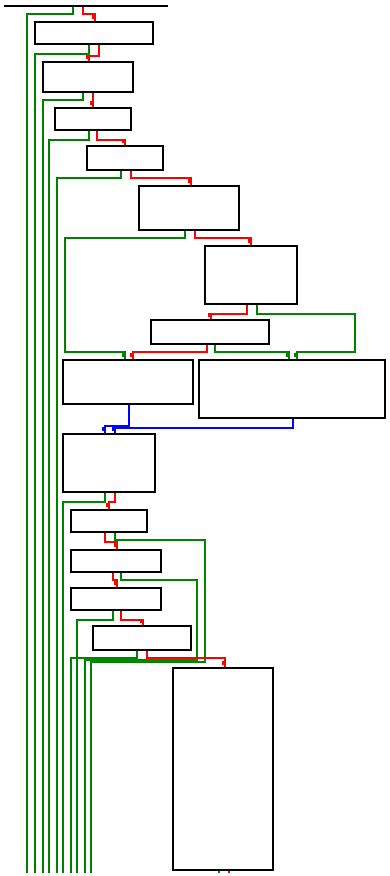


PATHOGEN.PY # FILTERING INVALID PATHS

A problem for both manual analysis and automated tools

What paths in the CFG are actually possible given the constraints? Given a certain configuration of registers/memory?

Solvable quite naturally with an SMT encoding of code. Conditional jmps become equalities/inequalities



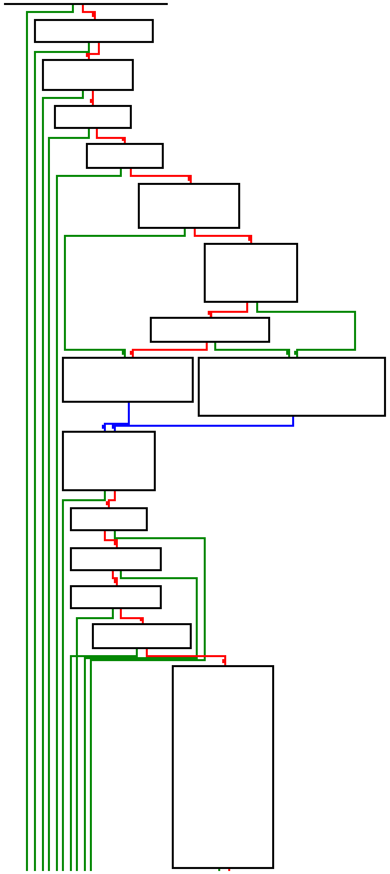
PATHOGEN.PY # FILTERING INVALID PATHS

A problem for both manual analysis and automated tools

What paths in the CFG are actually possible given the constraints? Given a certain configuration of registers/memory?

Solvable quite naturally with an SMT encoding of code. Conditional jmps become equalities/inequalities

1) Generate a path, codegraph.py + pathgenerator.py (No SMT involved here, just a backend for any static analysis stuff)

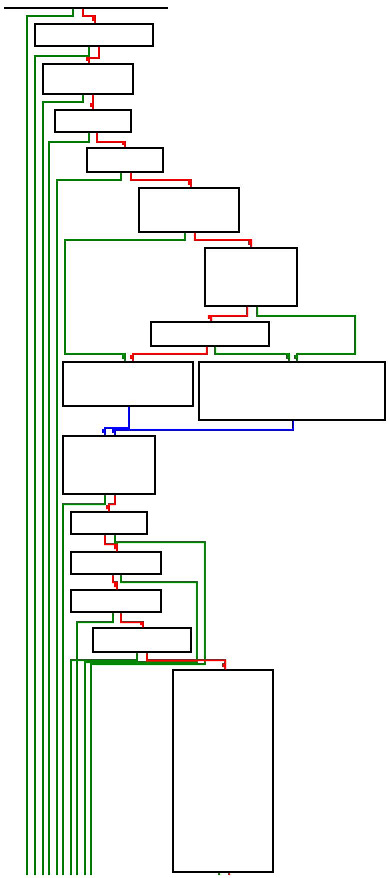


PATHOGEN.PY # FILTERING INVALID PATHS

A problem for both manual analysis and automated tools

What paths in the CFG are actually possible given the constraints? Given a certain configuration of registers/memory?

Solvable quite naturally with an SMT encoding of code. Conditional jmps become equalities/inequalities



1) Generate a path, codegraph.py + pathgenerator.py (No SMT involved here, just a backend for any static analysis stuff)

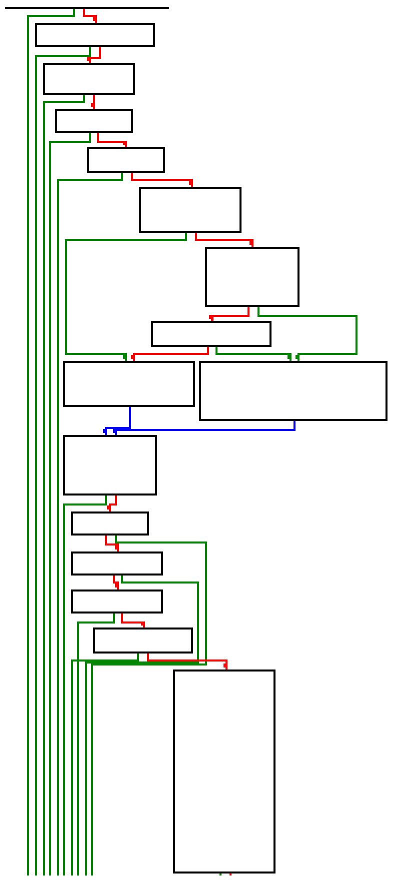
2) Build *path condition* over each path. After every conditional jmp call *checkSAT()*

PATHOGEN.PY # FILTERING INVALID PATHS

A problem for both manual analysis and automated tools

What paths in the CFG are actually possible given the constraints? Given a certain configuration of registers/memory?

Solvable quite naturally with an SMT encoding of code. Conditional jmps become equalities/inequalities



1) Generate a path, codegraph.py + pathgenerator.py (No SMT involved here, just a backend for any static analysis stuff)

2) Build *path condition* over each path. After every conditional jmp call *checkSAT()*

3) Discard any paths that return UNSAT

PATHOGEN.PY # FILTERING INVALID PATHS

Path iterator

Checks conditions as it walks the path

```
# Use the path generator to dump all possible paths
pg = PathGenerator(bb_graph.basic_blocks, bb_graph.bb_out_edges)
pg.imm = imm

cnt = 0
feasible_paths = []
for path in pg.generate_paths(start_addr):
    cnt += 1

    path.log(imm)
    if not prune_paths:
        continue

    p_walker = PathWalker(imm, debug=True)
    try:
        p_walker.walk(path)
        feasible_paths.append(path)
    except UnsatPathConditionException, e:
        imm.log("%s" % str(e))
```

PATHOGEN.PY # FILTERING INVALID PATHS

Check path taken condition

Check path not taken condition

```
def analyzeJccNoFollow(self, condition, finaladdress):
    if self.check_jcc_taken:
        self.state.solver.push()
        # Check if the jcc can be taken
        ret = self.state.solver.checkUnsat(condition)
        self.state.solver.pop()

        if not ret:
            self.jcc_taken = True
        else:
            self.jcc_taken = False
        # assert the condition as true and set everything for analysis
        self.jcc_taken_condition = condition

    if self.check_jcc_not_taken:
        self.state.solver.push()
        # Check if the fall-through is possible
        not_cond = self.state.solver.boolNotExpr(condition)
        ret = self.state.solver.checkUnsat(not_cond)
        self.state.solver.pop()

        if not ret:
            self.jcc_not_taken = True
        else:
            self.jcc_not_taken = False

        self.jcc_not_taken_condition = not_cond
```

pathogen.py # filtering invalid paths

DEMO

INFRASTRUCTURE

SequenceAnalyzer – Models x86 as operations over a set of SMT primitives.

Solver – Ctypes interface to the CVC3 SMT solver API. Supports a variety of theories including quantifier free, bit-vector arithmetic, linear arithmetic etc.

CodeGraph/PathGenerator – Purely static CFG building and path generation.

PathWalker – SMT based path traversal. Each conditional jump is checked for feasibilty and the path discarded if not SAT.

Part II: One Hammer, Many Nails

ROP

↓

Path Filtering

↓

Finding variable bounds

↓

Finding integer overflows

↓

Loops and solvers

↓

Performance/Scaling

VARBOUNDS.PY # FINDING VARIABLE RANGES

```
MOV ESI,EDX
SHR ESI,4
XOR ESI,ECX
AND ESI,0F0F0F0F
XOR ECX,ESI
SHL ESI,4
XOR EDX,ESI
PUSH EDI
MOV ESI,ECX
SHL ESI,12
XOR ESI,ECX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR ECX,EDI
MOV ESI,EDX
SHL ESI,12
XOR ESI,EDX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR EDX,EDI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV ESI,ECX
SHR ESI,8
XOR ESI,EDX
AND ESI,0FF00FF
XOR EDX,ESI
SHL ESI,8
XOR ECX,ESI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
MOV EAX,DWORD PTR SS:[EBP-
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV EDI,EDX
SHR EDI,0C
AND EDI,0FF0
MOV ESI,ECX
AND ESI,F000000F
OR EDI,ESI
MOV ESI,EDX
AND ESI,0FF
SHR EDI,4
SHL ESI,10
OR EDI,ESI
AND EDX,0FF00
OR EDI,EDX
AND ECX,0FFFFFFF
XOR EDX,EDX
MOV DWORD PTR SS:[EBP-8],E
MOV BL,BYTE PTR DS:[EDX+7
TEST BL,BL
MOV EDX,EDI
MOV ESI,ECX
```

VARBOUNDS.PY # FINDING VARIABLE RANGES

Given we control EDX at the start what range of values may ESI have at the end?

```
MOV ESI,EDX
SHR ESI,4
XOR ESI,ECX
AND ESI,0F0F0F0F
XOR ECX,ESI
SHL ESI,4
XOR EDX,ESI
PUSH EDI
MOV ESI,ECX
SHL ESI,12
XOR ESI,ECX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR ECX,EDI
MOV ESI,EDX
SHL ESI,12
XOR ESI,EDX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR EDX,EDI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV ESI,ECX
SHR ESI,8
XOR ESI,EDX
AND ESI,0FF00FF
XOR EDX,ESI
SHL ESI,8
XOR ECX,ESI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
MOV EAX,DWORD PTR SS:[EBP-
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV EDI,EDX
SHR EDI,0C
AND EDI,0FF0
MOV ESI,ECX
AND ESI,F000000F
OR EDI,ESI
MOV ESI,EDX
AND ESI,0FF
SHR EDI,4
SHL ESI,10
OR EDI,ESI
AND EDX,0FF00
OR EDI,EDX
AND ECX,0FFFFFFF
XOR EDX,EDX
MOV DWORD PTR SS:[EBP-8],E
MOV BL,BYTE PTR DS:[EDX+7
TEST BL,BL
MOV EDX,EDI
MOV ESI,ECX
```

VARBOUNDS.PY # FINDING VARIABLE RANGES

Given we control EDX at the start what range of values may ESI have at the end?

Dumb solution? For an *n* bit output register *r* create 2^n formulae, one for each value *v* of *r. C* in this case is the constraint *r == v*

```
MOV ESI,EDX
SHR ESI,4
XOR ESI,ECX
AND ESI,0F0F0F0F
XOR ECX,ESI
SHL ESI,4
XOR EDX,ESI
PUSH EDI
MOV ESI,ECX
SHL ESI,12
XOR ESI,ECX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR ECX,EDI
MOV ESI,EDX
SHL ESI,12
XOR ESI,EDX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR EDX,EDI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV ESI,ECX
SHR ESI,8
XOR ESI,EDX
AND ESI,0FF00FF
XOR EDX,ESI
SHL ESI,8
XOR ECX,ESI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
MOV EAX,DWORD PTR SS:[EBP-
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV EDI,EDX
SHR EDI,0C
AND EDI,0FF0
MOV ESI,ECX
AND ESI,F000000F
OR EDI,ESI
MOV ESI,EDX
AND ESI,0FF
SHR EDI,4
SHL ESI,10
OR EDI,ESI
AND EDX,0FF00
OR EDI,EDX
AND ECX,0FFFFFFF
XOR EDX,EDX
MOV DWORD PTR SS:[EBP-8],E
MOV BL,BYTE PTR DS:[EDX+7
TEST BL,BL
MOV EDX,EDI
MOV ESI,ECX
```

VARBOUNDS.PY # FINDING VARIABLE RANGES

Given we control EDX at the start what range of values may ESI have at the end?

Dumb solution? For an *n* bit output register *r* create 2^n formulae, one for each value *v* of *r*. *C* in this case is the constraint *r* == *v*

```
mov ebx, eax
add ebx 0xFF
and ebx, 0xFF
```

```
MOV ESI,EDX
SHR ESI,4
XOR ESI,ECX
AND ESI,0F0F0F0F
XOR ECX,ESI
SHL ESI,4
XOR EDX,ESI
PUSH EDI
MOV ESI,ECX
SHL ESI,12
XOR ESI,ECX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR ECX,EDI
MOV ESI,EDX
SHL ESI,12
XOR ESI,EDX
AND ESI,CCCC0000
MOV EDI,ESI
SHR EDI,12
XOR EDI,ESI
XOR EDX,EDI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV ESI,ECX
SHR ESI,8
XOR ESI,EDX
AND ESI,0FF00FF
XOR EDX,ESI
SHL ESI,8
XOR ECX,ESI
MOV ESI,EDX
SHR ESI,1
XOR ESI,ECX
MOV EAX,DWORD PTR SS:[EBP+
AND ESI,55555555
XOR ECX,ESI
ADD ESI,ESI
XOR EDX,ESI
MOV EDI,EDX
SHR EDI,0C
AND EDI,0FF0
MOV ESI,ECX
AND ESI,F000000F
OR EDI,ESI
MOV ESI,EDX
AND ESI,0FF
SHR EDI,4
SHL ESI,10
OR EDI,ESI
AND EDX,0FF00
OR EDI,EDX
AND ECX,0FFFFFFF
XOR EDX,EDX
MOV DWORD PTR SS:[EBP-8],E
MOV BL,BYTE PTR DS:[EDX+7
TEST BL,BL
MOV EDX,EDI
MOV ESI,ECX
```

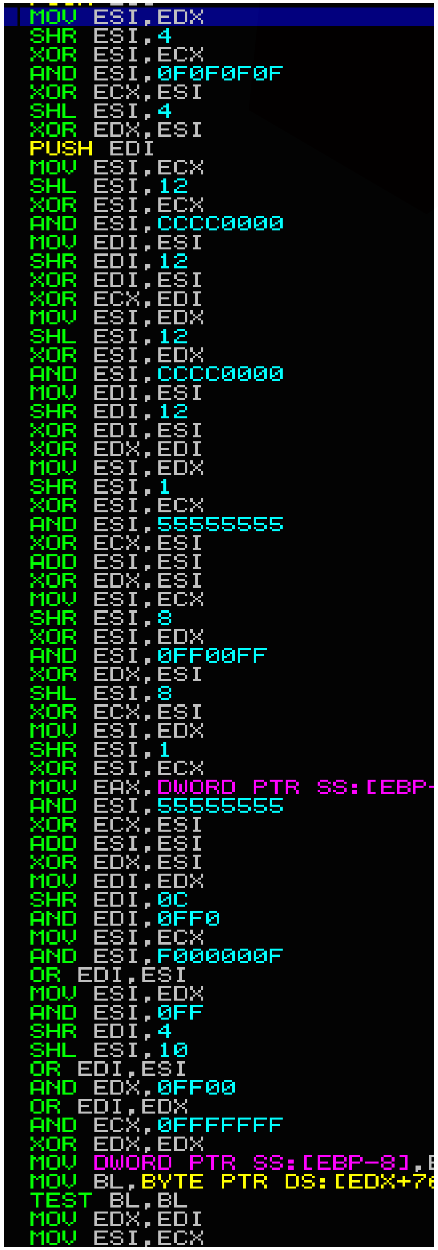
VARBOUNDS.PY # FINDING VARIABLE RANGES

Given we control EDX at the start what range of values may ESI have at the end?

Dumb solution? For an *n* bit output register *r* create 2^n formulae, one for each value *v* of *r. C* in this case is the constraint *r* == *v*

```
mov ebx, eax
add ebx 0xFF
and ebx, 0xFF
```

F: (= ebx^0, eax^0) ^ (= ebx^1, (+ ebx^0, 0xFF) ^
(= ebx^2, (AND ebx^1, 0xFF))



VARBOUNDS.PY # FINDING VARIABLE RANGES

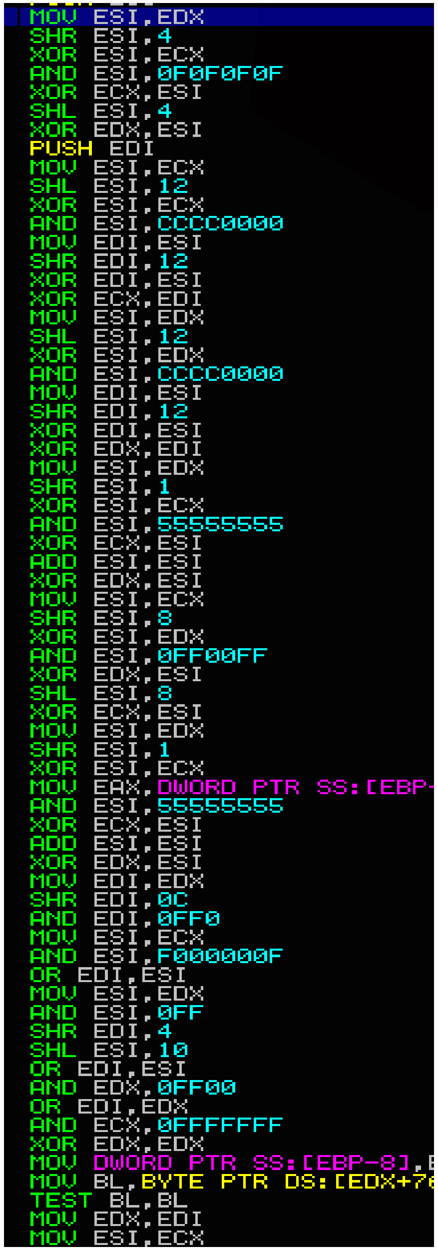
Given we control EDX at the start what range of values may ESI have at the end?

Dumb solution? For an *n* bit output register *r* create 2^n formulae, one for each value *v* of *r*. *C* in this case is the constraint *r* == *v*

```
mov ebx, eax
add ebx 0xFF
and ebx, 0xFF
```

F: (= ebx^0, eax^0) ^ (= ebx^1, (+ ebx^0, 0xFF) ^
(= ebx^2, (AND ebx^1, 0xFF))

C: (= ebx^2, x) for all x in [0, .., 2^n]



VARBOUNDS.PY # FINDING VARIABLE RANGES

Given we control EDX at the start what range of values may ESI have at the end?

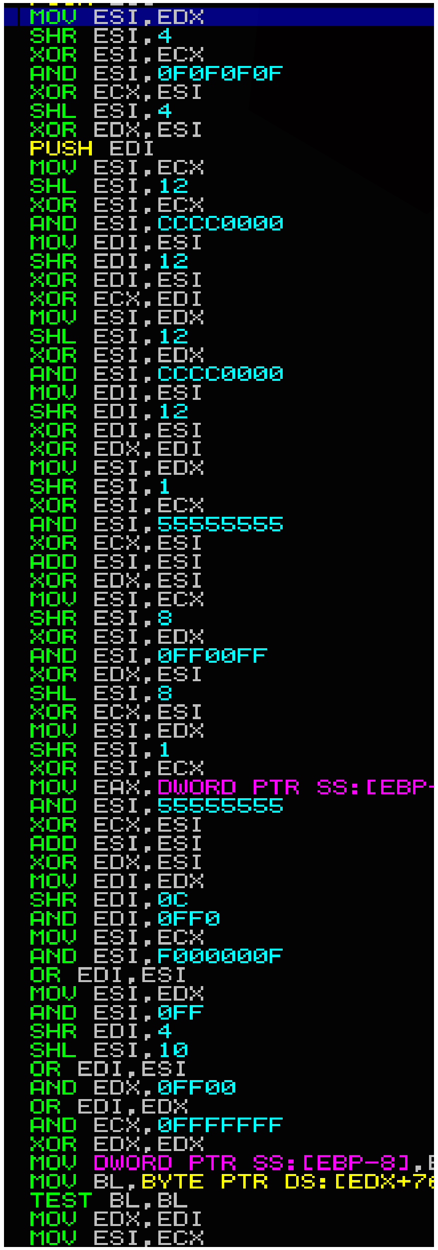
Dumb solution? For an *n* bit output register *r* create 2^n formulae, one for each value *v* of *r*. *C* in this case is the constraint *r* == *v*

```
mov ebx, eax
add ebx 0xFF
and ebx, 0xFF
```

F: (= ebx^0, eax^0) ^ (= ebx^1, (+ ebx^0, 0xFF) ^
(= ebx^2, (AND ebx^1, 0xFF))

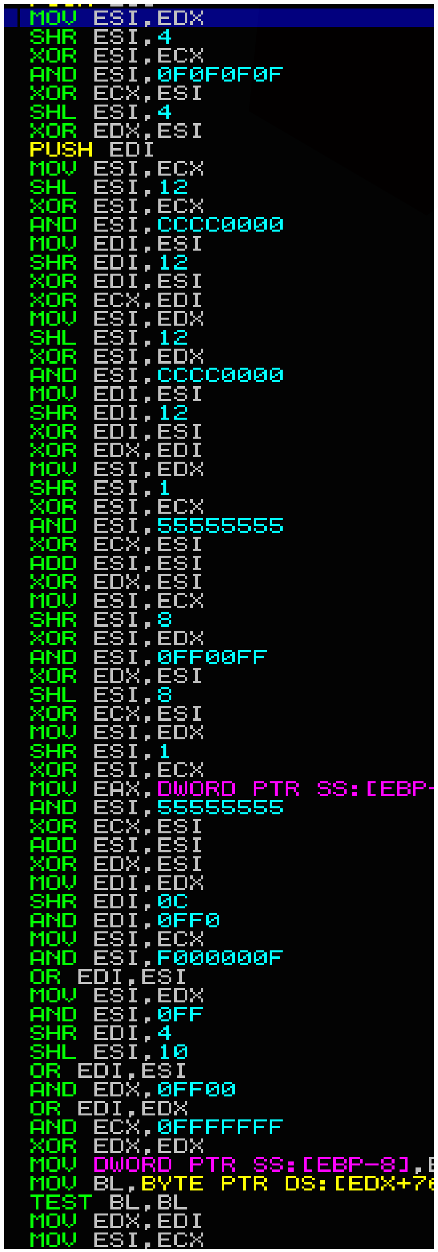
C: (= ebx^2, x) for all x in [0, .., 2^n]

Every SAT result indicates that *x* is a possible output value and *getConcreteModel()* gives the value required in EDX to achieve it



VARBOUNDS.PY # FINDING VARIABLE RANGES

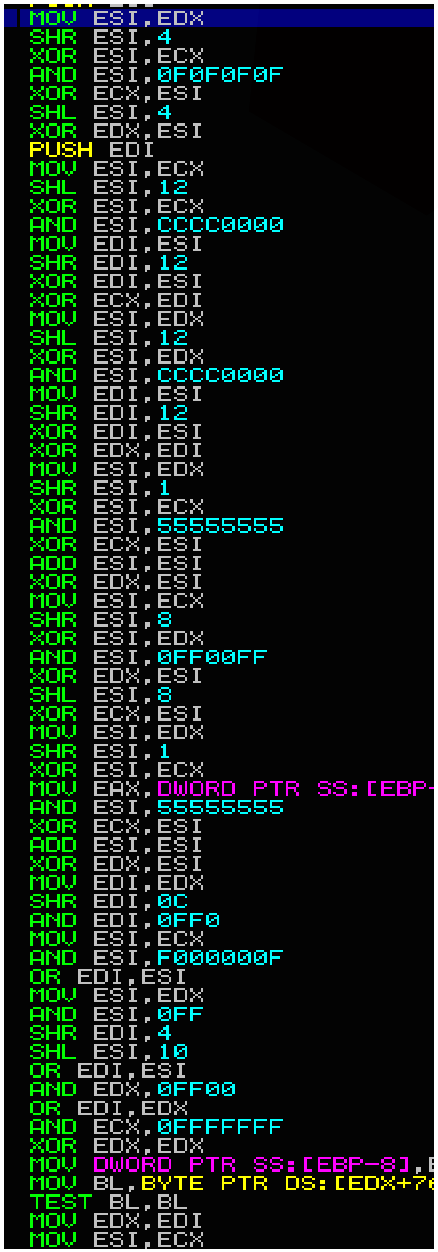
Better solution? Use bucket ranges. Split the range 0, 2^n into buckets and make C *lower_bound* <= *v* < *upper_bound* for each bucket. Direct == queries for all values in the valid buckets. Same worst case running time. Performs far better in many cases.



VARBOUNDS.PY # FINDING VARIABLE RANGES

Better solution? Use bucket ranges. Split the range 0, 2^n into buckets and make C *lower_bound* <= *v* < *upper_bound* for each bucket. Direct == queries for all values in the valid buckets. Same worst case running time. Performs far better in many cases.

Better still? For each bit *b* in *r* check if *b == 1* is satisfiable in order to find an absolute upper/lower bound before the previous stage. 32 queries, works well.



varbounds.py # Finding variable ranges

DEMO

varbounds.py # Finding variable ranges

We're still being rather dumb.

VARBOUNDS.PY # FINDING VARIABLE RANGES

We're still being rather dumb.

```
add eax, 10
mul ebx
rol ebx, ecx
```

Pure SMT solutions tend to have difficulty seeing the forest for the trees.

VARBOUNDS.PY # FINDING VARIABLE RANGES

We're still being rather dumb.

```
add eax, 10
mul ebx
rol ebx, ecx
```

Pure SMT solutions tend to have difficulty seeing the forest for the trees.

Often worth considering using an initial static analysis run first to limit cases and use a solver to resolve any remaining imprecision.

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

VARBOUNDS.PY # FINDING VARIABLE RANGES

We're still being rather dumb.

```
add eax, 10
mul ebx
rol ebx, ecx
```

Pure SMT solutions tend to have difficulty seeing the forest for the trees.

Often worth considering using an initial static analysis run first to limit cases and use a solver to resolve any remaining imprecision.

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

```
cmp eax, 0xFF
jbe EXIT
```

Quite a lot of latent information in arithmetic, logical and conditional branching instructions that can be extracted at varying expense

Abstract Interpretation in Brief and Vague "Detail"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

Abstract Interpretation in Brief and Vague "Detail"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

1) Pick an abstract domain

ABSTRACT INTERPRETATION IN BRIEF AND VAGUE "DETAIL"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

1) Pick an abstract domain

2) Build an abstraction function for each concrete instruction

Three valued logic
{
MUST_BE_SET,
CANT_BE_SET,
POSSIBLY_SET
}

Abstract Interpretation in Brief and Vague "Detail"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

1) Pick an abstract domain

2) Build an abstraction function for each concrete instruction

Three valued logic
{
MUST_BE_SET,
CANT_BE_SET,
POSSIBLY_SET
}

AND # f(x, y)
f(MBS, MBS) -> MBS
f(MBS, CBS) -> CBS
f(MBS, PBS) -> PBS
f(CBS, *) -> CBS
f(PBS, MBS) -> PBS
f(PBS, PBS) -> PBS
f(PBS, CBS) -> CBS

ABSTRACT INTERPRETATION IN BRIEF AND VAGUE "DETAIL"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

1) Pick an abstract domain

2) Build an abstraction function for each concrete instruction

3) Iterative analysis until a fixed point is reached

Three valued logic
{
    MUST_BE_SET,
    CANT_BE_SET,
    POSSIBLY_SET
}

AND # f(x, y)
f(MBS, MBS) -> MBS
f(MBS, CBS) -> CBS
f(MBS, PBS) -> PBS
f(CBS, *)   -> CBS
f(PBS, MBS) -> PBS
f(PBS, PBS) -> PBS
f(PBS, CBS) -> CBS

Abstract Interpretation in Brief and Vague "Detail"

Abstract Interpretation is a useful framework in many situations e.g. for *varbounds.py* and logical operators

1) Pick an abstract domain

2) Build an abstraction function for each concrete instruction

3) Iterative analysis until a fixed point is reached

Similar to our single bit queries but often far less computationally complex. In general makes many more problems tractable that a pure SMT based solution can't handle efficiently.

Three valued logic
{
MUST_BE_SET,
CANT_BE_SET,
POSSIBLY_SET
}

AND # f(x, y)
f(MBS, MBS) -> MBS
f(MBS, CBS) -> CBS
f(MBS, PBS) -> PBS
f(CBS, *) -> CBS
f(PBS, MBS) -> PBS
f(PBS, PBS) -> PBS
f(PBS, CBS) -> CBS

Abstract Interpretation in Brief and Vague "Detail"

Initial state: $EAX_{0\text{-}31}$ = PBS, $ECX_{0\text{-}7}$ = PBS, $ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

Abstract Interpretation in Brief and Vague "Detail"

Initial state: $EAX_{0\text{-}31}$ = PBS,
$ECX_{0\text{-}7}$ = PBS,
$ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

mov ebx, eax ; EBX_{0-31} = PBS

ABSTRACT INTERPRETATION IN BRIEF AND VAGUE "DETAIL"

Initial state: $EAX_{0\text{-}31}$ = PBS,
$ECX_{0\text{-}7}$ = PBS,
$ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

```
mov ebx, eax   ; EBX0-31 = PBS

add ebx, 0xFF ; NOP in this domain
```

Abstract Interpretation in brief and vague "detail"

Initial state: $EAX_{0\text{-}31}$ = PBS,
$ECX_{0\text{-}7}$ = PBS,
$ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

mov ebx, eax ; EBX_{0-31} = PBS

add ebx, 0xFF ; NOP in this domain

and ebx, 0xFF ; EBX_{0-7} = PBS, EBX_{8-31} = CBS

Abstract Interpretation in Brief and Vague "Detail"

Initial state: $EAX_{0\text{-}31}$ = PBS,
$ECX_{0\text{-}7}$ = PBS,
$ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

mov ebx, eax ; EBX_{0-31} = PBS

add ebx, 0xFF ; NOP in this domain

and ebx, 0xFF ; EBX_{0-7} = PBS, EBX_{8-31}= CBS

shl ebx, 0x10 ; EBX_{16-23} = PBS
; $EBX_{24-31,\ 0-15}$ = CBS

ABSTRACT INTERPRETATION IN BRIEF AND VAGUE "DETAIL"

Initial state: $EAX_{0\text{-}31}$ = PBS,
$ECX_{0\text{-}7}$ = PBS,
$ECX_{8\text{-}31}$ = MBS

```
mov ebx, eax
add ebx, 0xFF
and ebx, 0xFF
shl ebx, 0x10
and ebx, ecx
```

mov ebx, eax ; EBX_{0-31} = PBS

add ebx, 0xFF ; NOP in this domain

and ebx, 0xFF ; EBX_{0-7} = PBS, EBX_{8-31} = CBS

shl ebx, 0x10 ; EBX_{16-23} = PBS
; $EBX_{24-31,\ 0-15}$ = CBS

and ebx, ecx ; EBX_{16-23} = PBS
; $EBX_{24-31,\ 0-15}$ = CBS

Part II: One Hammer, Many Nails

ROP

Path Filtering

Finding variable bounds

Finding integer overflows

Loops and solvers

Performance/Scaling

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

SequenceAnalyzer() takes a list of bug checkers that will be called on each instruction (although if we use it through *PathWalker()* we get path filtering as well)

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

SequenceAnalyzer() takes a list of bug checkers that will be called on each instruction (although if we use it through *PathWalker()* we get path filtering as well)

```
def check_ins(self, sa, ins):
    res = None

    op = operations.operation(ins)
    disasm_str = op.removeLockPrefix()

    if disasm_str == "ADD":
        if self.debug:
            self.imm.log("check_ins (%s): ADD" % hex(ins.getAddress()),
                         ins.getAddress())
        solver = sa.state.solver

        dst = sa.buildState(ins.operand[0])
        src = sa.buildState(ins.operand[1])
        dst_val = sa.getValueFromState(dst)
        src_val = sa.getValueFromState(src)

        res_64 = solver.addExpr(dst_val, src_val, 64)

        gt_expr = solver.gtExpr(res_64, solver.constExpr(MAX_INT_32))
        status = solver.checkSat(gt_expr)

        if status:
            if self.debug:
                self.imm.log("check_ins (%s): Bug found" % \
                             hex(ins.getAddress()))
            res = BugCheckResults(ins.getAddress(),
                                  solver.getConcreteModel())
```

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

SequenceAnalyzer() takes a list of bug checkers that will be called on each instruction (although if we use it through *PathWalker()* we get path filtering as well)

Represent the *ADD* in the current context

```
def check_ins(self, sa, ins):
    res = None

    op = operations.operation(ins)
    disasm_str = op.removeLockPrefix()

    if disasm_str == "ADD":
        if self.debug:
            self.imm.log("check_ins (%s): ADD" % hex(ins.getAddress()),
                         ins.getAddress())
        solver = sa.state.solver

        dst = sa.buildState(ins.operand[0])
        src = sa.buildState(ins.operand[1])
        dst_val = sa.getValueFromState(dst)
        src_val = sa.getValueFromState(src)

        res_64 = solver.addExpr(dst_val, src_val, 64)

        gt_expr = solver.gtExpr(res_64, solver.constExpr(MAX_INT_32))
        status = solver.checkSat(gt_expr)

        if status:
            if self.debug:
                self.imm.log("check_ins (%s): Bug found" % \
                             hex(ins.getAddress()))
            res = BugCheckResults(ins.getAddress(),
                                  solver.getConcreteModel())
```

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

SequenceAnalyzer() takes a list of bug checkers that will be called on each instruction (although if we use it through *PathWalker()* we get path filtering as well)

Represent the *ADD* in the current context

Check if the result is > 2^{32}

```
def check_ins(self, sa, ins):
    res = None

    op = operations.operation(ins)
    disasm_str = op.removeLockPrefix()

    if disasm_str == "ADD":
        if self.debug:
            self.imm.log("check_ins (%s): ADD" % hex(ins.getAddress()),
                         ins.getAddress())
        solver = sa.state.solver

        dst = sa.buildState(ins.operand[0])
        src = sa.buildState(ins.operand[1])
        dst_val = sa.getValueFromState(dst)
        src_val = sa.getValueFromState(src)

        res_64 = solver.addExpr(dst_val, src_val, 64)

        gt_expr = solver.gtExpr(res_64, solver.constExpr(MAX_INT_32))
        status = solver.checkSat(gt_expr)

        if status:
            if self.debug:
                self.imm.log("check_ins (%s): Bug found" % \
                             hex(ins.getAddress()))
            res = BugCheckResults(ins.getAddress(),
                                  solver.getConcreteModel())
```

find_int_overflow.py # what it sounds like

DEMO

find_int_overflow.py # What it sounds like

In order to limit false positives this BugChecker is ideally used with a concrete memory/register model containing a few free variables.

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

In order to limit false positives this BugChecker is ideally used with a concrete memory/register model containing a few free variables.

Necessary to use lazy warning of bugs e.g. where the result of an overflow is used as opposed to where it occurs. Many overflows are later checked for and benign.

```
unsigned a, alloc_size;
alloc_size = get_from_user();

a = alloc_size + 1;

if (alloc_size > 0xff)
        goto exit;

x = malloc(a);
```

FIND_INT_OVERFLOW.PY # WHAT IT SOUNDS LIKE

In order to limit false positives this BugChecker is ideally used with a concrete memory/register model containing a few free variables.

Necessary to use lazy warning of bugs e.g. where the result of an overflow is used as opposed to where it occurs. Many overflows are later checked for and benign.

```
unsigned a, alloc_size;
alloc_size = get_from_user();

a = alloc_size + 1;

if (alloc_size > 0xff)
        goto exit;

x = malloc(a);
```

See Julien Vanegue's HES 2009 talk for a more developed idea focusing on integer overflows on the size parameter to heap allocations

INFRASTRUCTURE

SequenceAnalyzer – Models x86 as operations over a set of SMT primitives.

Solver – Ctypes interface to the CVC3 SMT solver API. Supports a variety of theories including quantifier free, bit-vector arithmetic, linear arithmetic etc.

CodeGraph/PathGenerator – Purely static CFG building and path generation.

PathWalker – SMT based path traversal. Each conditional jump is checked for feasibilty and the path discarded if not SAT.

BugChecker – Subclasses provide the check_ins method which will be passed the SMT context representing the current path.

Part II: One Hammer, Many Nails

ROP

↓

Path Filtering

↓

Finding variable bounds

↓

Finding integer overflows

↓

Loops and solvers

↓

Performance/Scaling

Loops in the World of SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

Loops in the world of SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

We can simulate ordering and control flow relatively well.

add ebx, 0xFF => (= ebx^1, (+ ebx^0, 0xFF)

Loops in the world of SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

We can simulate ordering and control flow relatively well.

add ebx, 0xFF => (= ebx^1, (+ ebx^0, 0xFF)

Loops however have no natural representation. Only option is to unroll. How?

Loops in the world of SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

We can simulate ordering and control flow relatively well.

add ebx, 0xFF =>
(= ebx^1, (+ ebx^0, 0xFF)

Loops however have no natural representation. Only option is to unroll. How?

1) Prove loop bounds

Loops in the world of SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

We can simulate ordering and control flow relatively well.

add ebx, 0xFF =>
(= ebx^1, (+ ebx^0, 0xFF)

Loops however have no natural representation. Only option is to unroll. How?

1) Prove loop bounds

2) Guess loop bounds.

LOOPS IN THE WORLD OF SMT/SAT?

Propositional logic has no concept of ordering, control flow or loops

We can simulate ordering and control flow relatively well.

add ebx, 0xFF =>
(= ebx^1, (+ ebx^0, 0xFF)

Loops however have no natural representation. Only option is to unroll. How?

1) Prove loop bounds

2) Guess loop bounds.

3) Approximate loop bounds

Part II: One Hammer, Many Nails

ROP

↓

Path Filtering

↓

Finding variable bounds

↓

Finding integer overflows

↓

Loops and solvers

↓

Performance/Scaling

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

Scaling of solvers on real world cases[2]

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

[2] See the SMT-COMP website for objective comparisons of different solvers. Many of the problems are derived from symbolic execution on real world problems

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

Scaling of solvers on real world cases[2]

Pure SMT/SAT based approaches can scale poorly due to the complexity of the generated constraints OR by being a poor match for the problem domain

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

[2] See the SMT-COMP website for objective comparisons of different solvers. Many of the problems are derived from symbolic execution on real world problems

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

Scaling of solvers on real world cases[2]

Pure SMT/SAT based approaches can scale poorly due to the complexity of the generated constraints OR by being a poor match for the problem domain

Whole program analysis – not really my interest, realistically not feasible for many tasks we care about e.g. general bug detection in real programs (in both dev time and scalability, we're not a university research group)

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

[2] See the SMT-COMP website for objective comparisons of different solvers. Many of the problems are derived from symbolic execution on real world problems

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

Scaling of solvers on real world cases[2]

Pure SMT/SAT based approaches can scale poorly due to the complexity of the generated constraints OR by being a poor match for the problem domain

Whole program analysis – not really my interest, realistically not feasible for many tasks we care about e.g. general bug detection in real programs (in both dev time and scalability, we're not a university research group)

Task based analysis – much more promising IMO. Research problem – Which tasks? How to integrate/interface?

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

[2] See the SMT-COMP website for objective comparisons of different solvers. Many of the problems are derived from symbolic execution on real world problems

PERFORMANCE/SCALING

Scaling of raw solver capabilities[1]

Scaling of solvers on real world cases[2]

Pure SMT/SAT based approaches can scale poorly due to the complexity of the generated constraints OR by being a poor match for the problem domain

Whole program analysis – not really my interest, realistically not feasible for many tasks we care about e.g. general bug detection in real programs (in both dev time and scalability, we're not a university research group)

Task based analysis – much more promising IMO. Research problem – Which tasks? How to integrate/interface?

More investigation required to discover typical formula complexity for general RE/Exploit dev problems. SMT-COMP has no such tests.

[1] I'm currently gathering results for some well defined instances. Should be up on http://seanhn.wordpress.com soon

[2] See the SMT-COMP website for objective comparisons of different solvers. Many of the problems are derived from symbolic execution on real world problems

CONCLUSION

SMT solving tech has evolved quite a lot over the past decade to the point where it is usable as an integrated part of our exploit dev and RE processes.

CONCLUSION

SMT solving tech has evolved quite a lot over the past decade to the point where it is usable as an integrated part of our exploit dev and RE processes.

Trying to replace a human completely isn't the goal, supporting them is.

CONCLUSION

SMT solving tech has evolved quite a lot over the past decade to the point where it is usable as an integrated part of our exploit dev and RE processes.

Trying to replace a human completely isn't the goal, supporting them is.

Figuring out the best way to integrate automated analysis into our work flows is a research problem in itself. Better interfaces, we need them.

CONCLUSION

SMT solving tech has evolved quite a lot over the past decade to the point where it is usable as an integrated part of our exploit dev and RE processes.

Trying to replace a human completely isn't the goal, supporting them is.

Figuring out the best way to integrate automated analysis into our work flows is a research problem in itself. Better interfaces, we need them.

Some problems are naturally ported to this domain (path filtering) while others (variable bound checking) are more of a cludge. More still (finding arithmetic flaws) suit collaboration with basic static analysis and a human.


QUESTIONS?
SEAN HEELAN
SEAN@IMMUNITYINC.COM