Scalable Defect Detection

Manuvin 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-overflow 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 Manuvin): 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

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

espX: Buffer Overrun Checker

Basics: Validation of Buffer Accesses

Example 1

```
BYTE *Buf = Alloc(size_t n);
void FillRects(RECT *r, size_t n);
void FillPoints(POINT *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(size(RECT) + p * size(POINT));
assume: offset(buf) = 0;
bcap(buf) = 16 * r * 8 * p;
assert: offset(buf) + 16 * r ≤ bcap(buf);
fillRects(RECT *buf, r);
buf = r * size(RECT);
assert: offset(buf) + 8 * p ≤ bcap(buf);
fillPoints(POINT *buf, p);

Analysis of Example 1

BYTE *buf = Alloc(size(RECT) + p * size(POINT));
assume: offset(buf) = 0;
bcap(buf) = 16 * r * 8 * p;
assert: offset(buf) + 16 * r ≤ bcap(buf);
fillRects(RECT *buf, r);
buf = r * size(RECT);
assert: offset(buf) + 8 * p ≤ bcap(buf);
fillPoints(POINT *buf, p);

Need symbolic state tracking + linear integer theorem prover

Dataflow Analysis

Task: find invariants at CFG-nodes

Find a map A: V → Abs
stable under T: E x Abs → Abs
Abs: lattice of abstract values
Stability condition:
A(v) = ∪ {T(e, A(u)): e = (u, v) ∈ E} or A = T(A) is a fixed point of T
If T monotone, Abs complete, then
least A = ∪ {T(ϕ); i = ϕ, ...}.
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
(for 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?
### 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 *s = *Allocate(size);  
    wchar_t*str = *Allocate(*size);  
    int len = *Allocate(*size) + 1;  
    wchar_t *tmp = *Allocate(*size) + 1;  
    if (len > 100)  
        *StringCopy(tmp, str, len);  
    else  
        *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:
  - \( bcap(tmp) \geq len \)
  - \( bcap(tmp)+len \geq 200 \)
  - \( bcap(tmp) \geq len+200 \)
- 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 \( A : V \rightarrow Set(state) \) using \( t : E \times state \rightarrow state \)
- \( T : Set(state) \rightarrow Set(state) \) 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.
- \( WidenedState = LoopNestingLevel \rightarrow State \)
- \([S_1, S_2, S_3]\): state is \( S_3 \) now, and was \( S_1 \) at loop level
- 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

Example 1 – provable error

```c
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
  - `wcsncpy_s(buf, sizeof(buf), s); espX Warning 26000`
Example 2 – incorrect validation

```c
int glib[NBUF_SIZE];
bool read(int i, int *val) {
    if (i > NBUF_SIZE) // Off by one
        return false;
    assert: i < NBUF_SIZE
    *val = glib[i];
}
espX Warning 26014: Cannot prove: i < NBUF_SIZE
Can prove: i < NBUF_SIZE + 1
```

Example 3 – missing validation

```c
void Transform(char *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

- Validated
- Unsafe
- High priority
- Unknown

Loop Invariant Inference

```c
void Stripspaces(char *dest, char *src, size_t n) {
    while (*src != 0 && n > 1) {
        assume: n > 1
        if (*src != 0) {
            assert: offset(dest) < n
            *dest = *src;
            n--;
        }
        else {
            *dest = 0;
        }
    }
espX deduces offset(dest) and n are synchronized variables in the loop
```

Combining Theorem Provers

- Example: uuencode into 6-bit ASCIIs
  ```
  | 011110 | 0110000 | 111001 |
  | 110001 | 0001000 | 000101 |
  | 000111 | 0010100 |
  ```
  ```c
  void uuencode(u64 *dest, BYTE<n+2/34> *src, 
  BYTE<n+2/34> *dest, size_t n)
  ```
  (Real spec added by a developer to real code)
  - A second layer of theorem prove 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)

```cpp
void IssueMessage()
{
    // Hard-coded strings
    ::MessageBox(NULL,
                 L"Failed to load file",
                 MB_ERROR | MB_OK);
}
```

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:

```cpp
int f(int i, int n) {
    if (i > n) return 1;
    for all edges e in p.
    if (i + e = 2) return 1;
}
```

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

Path Iteration

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

```cpp
int f(int i, int n) {
    if (i > n) return 1;
    for all edges e in p.
    if (i + e = 2) return 1;
}
```

- Benefit of abstraction:
  - Under-the-hood improvements
  - “transact function”
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): int1. checker & meta data gen.
  - espC (C/C++): concurrency checker
  - iCatcher (.NET): cross-site scripting checker for ASP.NET
  - NullPr (C/C++/.NET): spec-based null-pr 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 Yielding Sensing
- Keys to high-quality scalable checkers
  
- Designing high-performing type inference algorithms
- The influence of type inference algorithms on performance
- μSpaCE: a compiler infrastructure for high-quality C/C++
- μSpaCE: a tool for analyzing C/C++ programs

- μSpaCE: a tool for analyzing C/C++ programs

http://www.microsoft.com/cce
http://research.microsoft.com/users/they
she-yang@microsoft.com

© 2007 Microsoft Corporation. All rights reserved.
This presentation is for informational purposes only.
MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.