Scalable Defect Detection

Manuvir Das, Zhe Yang, Daniel Wang
Center for Software Excellence
Microsoft Corporation

Part II
High-Quality Scalable Checking
using modular path-sensitive analysis

Zhe Yang
Center for Software Excellence
Microsoft

Secret Sauce for a Practical Checker

Keys to high-quality scalable checkers
- Scalability: *checking each function in isolation*
- Quality: *path sensitivity and defect prioritization*

Approach proven by our experience at Microsoft
- espX: buffer-overrun checker, widely deployed and used to get 20,000+ bugs found and fixed
- µSpaCE: checker-building SDK, used by non-experts to build domain-rule-enforcing checkers

Scalability and Quality Overview

Scalability: Inter-Procedural Analysis?

Lecture 1 (by Manuvir): scalable inter-procedural analysis is possible, with
- Good Techniques: summarization, etc.
- Constraints on problems: finite automata, etc.

But
- Intractable for complex states (buffer overrun).
- Mismatch with the modular reasoning by devs.
  - "If an error is detected, who to blame"

Linear Scalability by Modular Analysis

If we can afford to analyze each function in isolation
- Scales up linearly in # of functions and scales out
- Allows using complex states for accuracy

But it's a big "if".
- For example, is this function safe?
  void f(int **buf*, size_t *n*)
  { for (size_t *i*=0; *i* <= *n*; *i* ++) *buf[i]* = 0; }
- Modular analysis requires specifications of the usage context (e.g., "*buf* has *n* elements").

Assumption: specification possible

- "Did you say specifications?"
  - Isn't it a pipe dream to design practical spec langs?
  - Who is going to add specs to millions of functions?
- This is the subject of Lecture 3 on SAL (by Dan)
- For now, assume functions come equipped with necessary specifications of contexts.
  - Say "void f(int *buf*, size_t *n*)" →
    "void f(int<n> *buf*, size_t n)"
- So we can discuss modular checking in full detail.

Quality: The measures

- *Accurate*: fix rate (% bugs fixed), false positive rate (% of reported bugs deemed noise)
  - Dev's perspective: frustrated with bogus issues.
- *Comprehensive*: validation rate (% of safe code), false negative rate (% of missed issues)
  - Exec's perspective: measure of coverage/progress.
- *Clear* and *Actionable*: easy to understand the reported defects and take appropriate actions

Quality Measures: Historical Perspective at Microsoft

- Early years
  - Bugs found by static analysis met with excitement
  - Accuracy is the obvious tool quality for devs
- After a few years of worm-induced news
  - "how many bugs are left?"
  - Measure of coverage calls for comprehensive validation
- Use of symbolic abstraction improved coverage
  - "I can't understand what this message is saying."
  - "There are so many issues and so little time left."
  - Messages need to be clear and prioritized

Achieving Quality

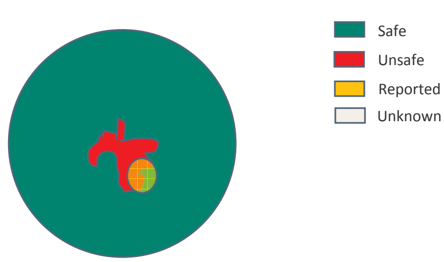
Clarity for developers to take action

- Using path-sensitive analysis instead of data flow analysis (since devs reason with paths)

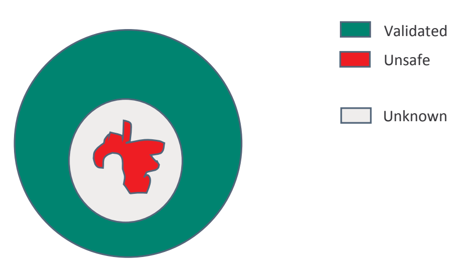
Conflicting Goals: (Accurate) defect detection vs (comprehensive) validation?

- Both: use comprehensive validation as a basis, and then expose defects through prioritization

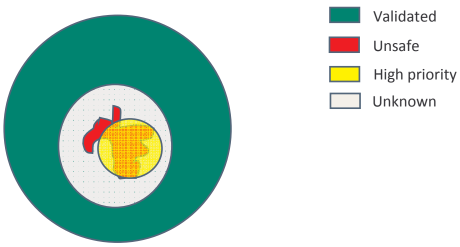
Detection



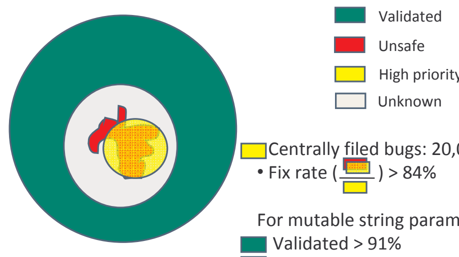
Validation



Comprehensive checking



Comprehensive checking: espX buffer overrun numbers on Vista



Centrally filed bugs: 20,000+
- Fix rate ( ) > 84%

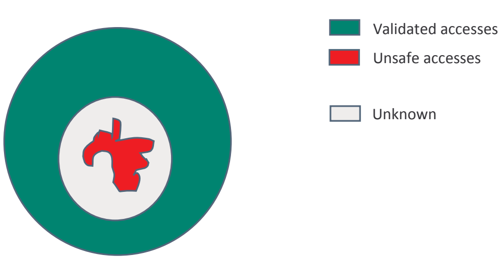
For mutable string parameters:
Validated > 91%
"How many bugs are left?" < 9%

espX: Buffer Overrun Checker

Outline of this section of talk

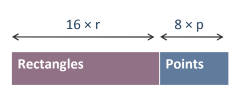
- Basics of a buffer-overrun checker
- Prior Art: merge-based dataflow analysis
- Our Approach: path-sensitive analysis
- Warning bucketing for prioritization

Basics: Validation of Buffer Accesses



Example 1

```
BYTE<n> *Alloc(size_t n);
void FillRects(RECT<n> *r, size_t n);
void FillPoints(POINT<n> *p, size_t n);

void Fill(unsigned int r, unsigned int p)
{
    BYTE *buf = Alloc(r * sizeof(RECT) + p * sizeof(POINT));
    FillRects((RECT *)buf, r);
    buf += r * sizeof(RECT);
    FillPoints((POINT *)buf, p));
}
```



“Instrumenting” the Program

```
BYTE *buf = Alloc(r * sizeof(RECT) +
                  p * sizeof(POINT));
assume: offset(buf) = 0 
        bcap(buf) = 16 × r + 8 × p
assert: offset(buf) + 16 × r ≤ bcap(buf)
FillRects((RECT *)buf, r);

buf += r * sizeof(RECT);

assert: offset(buf) + 8 × p ≤ bcap(buf)
FillPoints((POINT *)buf, p);
```

```
BYTE<n> *Alloc(size_t n);

FillRects(RECT<n> *r, size_t n);

FillPoints(POINT<n> *p, size_t n);
```


Analysis of Example 1

```
BYTE *buf = Alloc(r * sizeof(RECT) +
                  p * sizeof(POINT));
assume: offset(buf) = 0 
        bcap(buf) = 16 × r + 8 × p
assert: offset(buf) + 16 × r ≤ bcap(buf)
FillRects((RECT *)buf, r);

buf += r * sizeof(RECT);

assert: offset(buf) + 8 × p ≤ bcap(buf)
FillPoints((POINT *)buf, p);
```

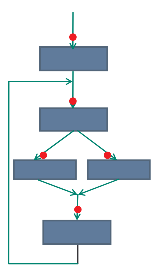
{r ≥ 0; p ≥ 0}

{**bcap(buf) = 16 × r + 8 × p; offset(buf) = 0;** r ≥ 0; p ≥ 0}
[offset(buf) + 16 × r = 16 × r ≤ bcap(buf)] PASS

{bcap(buf) = 16 × r + 8 × p; **offset(buf) = 16 × r**; r ≥ 0; p ≥ 0}
[offset(buf) + 8 × p = 16 × r + 8 × p ≤ bcap(buf)] PASS

Need **symbolic state tracking** +
linear integer theorem prover

Dataflow Analysis



Given flow graph (V, E)

Task: find invariants at CFG-nodes

Find a map $A: V \rightarrow Abs$
stable under $T: E \times Abs \rightarrow Abs$

Abs: lattice of abstract values

Stability condition:
$A(v) = \bigsqcup \{ T(e, A(u)) : e = (u, v) \in E \}$
or $A = \check{T}(A)$ is a fixed-point of $\check{T}$

If *T* monotone, *Abs* complete, then least $A = \bigsqcup \{\check{T}^i(\perp) : i = 0, \ldots\}$.

This terminates if Abs finite in height.

Work-list algorithm used in practice.

Dataflow Analysis for Buffer Overruns [Dor et al:PLDI 2003]

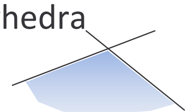
To track the symbolic states, use a dataflow analysis

- Abs = Set of Linear Inequality Constraints
- T : suitably abstracted from concrete semantics

But what about the join operator?

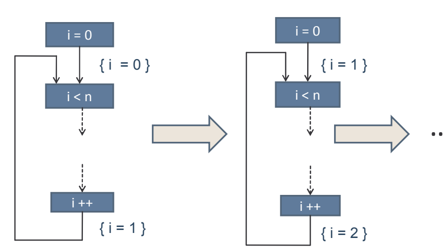
- I.e., how do you merge two sets of linear constraints into another set of linear constraints (that is implied by both of them)?
- Answer: Polyhedra (Cousot/Halbwachs:POPL78)

The Lattice of Polyhedra

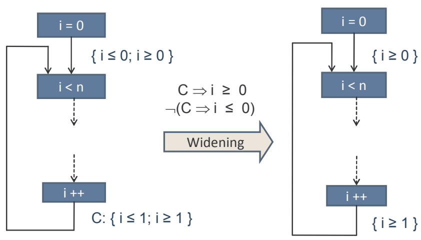


- Geometric Interpretation:
  - One linear inequality gives a half-space
  - A set of linear inequalities is a (maybe-not-closed) convex polyhedron (n-dimensional polygon)
- Join-operator needs to find the smallest enclosing polyhedron (convex hull problem)
- Algorithm involves lots of linear programming
- infinite-height lattice: termination for loops?

What about Loops ?



Loop Widening



Handling loops

- Use a loop widening algorithm to ensure that the analysis terminates
  - Widening operator: Weaken the constraints along a back edge of a loop in a way that ensures that finite number of such weakenings is sufficient
  - Mathematically, any chain formed by repeated application of the widening operator is finite.

Issues with Polyhedra

- Complexity (implementation & cost)
- Several restricted version proposed and used:
  - Octagons (at most two variables; coefficients 1, -1)
  - Arbitrary predetermined shapes.
- Inaccuracy
  - The convex hull won't be accurate closure
  - Real-numbered coefficients would appear
  - An approximation for Integer Linear Constraints

Bigger issue: feedback to developer

Example 2

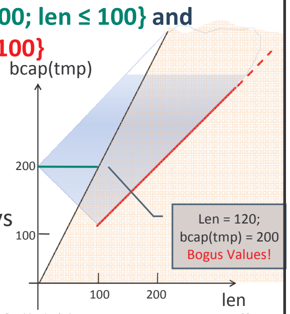
```
BYTE<size> *Alloc(size_t size);
void  StringCopy(wchar_t<n>*dest, const wchar_t {null-terminated} *src, size_t n);

void ProcessString(wchar_t *str)
{
    wchar_t buf[100];
    wchar_t *tmp = &buf;

    int len = wcslen(str) + 1;
    if (len > 100)
        tmp = (wchar_t *)Alloc(len);     Should be Alloc(len * sizeof(wchar_t))

    StringCopy(tmp, str, len);           Buffer overrun
    ...
}
```

{bcap(tmp) = 200; len ≤ 100} vs {bcap (tmp) = len; len > 100}

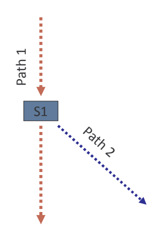
Example 2 with Polyhedra

- Merging **{bcap(tmp) = 200; len ≤ 100} and {bcap(tmp) = len; len > 100}**
- That is:
  bcap(tmp) ≥ len
  bcap(tmp)+len ≥ 200
  bcap(tmp) ≥ len+200
- Obscure message to devs

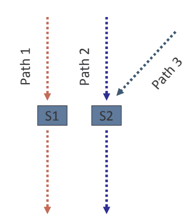
Need path-based analysis:
"overflow when len ≥ 100"



Sharing in Path Sensitive Analysis



Paths with common prefix share analysis



Paths which reach the same program point with the same symbolic state share a suffix

Path-Sensitive Dataflow Analysis

- In its simplest form, path-sensitive analysis can be characterized as a dataflow analysis
- Find *A: V → Set(state)* using *t: E x state → state*
- *T: Set(state) → Set(state)* is the point-wise lifted version of *t*, using set-union.
- *Set(state)* is a complete lattice; *T* is monotone
- But *Set(state)* is infinite in height when the universe of states is infinite.

Widening in Path-Sensitive Analysis

- Issue: We share paths only when the symbolic states are the same at a node; but loops induce infinite number of states.
- "Widening? But what state to widen against?"
- Idea: at back edge, widen against the path itself
- Solution: extend the state to record the path-history of states at each loop entry node.
- *WidenedState = LoopNestingLevel → State*
  [s_1, s_2, s_3]: state is s_3 now, and was s_i at loop level i
- Exercise: work out the detail

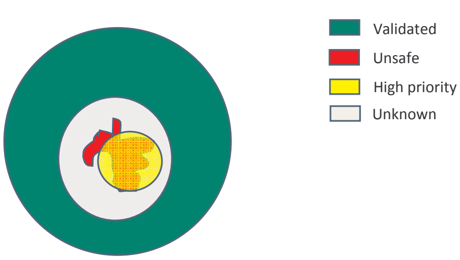
Fast Theorem Prover for Integer Linear Inequalities?

- Not asking for: constraint solver, completeness
- Observation 1: developers reasoning about linear constraints in a simple way; often: a proof is just a linear combination with small integer coefficients.
- Observation 2: difference constraint theorem prover is easy to construct. [CLR:alg-textbook]
- Exercise: figure out an algorithm.

Elements of the checker

- Symbolic state tracking with linear inequalities
  - Path sensitive analysis
  - Path sensitive loop widening
- Fast linear integer theorem prover

Defect Bucketing/Prioritization



Example 1 – provable error

```
if (CanProve(buffer index < buffer size))
    Validated Access
else
    if (CanProve(buffer index >= buffer size))
        Provable Error
    else
        Possible Error
```

- e.g. passing byte count instead of element count

```
wcscpy_s(buf,sizeof(buf),s); espX Warning 26000
```

Example 2 – incorrect validation

```
int glob[BUF_SIZE];
bool read(int i, int *val) {
      if (i > BUF_SIZE)   // Off by one
       return false;
      assert: i < BUF_SIZE
      *val = glob[i];
    …

espX Warning 26014:
Cannot prove: i < BUF_SIZE
Can prove: i < BUF_SIZE + 1
```

e.g. MS01-033(Code Red), MS04-036(NNTP), MS04-035(SMTP)

Example 3 – missing validation

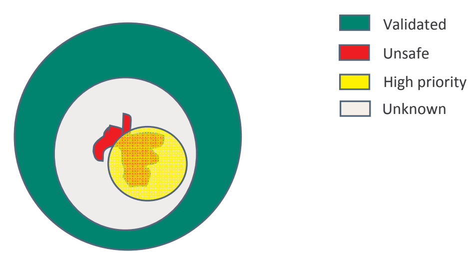
```
void Transform(char<size> *dest,
               const char{null-terminated} *src,
               size_t size) {
    assert: strlen(src) + 1 <= size
    memcpy(dest, src, strlen(src) + 1);
}

espX Warning 26015:
Constraint set does not relate size and strlen(src)
```

e.g. MS03-026(Blaster), MS05-039 (Zotob)

Warning bucketing criteria

- Are heuristics based on observations of common coding mistakes
- Are semantic, not syntactic, in nature
  - Makes them robust
- Validated by security bulletin bug data and Watson crash data

Precision Improvements



Loop Invariant Inference

```
void StripSpaces(char<n> *dest, char *src,   size_t n)
{
    while (*src != 0 && n > 1)  {
       assume: n > 1
       if (*src != ‘ ’)  {
              assert: offset(dest) < n0
              *dest++ = *src;
              n--;
       }
       src++;
    }
    *dest = ‘\0’;
}
```

Need loop invariant offset(dest) + n = n_0

espX deduces offset(dest) and n are *synchronized variables* in the loop

Combining Theorem Provers

- Example: uuencode into 6-bit ASCIIs

10111010 | 01100100 | 00111001 | 10010000 | 11110101

⇩

0010111 | 00100110 | 0001000 | 00111001 | 0010010 | 00001111 | 0001010

void uuencode(BYTE<n> *src,
BYTE<(n+2)/3*4> *dest, size_t n)
(Real spec added by a developer to real code)

- A second layer of theorem prover to uninterpreted operations, integer divisions, modular arithmetic, bitwise operations & etc.

espX Summary

- espX have made comprehensive defect detection a reality for buffers
  - Tens of thousands of bugs found and fixed
  - “How many bugs are left ?”
    < 9% for mutable string buffers in Vista
  - Specifications also important (Details in Lecture 3)
- Achieved using
  - Modular Path-sensitive analysis
  - Careful warning bucketing and prioritization
  - Assortment of precision-refinement techniques

Devs want to build good checkers, too

- Developers who are domain experts often want to enforce certain domain-specific rules
- Encouraged by our work, they want to go static
- E.g.: Project Goldmine (Internationalization)

```
void IssueMessage()
{
  ::MessageBox(NULL,
       L"Failed to load file",    <------ Hard-coded strings passed to user facing API
       MB_ERROR |MB_OK);
}
```


Developer-Generated Analyses: µSpaCE

(Or Better: You Checker)

How Do We Share Our Expertise?

- We understand static analysis well, but we can’t solve problems for all domains
- MS solution: an SDK for domain experts to build path-sensitive dataflow analysis
- Challenge: intelligible explanation, i.e., without lattice, monotone function, join, etc.
- Our explanation: based on “Path Iteration”

Path Iteration

- Get set of paths; traverse them separately
- Simulation-style code:

```
For all paths p.
{
   …
   For all edges e in p.
      …
}
```

```
int f(int i, int n)
{
  if (i)
    i = 1;
  else
    i = 2;
  i=3;
  while (i < n)
    i = 4;
  return i;
}
```

- Limitation:
  - Cannot have full coverage
  - No sharing of analysis across paths

Path Iteration

- Get set of paths; traverse them separately
- Explicit simulation state:

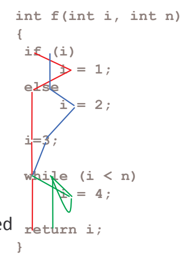
```
For all paths p.
{
   Φ = Φ_initial;
   For all edges e in p.
      Φ = t(Φ, e);      <-- "transfer function"
}
```

```
int f(int i, int n)
{
  if (i)
    i = 1;
  else
    i = 2;
  i=3;
  while (i < n)
    i = 4;
  return i;
}
```

- Benefit of abstraction
  - Under-the-hood improvements

Path-Sensitive Analysis

- Define transfer function with explicit abstract state
- The µSpaCE engine maintains
  - A state set pr. node
  - Reuses path computations
  - Covers state space 100%
- The fine print:
  - State domain needs to be finite
  - Or else widening operator needed



µSpaCE SDK for Building Checks

- SDK: a concise core (with virtual transfer functions)
  + oracles (memory model, spec semantics, etc.)
- Multiple clients in one year
  - Goldmine (C/C++/.NET): intl. checker & meta data gen.
  - espC (C/C++): concurrency checker
  - iCatcher (.NET): cross-site scripting checker for ASP.NET
  - NullPtr (C/C++/.NET): spec-based null-ptr checker
- All these clients have found real bugs;
  they are getting deployed company wide

Summary

Keys to high-quality scalable checkers

- Scalability: *checking each function in isolation*
- Quality: *path sensitivity and defect prioritization*

Approach proven by our experience at Microsoft

- espX: buffer-overrun checker, widely deployed and used to get 20,000+ bugs found and fixed
- µSpaCE: checker-building SDK, used by non-experts to build domain-rule-enforcing checkers

Exercises & Recommended Readings

for "High-Quality Scalable Checking using Modular Path-Sensitive Analysis"

Exercises:

1. **[Path-Sensitive Loop Widening]** Work out the mathematical detail of path-sensitive loop widening. Prove that the algorithm terminates.
2. **[Theorem Proving]** Using the observation that linear combination with small integer coefficients would suffice most of the time for developers' reasoning, construct a theorem prover for linear inequality constraints based on graph search. A good starting point is the reduction of solving difference constraints to the shortest-path problem (Cormen, et al. 2001). How would you add the reasoning power for modular arithmetic?
3. **[Defect Presentation]** A path-sensitive analysis uses sharing to achieve coverage while ensuring termination and maintaining a reasonable cost. But when you want to show a potential issue to a developer, it is best to show them the symbolic information along code paths. For one defect found, there could be infinite number of paths leading to the defects, due to the presence of loops How would you present the set of infinite paths to the developers in an understandable manner?

If you feel like, you can send your solutions, comments, or questions to me (zhe.yang@microsoft.com).

Recommended Readings

On Buffer Access Checking

(Dor, Rodeh and Sagiv 2003): Assert-assume framework for buffer-access checking.

(Cousot and Halbwachs 1978): Original paper on the lattice of polyhedral and its application to find linear constraints in a program.

(Hackett, et al. 2006): Our work on buffer access checking. This article offers an overview of the workflow in use at Microsoft, the annotation language, the inference algorithm of the annotations, as well as a high-level sketch of the path-sensitive checker.

General Introduction to Program Analysis and Domain Theory

(Schwartzbach n.d.): An efficient introduction to the techniques of program analysis.

(Nielson, Nielson and Hankin 1999): Standard textbook on program analysis.

(Winskel 1993): Chapter 8 offers a good introduction to domain theory

Path-Sensitive Dataflow Analysis

(Reps, Horwitz and Sagiv 1995): Seminar work on scalable inter-procedural analysis. The simplest form of path sensitive analysis can be understood as graph reachability.

•••

Microsoft®

http://www.microsoft.com/cse
http://research.microsoft.com/users/zhey
zhe.yang@microsoft.com