Microsoft

Microsoft Research

Microsoft Research

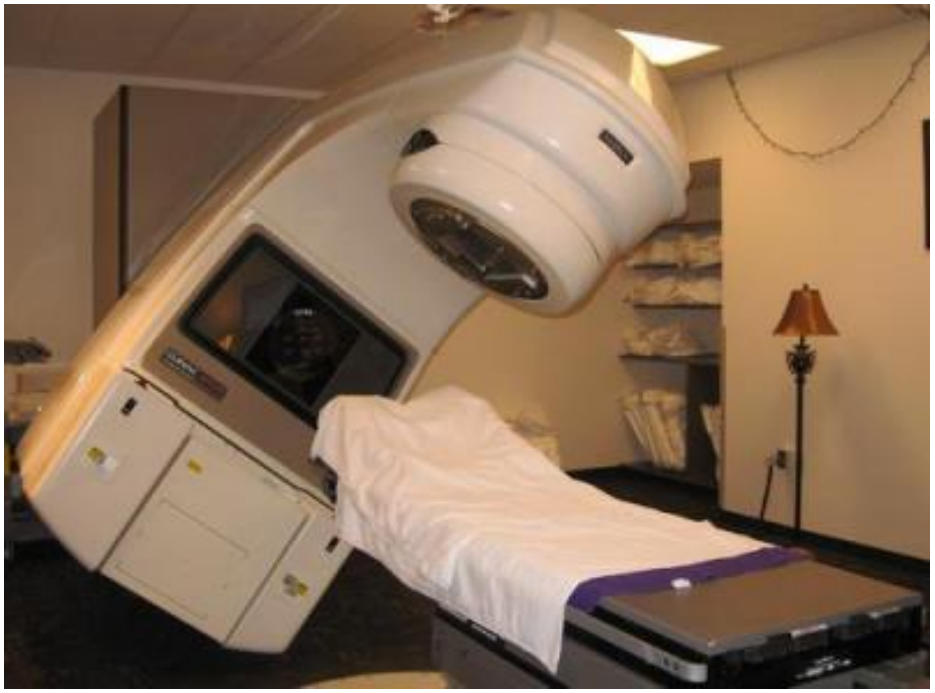
Microsoft

Automatic Software Verification with SMT

Nuno Lopes

Deadly Software Bugs

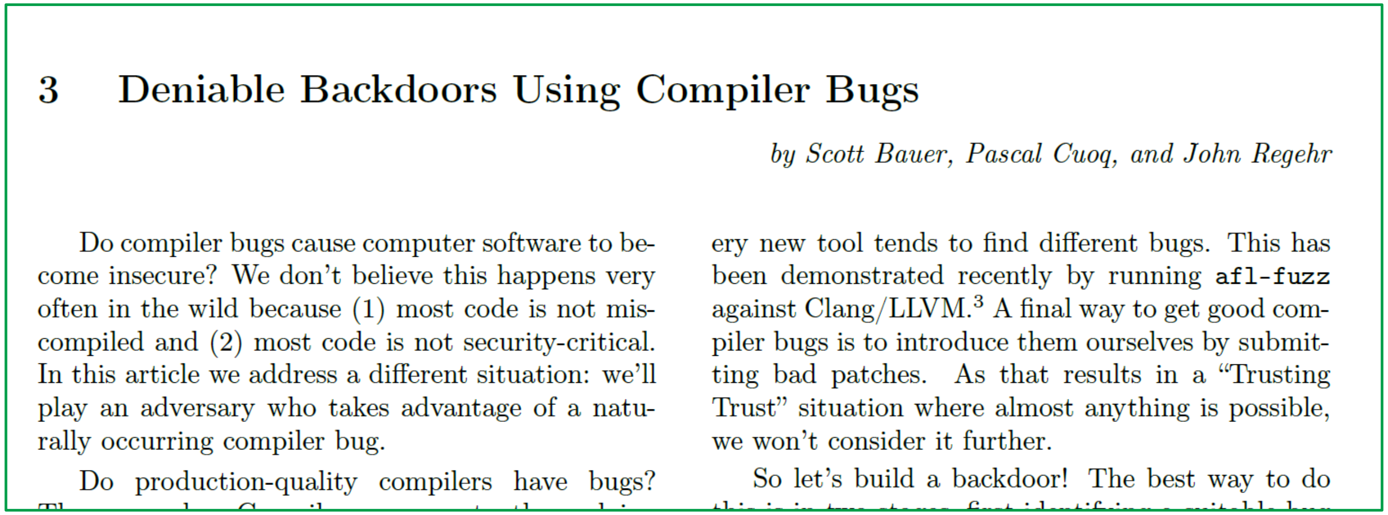
Ariane 5



Therac-25

Compiler bugs

- New trend to exploit compiler bugs to introduce backdoors
- Major projects to verify compilers (LLVM, Visual Studio C++)



3 **Deniable Backdoors Using Compiler Bugs**

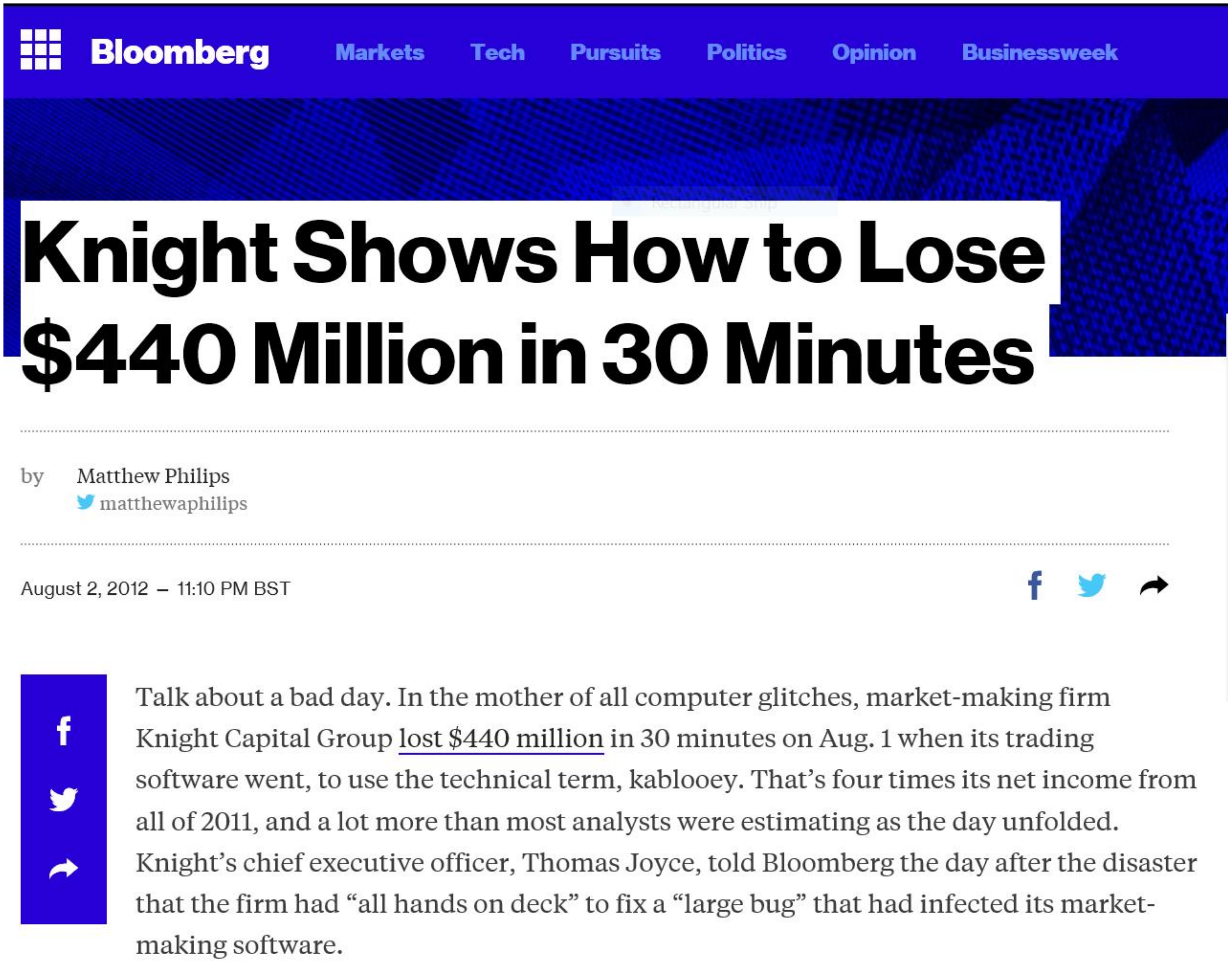
by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs?

ery new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do

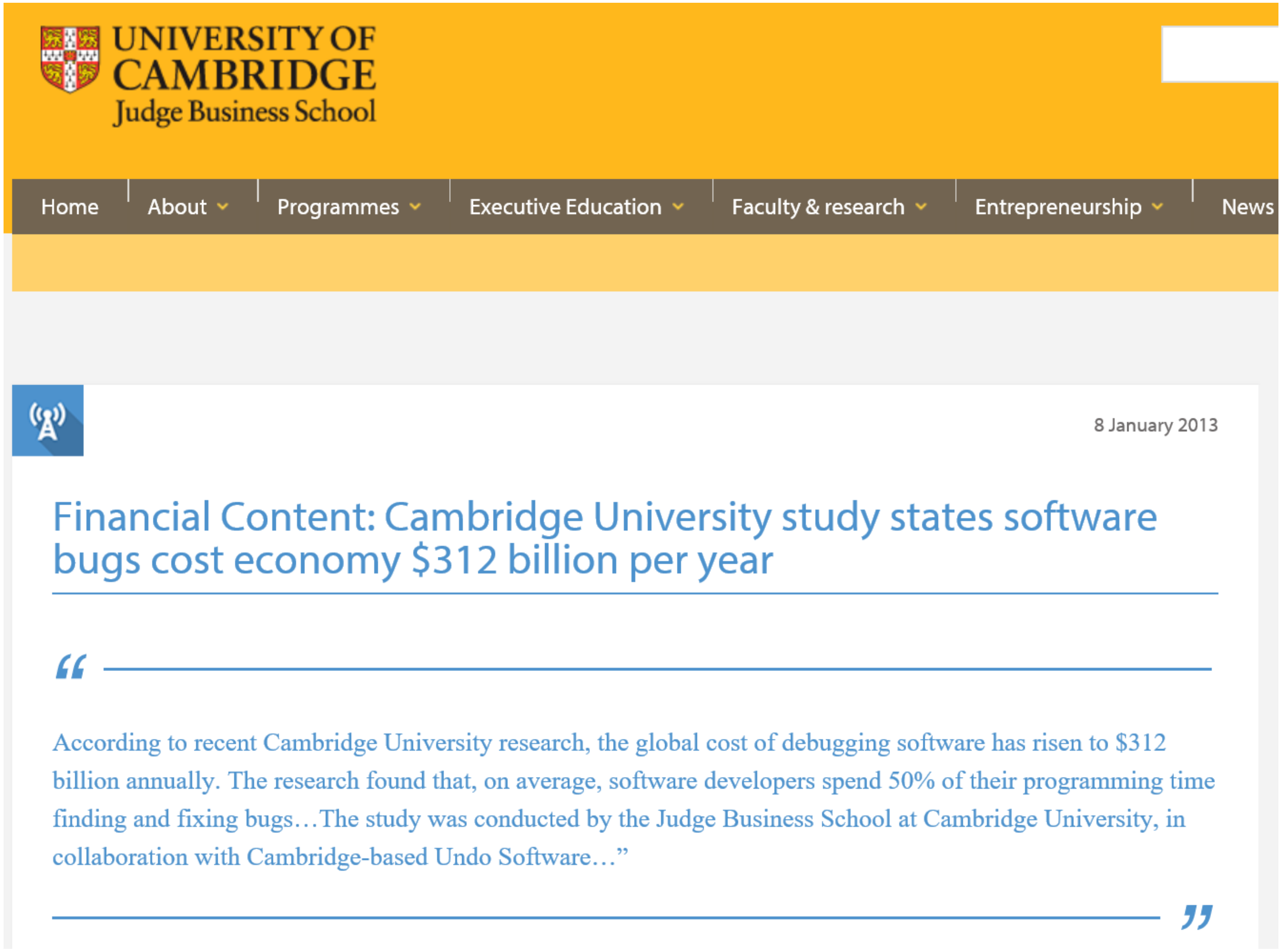


Knight Shows How to Lose $440 Million in 30 Minutes

by Matthew Philips
matthewaphilips

August 2, 2012 – 11:10 PM BST

Talk about a bad day. In the mother of all computer glitches, market-making firm Knight Capital Group lost $440 million in 30 minutes on Aug. 1 when its trading software went, to use the technical term, kablooey. That's four times its net income from all of 2011, and a lot more than most analysts were estimating as the day unfolded. Knight's chief executive officer, Thomas Joyce, told Bloomberg the day after the disaster that the firm had "all hands on deck" to fix a "large bug" that had infected its market-making software.

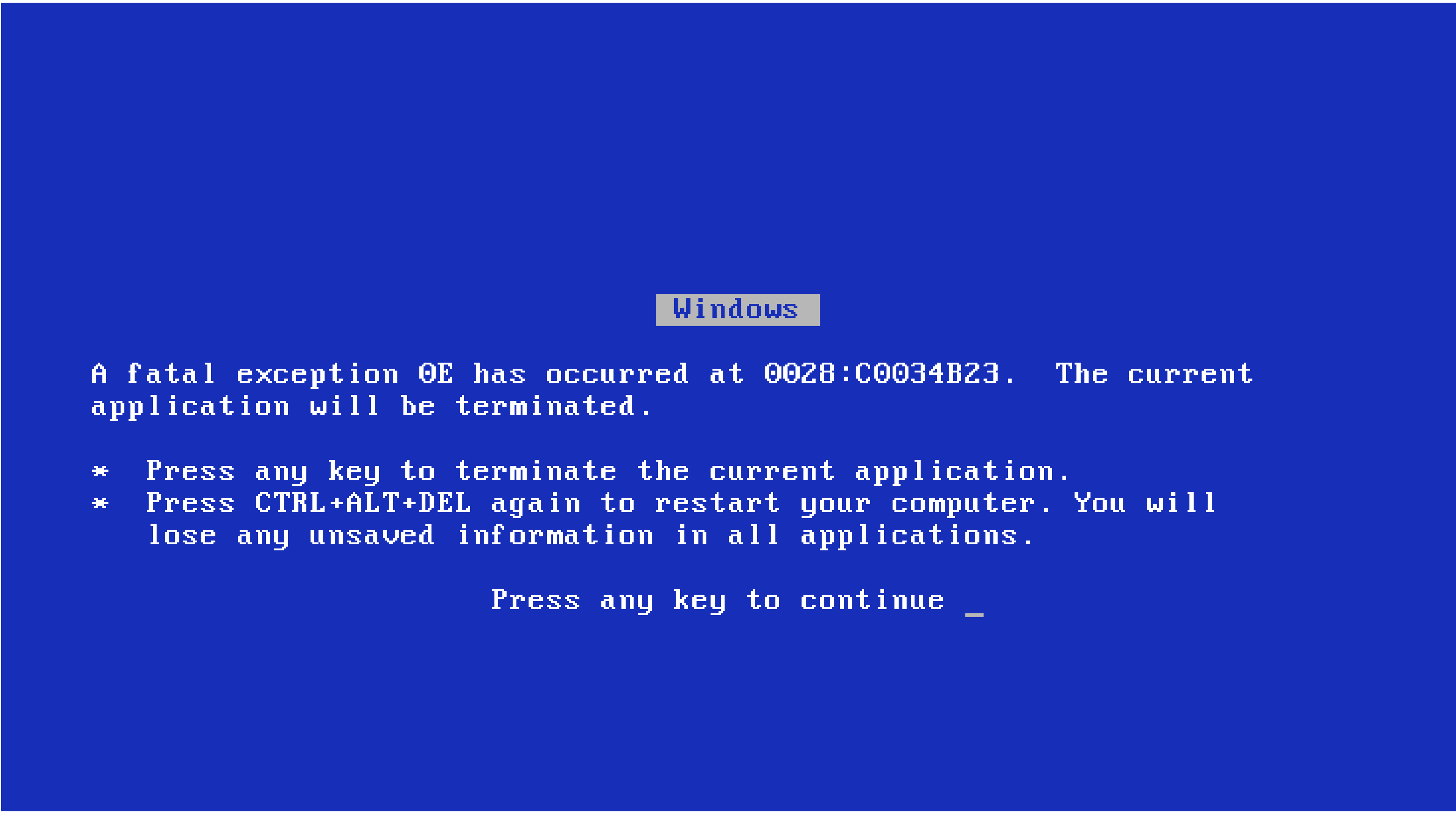


8 January 2013

Financial Content: Cambridge University study states software bugs cost economy $312 billion per year

"According to recent Cambridge University research, the global cost of debugging software has risen to $312 billion annually. The research found that, on average, software developers spend 50% of their programming time finding and fixing bugs…The study was conducted by the Judge Business School at Cambridge University, in collaboration with Cambridge-based Undo Software…"

Bugs are Annoying too..



```
                              Windows

A fatal exception 0E has occurred at 0028:C0034B23.  The current
application will be terminated.

*  Press any key to terminate the current application.
*  Press CTRL+ALT+DEL again to restart your computer. You will
   lose any unsaved information in all applications.

                      Press any key to continue _
```

Why should **YOU** care?

- Be the hero: save lives and/or money
- Be the hero: save us from hackers
- Be the hero: save us from blue screens

- **All** major companies hiring in this area

Can this assert() crash?

```
void f(int n) {
  int j = 0;
  for (int i = 0; i < n; ++i) {
    j += 1;
  }
  assert(j >= n);
}
```

n = 0, j = 0 ✓
n = 1, j = 1 ✓
…
n = -1, j = 0 ✓
n = -2, j = 0 ✓

Today

- Applying SMT to software verification

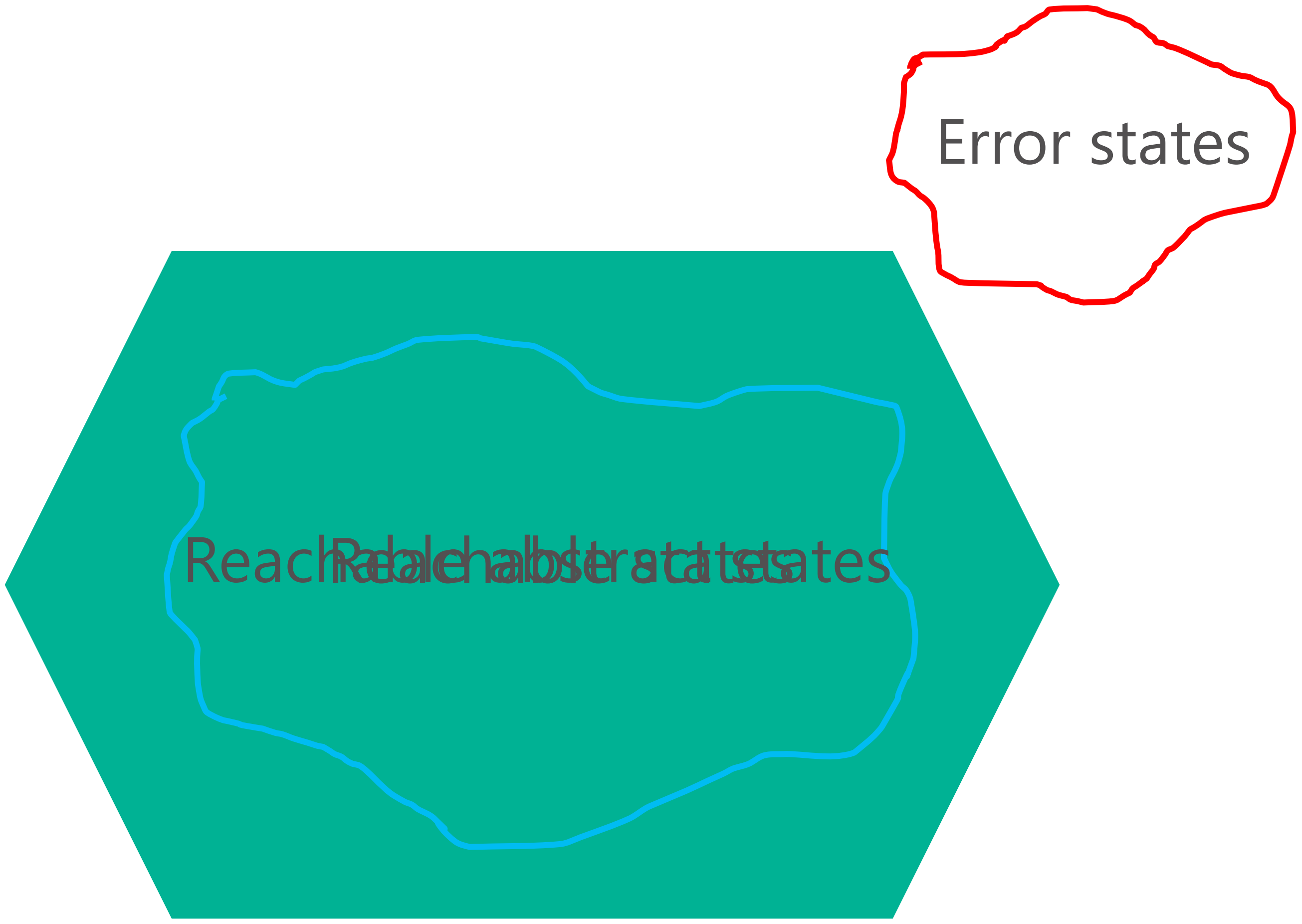
- SMT: learning from program analysis

Program Analysis

- Abstract interpretation / static analysis
- Symbolic execution
- Bounded model checking (BMC)
- Model checking

- Different clients: program verification, bug finding, compiler optimization, refactoring, code metrics, …

Verification



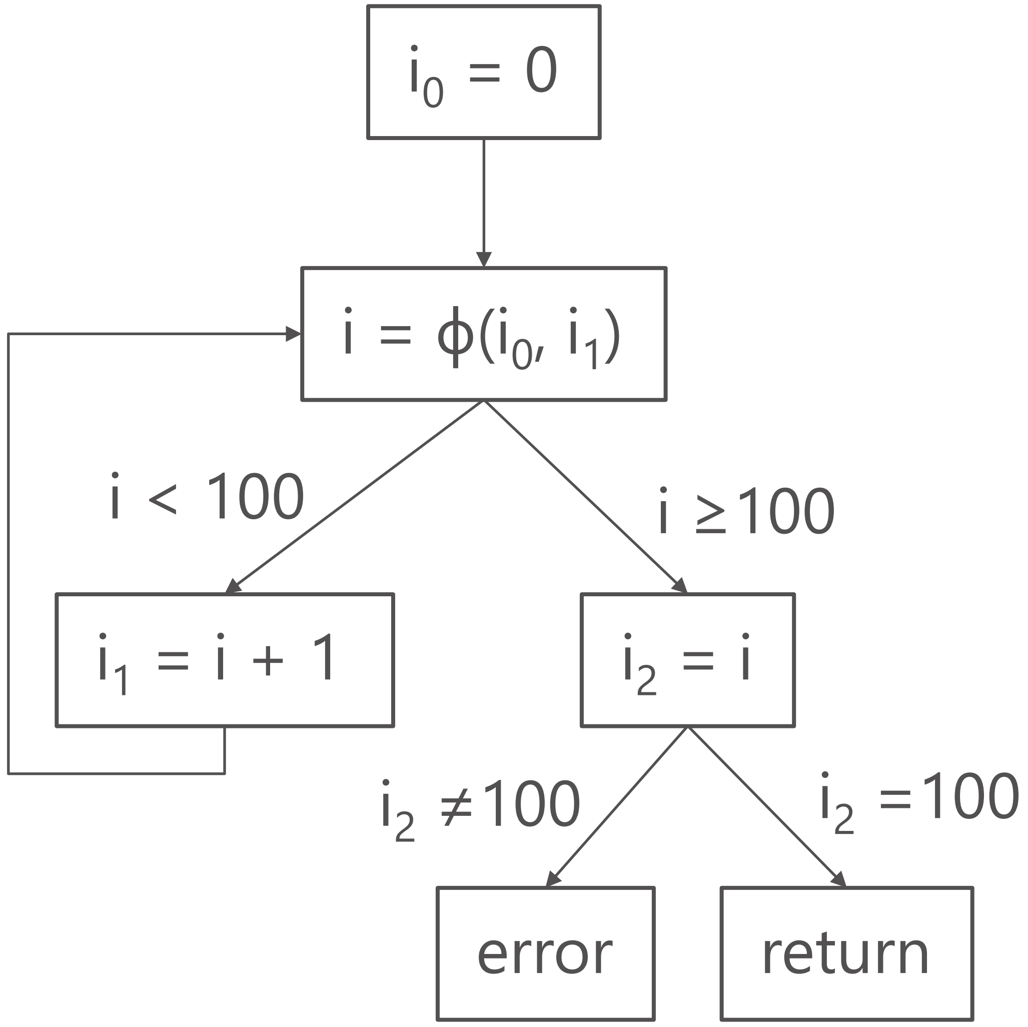
Abstract Interpretation (AI)

- Execute program with abstract values (e.g., intervals)
- Widening: converge faster (loses completeness)

- Abstraction: forget (potentially) irrelevant information

AI: example

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



Interval domain

Interval:

$\{[l, h] \mid l \leq h, l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}\}$

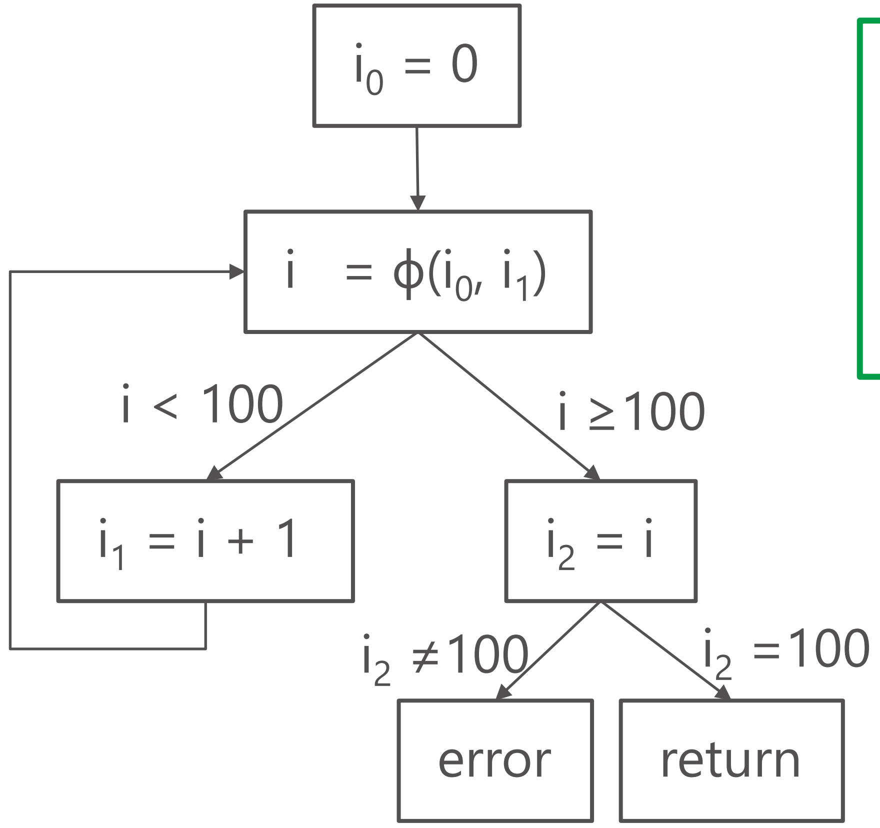
Abstraction function α:

α(s) = [min(s), max(s)]

Example:

α({1, 3}) = [1, 3]

AI with intervals



$$i_0 = [0,0]$$
$$\mathrm{i}' = (i_0 \cup i_1) \cap [-\infty, 99]$$
$$i_1 = i' + [1,1]$$
$$i_2 = (i_0 \cup i_1) \cap [100, +\infty]$$

Does this always hold?

$$i_2 \cap (i_2 \neq 100) = \emptyset$$

AI: least fixed-point (lfp)

$$\begin{aligned} i_0 &= [0,0] \\ \mathrm{i}' &= (i_0 \cup i_1) \cap [-\infty, 99] \\ i_1 &= i' + [1,1] \\ i_2 &= (i_0 \cup i_1) \cap [100, +\infty] \end{aligned}$$

1st iteration	2nd	3rd	4th	5th
$i_0 = [0,0]$	$i_0 = [0,0]$	$i_0 = [0,0]$	$i_0 = [0,0]$	$i_0 = [0,0]$
$i' = \emptyset$	$i' = [0,0]$	$i' = [0,0]$	$i' = [0,1]$	$i' = [0,1]$
$i_1 = \emptyset$	$i_1 = \emptyset$	$i_1 = [1,1]$	$i_1 = [1,1]$	$i_1 = [1,2]$
$i_2 = \emptyset$	$i_2 = \emptyset$	$i_2 = \emptyset$	$i_2 = \emptyset$	$i_2 = \emptyset$

AI: least fixed-point (lfp)

$$\begin{aligned} i_0 &= [0,0] \\ \mathrm{i}' &= (i_0 \cup i_1) \cap [-\infty, 99] \\ i_1 &= i' + [1,1] \\ i_2 &= (i_0 \cup i_1) \cap [100, +\infty] \end{aligned}$$

$$\begin{aligned} i_0 &= [0,0] \\ i' &= \emptyset \\ i_1 &= \emptyset \\ i_2 &= \emptyset \end{aligned}$$

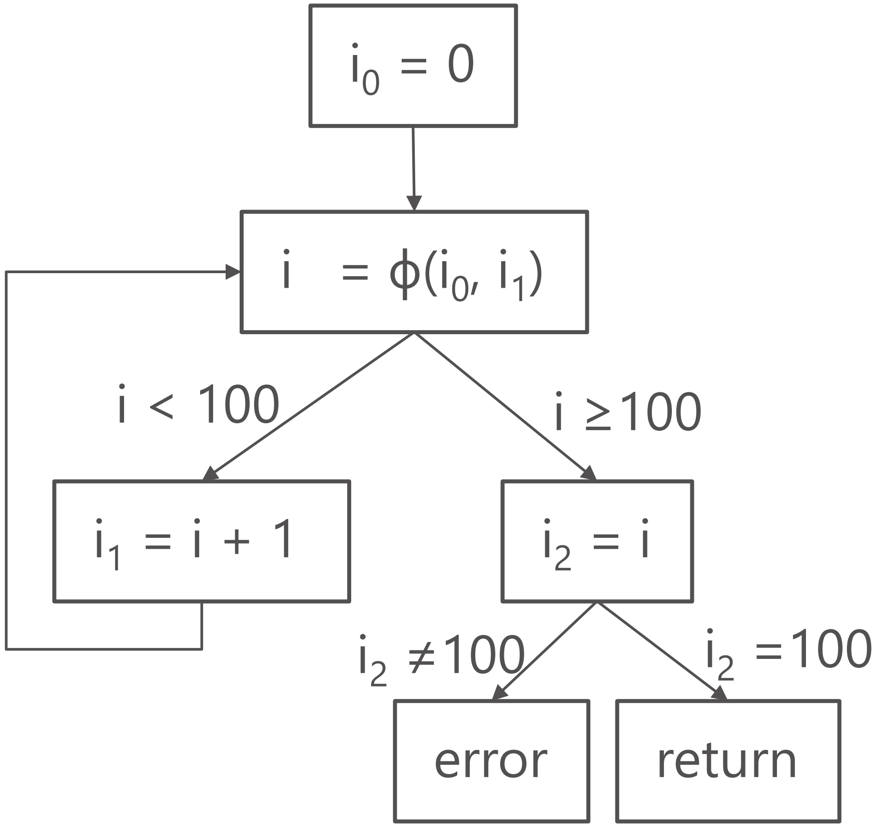
1st iteration

…

$$\begin{aligned} i_0 &= [0,0] \\ i' &= [0,99] \\ i_1 &= [1,100] \\ i_2 &= [100,100] \end{aligned}$$

last iteration

AI: least fixed-point (lfp)



$$
\begin{aligned}
i_0 &= [0,0] \\
i' &= (i_0 \cup i_1) \cap [-\infty, 99] \\
i_1 &= i' + [1,1] \\
i_2 &= (i_0 \cup i_1) \cap [100, +\infty]
\end{aligned}
$$

$$
\begin{aligned}
i_0 &= [0,0] \\
i' &= [0,99] \\
i_1 &= [1,100] \\
i_2 &= [100,100]
\end{aligned}
$$

Does this always hold?

$i_2 \cap (i_2 \neq 100) = \emptyset$

AI: Widening

- What we just did was super slow!
- Faster to execute program

- Widening: converge lfp faster:

$$[l, h] \nabla [\mathrm{l}', \mathrm{h}'] = [\mathrm{if\ l}' < \mathrm{l\ \ then} \ -\infty \ \mathrm{else\ l},$$
$$\mathrm{if\ h}' > \mathrm{h\ then} + \infty \ \mathrm{else\ h}]$$

AI: lfp with widening

$$\begin{aligned} i_0 &= [0,0] \\ \mathrm{i}' &= (i' \,\nabla (i_0 \cup i_1)) \cap [-\infty, 99] \\ i_1 &= i' + [1,1] \\ i_2 &= (i_0 \cup i_1) \cap [100, +\infty] \end{aligned}$$

$$\begin{aligned} i_0 &= [0,0] \\ i &= [0,99] \\ i_1 &= [1,100] \\ i_2 &= [100,100] \end{aligned}$$

$$\begin{aligned} i_0 &= [0,0] \\ i' &= \emptyset \\ i_1 &= \emptyset \\ i_2 &= \emptyset \end{aligned}$$

1st iteration

$$\begin{aligned} i_0 &= [0,0] \\ i' &= [0,0] \\ i_1 &= \emptyset \\ i_2 &= \emptyset \end{aligned}$$

2nd

$$\begin{aligned} i_0 &= [0,0] \\ i' &= [0,0] \\ i_1 &= [1,1] \\ i_2 &= \emptyset \end{aligned}$$

3rd

$$\begin{aligned} i_0 &= [0,0] \\ i' &= [0,99] \\ i_1 &= [1,1] \\ i_2 &= \emptyset \end{aligned}$$

4th

$$\begin{aligned} i_0 &= [0,0] \\ i' &= [0,99] \\ i_1 &= [1,100] \\ i_2 &= \emptyset \end{aligned}$$

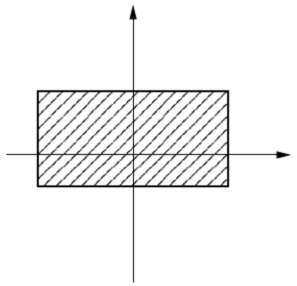
5th

Abstract Interpretation

- Least fixed-point over equations
- Each iteration: multiple abstract domain operations
  - Arithmetic operations, union, conjunction, etc
  - Usually **not** implemented with SMT

- SMT can be used to compute optimal results

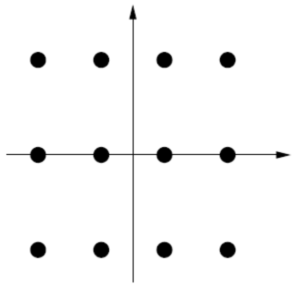
AI: Many domains



Intervals

$\boldsymbol{X_i \in [a_i, b_i]}$

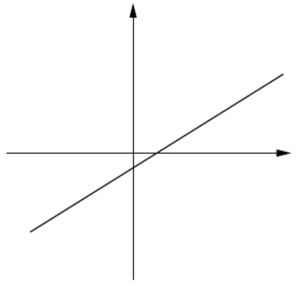
[Cousot-Cousot-76]



Simple Congruences

$\boldsymbol{X_i \equiv a_i\ [b_i]}$

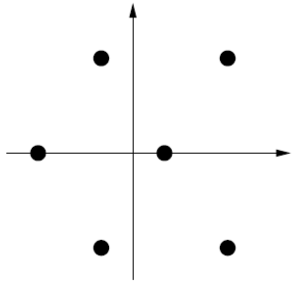
[Granger-89]



Linear Equalities

$\sum_i \alpha_i \boldsymbol{X_i} = \beta$

[Karr-76]



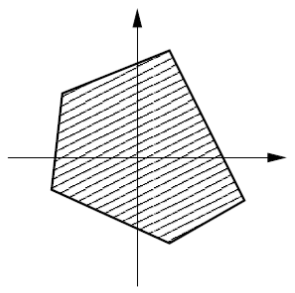
Linear Congruences

$\sum_i \alpha_i \boldsymbol{X_i} \equiv \beta\ [\gamma]$

[Granger-91]

Bertrand Jeannet, Antoine Miné [CAV'09]

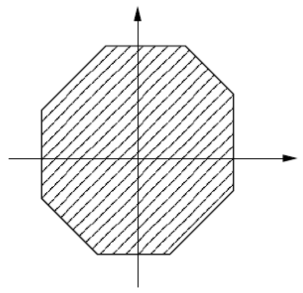
AI: Many domains



Polyhedra

$\sum_i \alpha_i X_i \geq \beta$

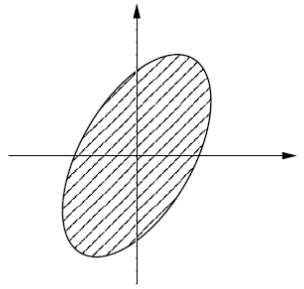
[Cousot-Halbwachs-78]



Octagons

$\pm X_i \pm X_j \leq \beta$

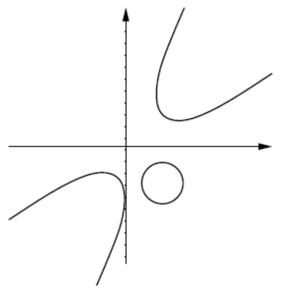
[Miné-01]



Ellipsoids

$\alpha X^2 + \beta Y^2 + \gamma XY \leq \delta$

[Feret-04]



Varieties

$P(\vec{X}) = 0,\ P \in \mathbb{R}[\mathrm{Var}]$

[Sankaranarayanan-Sipma-Manna-04

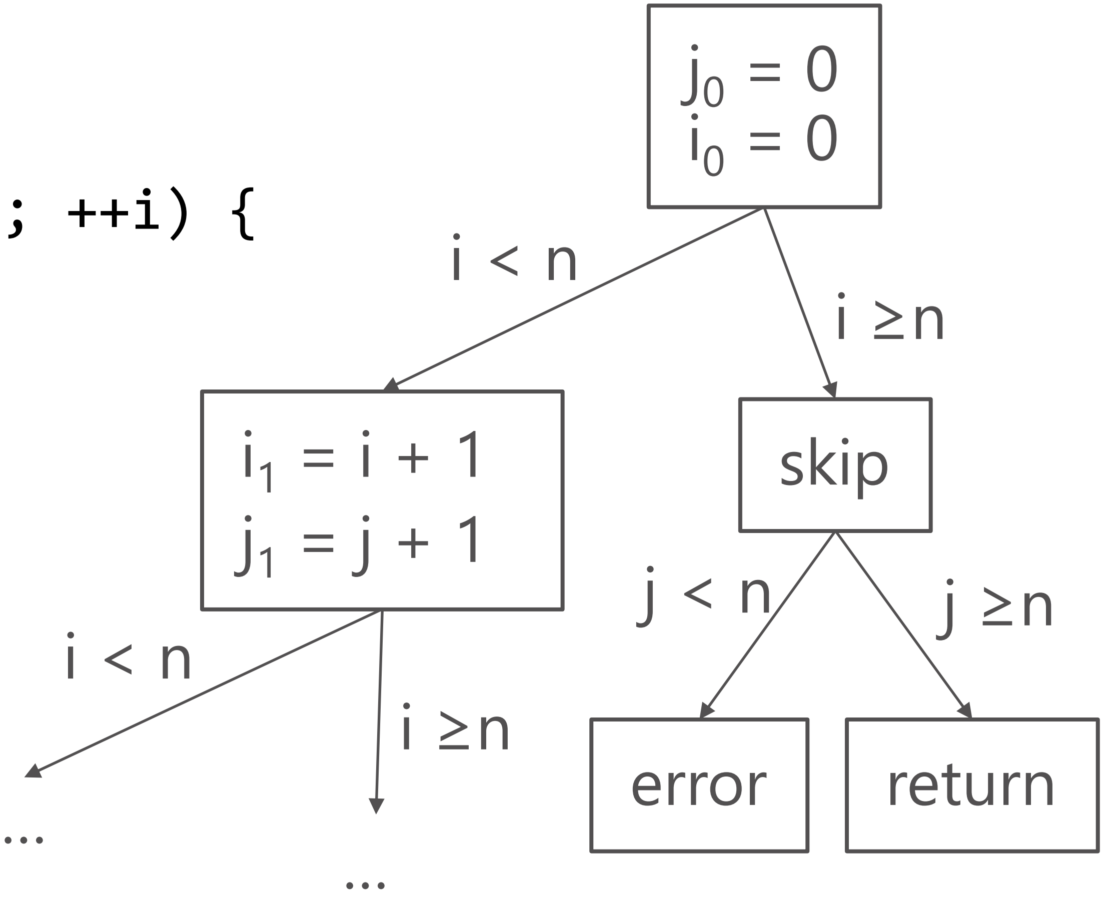
Bertrand Jeannet, Antoine Miné [CAV'09]

Symbolic execution

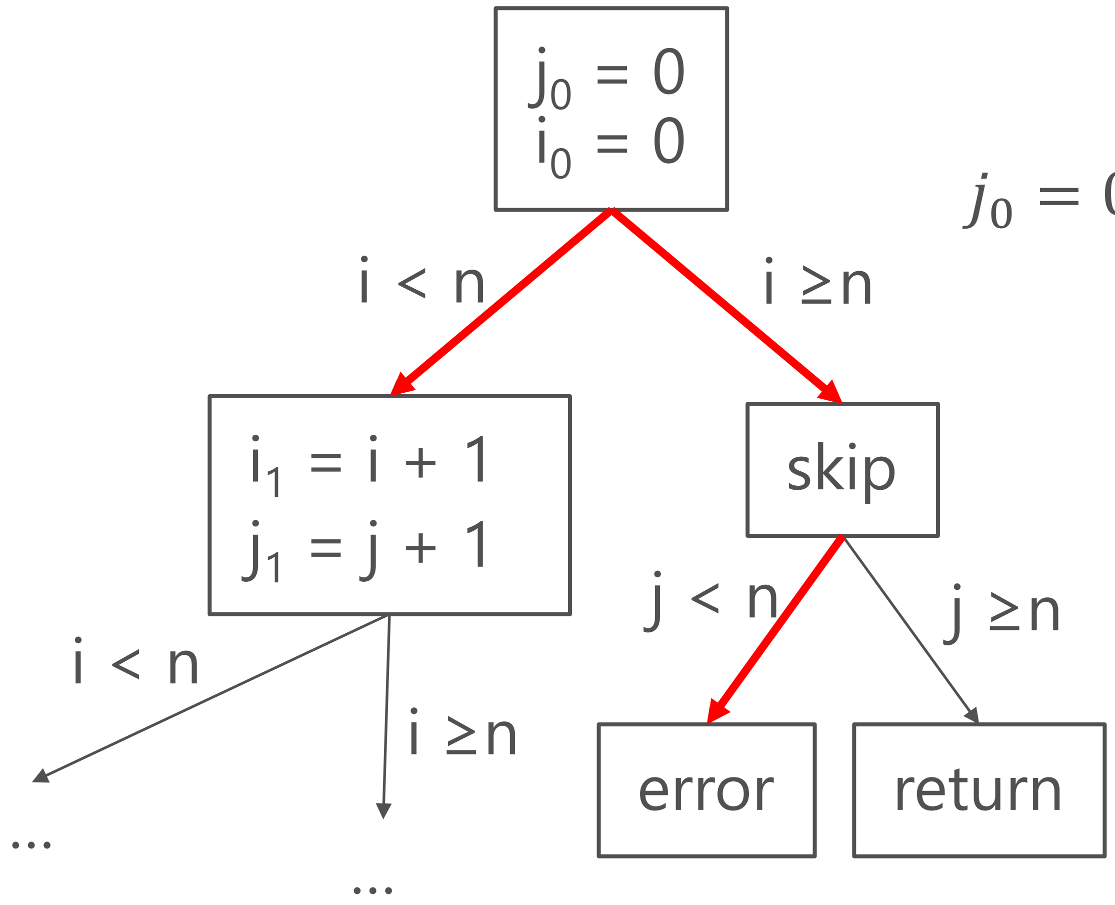
- Execute program with an SMT solver
- Mostly for bug finding; usually doesn’t terminate
- Usually 1 query per branch

SE: example

```
void f(int n) {
  int j = 0;
  for (int i = 0; i < n; ++i) {
    j += 1;
  }
  assert(j >= n);
}
```



SE: example



SMT queries

$$j_0 = 0 \land i_0 = 0 \land i_0 < n$$

$$j_0 = 0 \land i_0 = 0 \land i_0 \geq n \land j_0 < n$$

SE: implementation

- Incremental queries along a path
- fork() to copy SMT solver state (on branching)
- Usually BFS, easily parallelizable
- Mixed SAT/UNSAT queries

BMC: Bounded Model Checking

- Similar to symbolic execution
- But usually encodes whole program in single query
- 1 query per unfolding

BMC: example

```
void f(int n) {
  int j = 0;
  for (int i = 0; i < n; ++i) {
    j += 1;
  }
  assert(j >= n);
}
```

0 unfoldings

```
void f(int n) {
  int j = 0;
  int i = 0;
  assume(i >= n);
  assert(j >= n);
}
```

1 unfolding

```
void f(int n) {
  int j = 0;
  int i = 0;
  assume(i < n);
  j += 1;
  ++i;
  assume(i >= n);
  assert(j >= n);
}
```

BMC: SMT queries

0 unfoldings

```
void f(int n) {
  int j = 0;
  int i = 0;
  assume(i >= n);
  assert(j >= n);
}
```

1 unfolding

```
void f(int n) {
  int j = 0;
  int i = 0;
  assume(i < n);
  j += 1;
  ++i;
  assume(i >= n);
  assert(j >= n);
}
```

$$j_0 = 0 \land\ i_0 = 0 \land \mathrm{i}_0 \geq \mathrm{n} \land j_0 < n$$

$$j_0 = 0 \land\ i_0 = 0 \land \mathrm{i}_0 < \mathrm{n} \land \mathrm{i}_1 = \mathrm{i}_0 + 1 \land \mathrm{j}_1 = \mathrm{j}_0 + 1 \land \mathrm{i}_1 \geq \mathrm{n} \land j_1 < n$$

BMC: implementation

- Mostly UNSAT queries (SAT = bug in program)
- Not incremental

- Advanced: termination with k-induction (not covered)

Model Checking

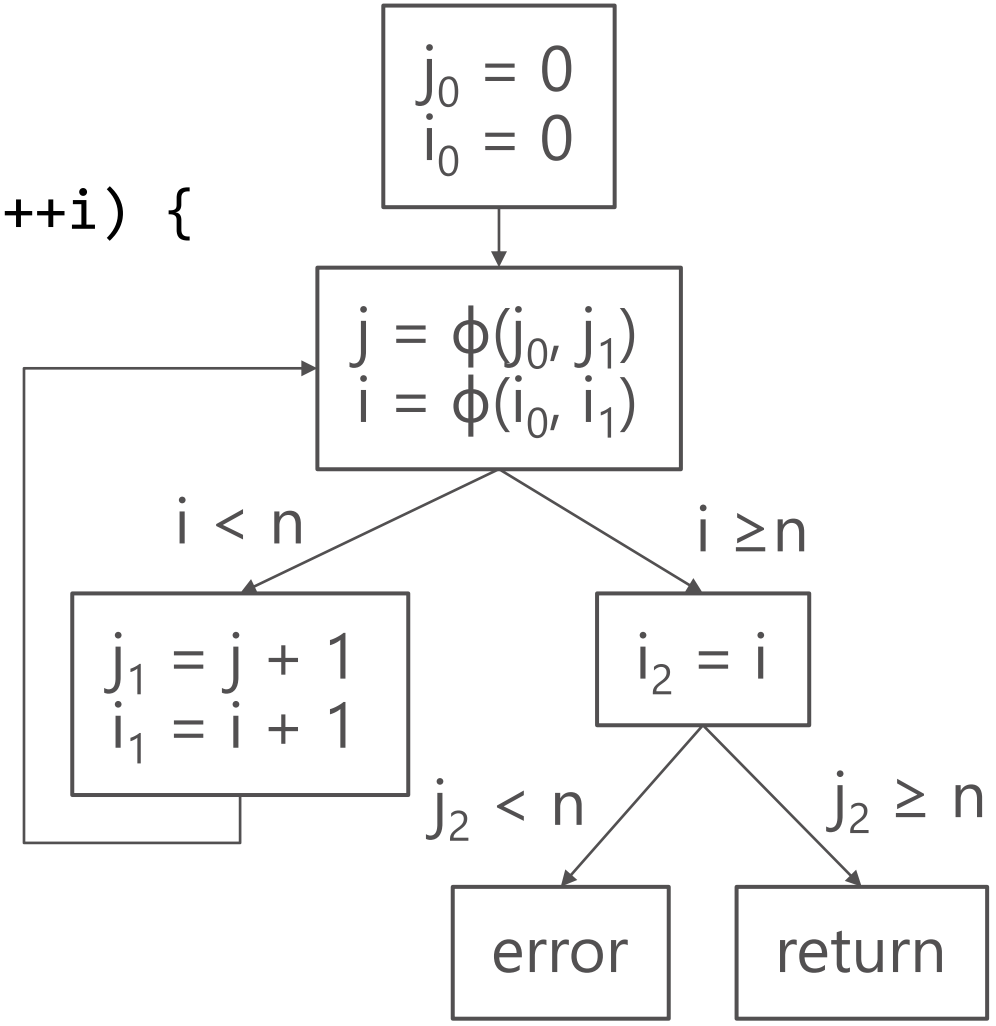
- Abstract Interpretation, but abstraction is built as needed
- CEGAR loop to refine abstraction (counterexample-guided abstraction refinement)
- Abstraction: as coarse as possible, precise when needed

SMC: Cartesian predicate abstraction

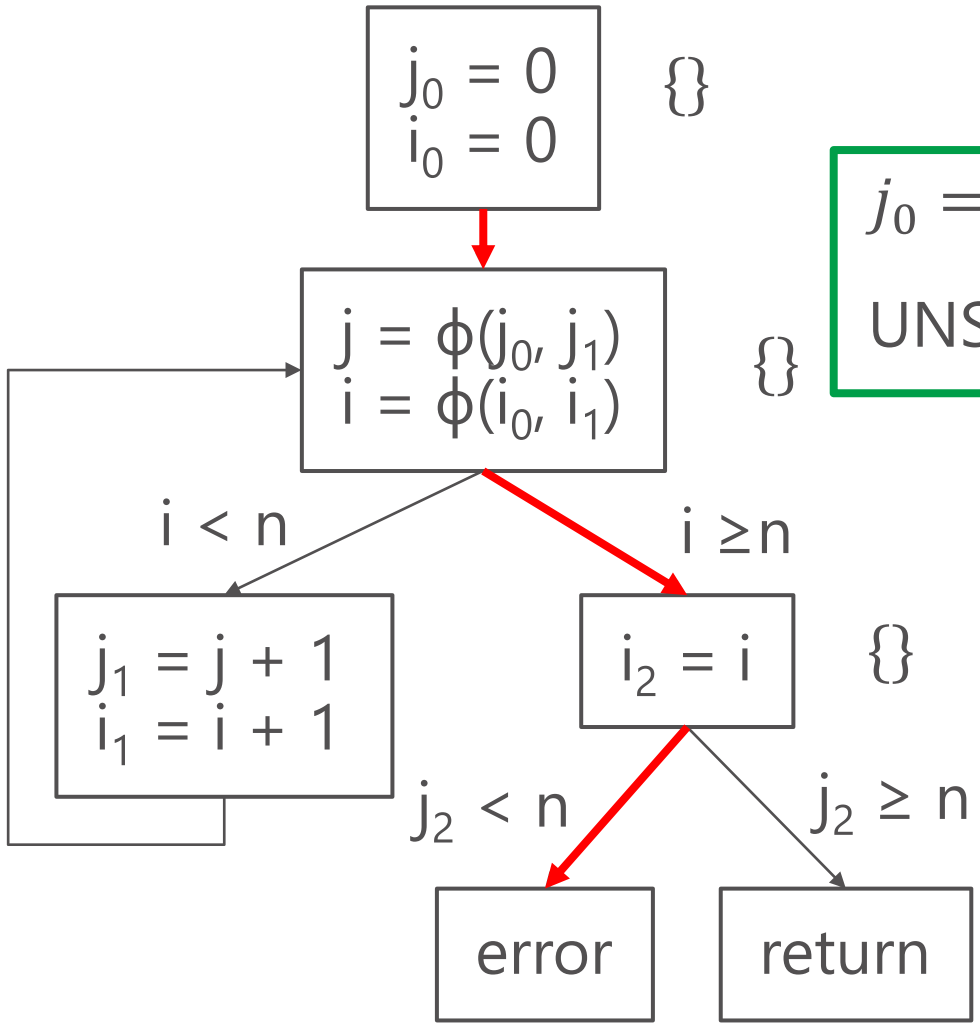
- Set of predicates, e.g.,
  $$P = \{x \geq 0, x + y < 2, x = 3\}$$
- $\alpha(x = 0 \land y < 1) = \{0,1\}$
- Entailment can be approximated syntactically:
  - If $a \subseteq b$ then $b \rightarrow a$
  - E.g. $\{1\} \subseteq \{1,3\}$ so $x \geq 0 \land\ x + y < 2 \rightarrow x \geq 0$

SMC: Example

```
void f(int n) {
  int j = 0;
  for (int i = 0; i < n; ++i) {
    j += 1;
  }
  assert(j >= n);
}
```



SMC: Example

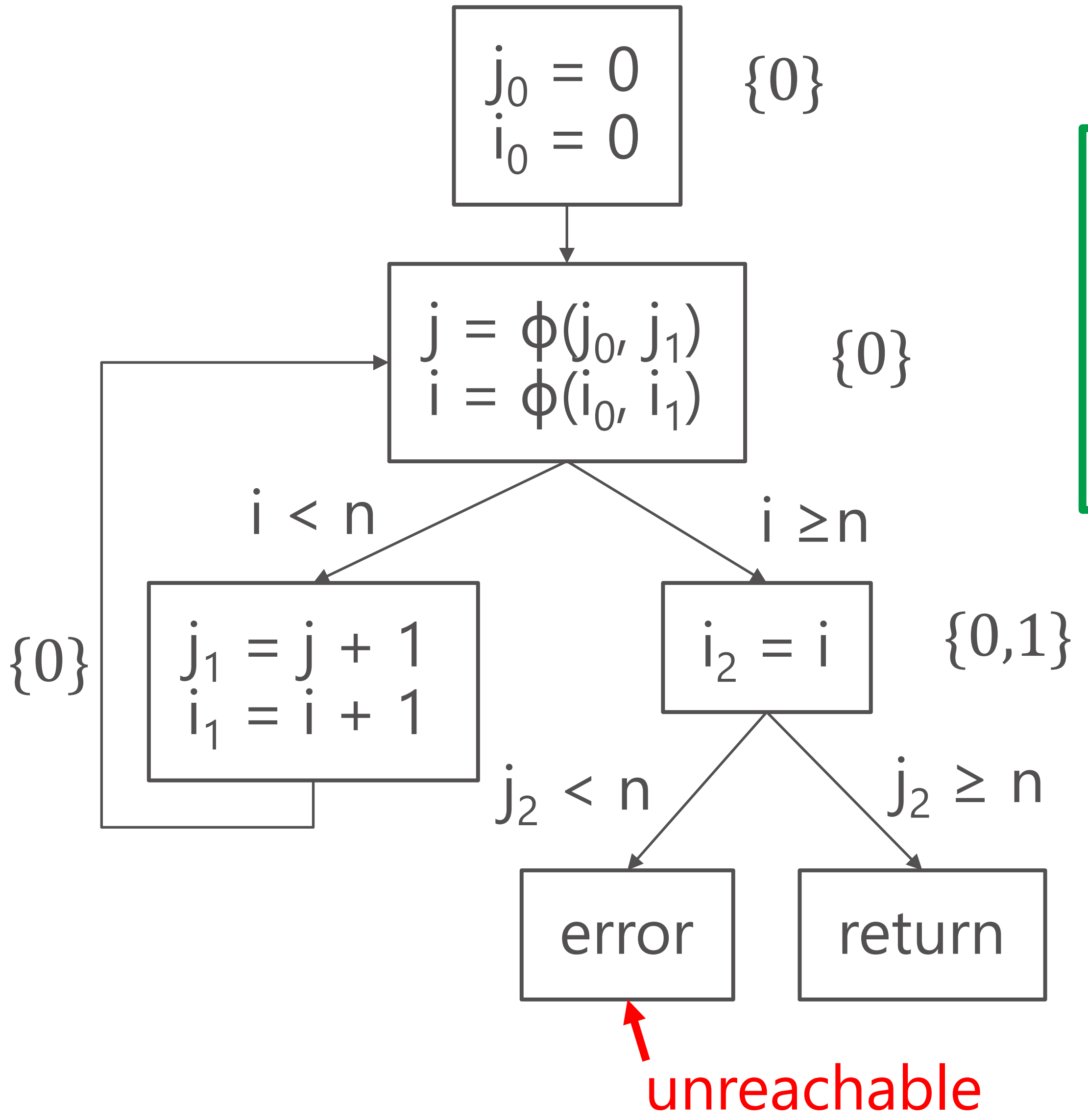


$P = \{\}$

$j_0 = 0 \wedge i_0 = 0 \wedge i_0 \geq n \wedge j_0 < n$

UNSAT = abstraction too coarse

SMC: Example Refinement



$$P = \{i = j, i \geq n\}$$

$$j = 0 \land i = 0 \rightarrow p_0(i,j)$$
$$p_0(i,j) \rightarrow p_1(i,j)$$
$$p_1(i,j) \land i \geq n \rightarrow p_2(i,j,n)$$
$$p_2(i,j,n) \land j < n \rightarrow \perp$$

$$p_0(i,j) \coloneqq i = j$$
$$p_1(i,j) \coloneqq i = j$$
$$p_2(i,j,n) \coloneqq i \geq n$$

SMC: Refinement

- Many algorithms: interpolants from SMT proof of UNSAT, algebra equations (Farkas lemma), polynomial rings, etc
- Predicate list: global, per “line”, per function, etc

SMC: Implementation

- Either discard all state across iterations, maintain partial state or whole sub-trees
- BFS vs DFS: no clear choice
- 1 query per predicate per “line” per iteration

Comparison

	Abstraction	**# SMT queries**	**Query result**	**Incremental?**
Abstract Interp	Fixed	1 per "line" per iteration	SAT	No
Symbolic Exec	No	1 per path	mixed	Yes
BMC	No	1 per unfolding	UNSAT	No
Model Checking	Adaptive	1 per "line" per predicate per iteration	UNSAT	No

Though clients

- Variety of theories: integers, rationals, reals, bit-vectors, floats
- Plus arrays and UFs
- Plus: incremental, copy SMT state, small queries and huge queries
- Resource bounds (time, memory)
- Sometimes w/ quantifiers

Preprocessing is awesome

	QARMC	QARMC -nopreprocess
cdaudio.bug	7s	70s (10x)
diskperf	88s	> 600s (> 7x)
summ_mccarthy	0.5s	> 50s (> 100x)
qrdcmp	1s	> 100s (> 100x)
tridag	0.7s	1.4s (2x)

QARMC: preprocessing

- C frontend:
  - Alias analysis
  - Trim read/write variable sets per function
  - ….

- At horn clause level:
  - Simplify formulas
  - Bottom-up / top-down symbol reachability
  - Inline clauses
  - Equality propagation
  - Subsumed clause elimination
  - Acceleration (compute simple loops)
  - Inter-clause transformations, like: constant/equality propagation, …

What does this mean for SMT?

- Program analysis properties are often shallow
  - Most stuff is irrelevant

- Cheap preprocessing pays off
- Compilers have cheap constraint solvers

- SMT solvers are already awesome; Can we aim for even more awesomeness?

API Experiments: queries/sec

	Z3 BV	SICStus Prolog CLPQ	LLVM ConstantRange
True/False	3,657	264,550 (72x)	
$a \geq b$	2,886	14,682 (5x)	
$a \geq b \wedge b \geq c$	883	4,996 (6x)	
$f(a) \geq b$	1,748		
$f(a) \geq b \wedge f(b) \geq c$	1,091		
$[0,42] \cap [5,99]$			20 Million

5,000x potential

All latest versions, Intel x64

API Experiments: take away

- Small queries are too slow
- Everything must be lazy, and only used when needed

- With ~4k SMT queries/sec, can only visit 40 states/sec if predicate list size = 100
  - 2 to 4 orders of magnitude slower than specialized abstract domains

Drawing ideas from Compilers

- Efficient algorithms for program analysis
- Super cheap (and imprecise) constraint solvers
- Can they be used in SMT solvers specialized for program analysis?

WARNING

I’m no SMT expert.

What follows is a list of ideas that could potentially maybe work in SMT. Mostly not new; some tried and failed

But: compiler algorithms did wonders in software verification tools

Use at your own risk.

Alias analysis

```
int f(int *a, int k) {
  int b[5];
  for (unsigned i = 0; i < 5; ++i) {
    b[i] = a[i] + i;
  }
  return b[k];
}
```

Can’t alias (i.e., don’t overlap)

Alias analysis

```c
int f(int *a, int k) {
  int b[5];
  for (unsigned i = 0; i < 5; ++i) {
    b[i] = a[i] + i;
  }
  return b[k];
}
```

Straightforward encoding:

$$mem' = (store\ (+\ b\ 0)\ (+\ (select\ (+\ a\ 0)\ mem)\ 0)\ mem)$$
$$mem'' = (store\ (+\ b\ 1)\ (+\ (select\ (+\ a\ 1)\ mem')\ 1)\ mem')$$

...

Equivalent to:

$$mem' = (store\ (+\ b\ 0)\ (+\ (select\ (+\ a\ 0)\ mem)\ 0)\ mem)$$
$$mem'' = (store\ (+\ b\ 1)\ (+\ (select\ (+\ a\ 1)\ mem)\ 1)\ mem')$$

...

Alias analysis: Expand 2 unfoldings

Straightforward encoding:

$$mem'' = (store\ (+\ b\ 1)\ (+\ (select\ (+\ a\ 1)$$
$$(store\ (+\ b\ 0)\ (+\ (select\ (+\ a\ 0)\ mem)\ 0)\ mem)\ 1)$$
$$(store\ (+\ b\ 0)\ (+\ (select\ (+\ a\ 0)\ mem)\ 0)\ mem))$$

Equivalent to:

$$mem'' = (store\ (+\ b\ 1)\ (+\ (select\ (+\ a\ 1)\ mem)\ 1)$$
$$(store\ (+\ b\ 0)\ (+\ (select\ (+\ a\ 0)\ mem)\ 0)\ mem))$$

Alias Analysis in SMT

- Propagate aliasing constraints **cheaply** and simplify array formulas

- E.g.: $(a + sizeof(a) \geq b \vee$
  $b + sizeof(b) \geq a) \wedge$
  $(= (select\ \ldots) \ldots)$

Array Theory in Z3

- Translation Validation for MSVC++
- Array theory vs encoding with (ite ..) and UFs
- Performance comparable with light alias analysis in vcgen
- Arrays slower if no analysis

Data-flow analysis

- Abstract interpretation
- Bottom-up vs top-down
- Varied domains: reachable definitions, live variables, signs, known bits, etc…
- Path sensitive vs path insensitive

DFA: Known bits

```
char f(char x) {
  char y = 0;                   y = 00000000
  if (x < 0)
    y = 3;                      y = 00000011
                                y = 000000xx
  if (x >= 3 && (y & 1) != 0)     ?
    return 0;
  return 1;
}
```

DFA: Known bits (path-sensitive)

```c
char f(char x) {
  char y = 0;
  if (x < 0)
    y = 3;

  if (x >= 3 && (y & 1) != 0)
    return 0;
  return 1;
}
```

y = 00000000

y = 00000011 / x = 1xxxxxxx

false! / false!

Data-flow: propagate bounds

- Context-sensitive propagation (left-right and right-left): $\boldsymbol{a \wedge b \wedge c}$
- Wrapping interval domain [l,h]
  - $l \leq h$: $[l, \mathrm{h}]$
  - $l > h$: $[l, max] \cup [0, h]$

$$\begin{tikzcd} x \geq 2 \wedge \cdots \wedge ite\,(x < 0, a, b) = 0 \arrow[d] \\ x \geq 2 \wedge \cdots \wedge b = 0 \end{tikzcd}$$

Propagate bounds: Example

- Example: Proof of correctness of automatic vectorization by MSVC++:
  - Vanilla Z3: > 300s
  - Z3+bounds: 0.05s

- However: not always profitable (too slow)
- What if “path-insensitive” (less precise)?

Tracing JIT / Profile-guided optimization (PGO)

JavaScript

```
function fn(a, b) {
  return a + b;
}
```

- The compiler may not know in advance the type of a and b
- Solution? Measure

Tracing JIT / Profile-guided optimization (PGO)

JavaScript

```
function fn(a, b) {
  return a + b;
}
```

$\downarrow$

```
function fn(a, b) {
  if (typeof a === 'number' && typeof b === 'number')
    return Number.add(a, b);
  if (typeof a === 'string' && typeof b === 'string')
    return String.concat(a, b);
  return a + b;
}
```

Top 2 types
at runtime

Profile-guided optimization (PGO)

- If can’t be predicted statically, **measure** at runtime and **adapt**

- SMT: can we specialize formulas where conflicts are more frequent?
- Can we specialize formulas for “simplifying” values? (e.g., zero)

Bit-blasting

- Brute-force approach
- Assume no overflow and use reals/integers (or case-split formulas)?
- Fallback only when needed (and only for the sub-formulas that need it)

I WANT YOU
TO DO AN
INTERNSHIP
memegenerator.net

Conclusion

- SMT solvers are awesome: they allowed complex program analysis to develop
- Know your client: adapt algorithms and APIs
- SMT solvers ~~can~~ should learn from program analysis

Microsoft Research

Microsoft