A static analyzer for finding dynamic programming errors

William R. Bush*, Jonathan D. Pincus and David J. Sielaff

Table of contents

01

Introduction

What means?

It is a technique that identifies errors that occur during a program execution without having to really execute the code and instead analysing it during the compilation phase

How can we relate ?

Positive impact on the technological industry where you can have, mainly a reduction of costs and a real impact on the users that will have a much better and safer user-experience

02

Limitations

Traditional methods of error verification

Errors' verification

Compilers

- Limited scope
- Incomplete coverage
- Superficial assumptions

Specialized evaluators

False errors

Report non-existent errors and find errors in inaccessible code

Addition of the workload

Adds significant programmer effort

Both of these verification tools show various problems and challenges so are not ideal solutions for our final goal

PURIFY

Implies some limitation by being a debugger and thus needing test cases to verify errors

03
PREfix

Methodology

Concepts

Function execution simulation

Detects inconsistencies in execution paths using language consistency rules

Bottom-up approach

Identifies defects starting from leaf functions up to the root of the call graph

Simple results on PREfix usage

Function that, in short terms, does memory allocation following by inicializing it

```
example1.c(11) : warning 14 : leaking memory
    problem occurs in function 'f'
    The call stack when memory is allocated is:
        example1.c(9) : f
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size > 0' here
        example1.c(10) : when 'size == 1' here
    Path includes 4 statements on the following lines: 8 9 10 11
    example1.c(9) : used system model 'malloc' for function call:
        'malloc(size)'
    function returns a new memory block
        memory allocated
```

```
example1.c(12) : warning 10 : dereferencing uninitialized pointer 'result'
    problem occurs in function 'f'
    example1.c(6) : variable declared here
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size <= 0' here
    Path includes 3 statements on the following lines: 8 10 12
```

04.1

Analizer Structure and Implementation

Introduction

- **Source Code Analysis**: PREfix begins by analyzing the source code using a standardized C/C++ compiler front end, generating Abstract Syntax Trees (ASTs).

- **Function Simulation Order**: It determines the order of function simulation through a topological sorting based on caller-callee relationships.

- **Path Simulation**: PREfix simulates all possible paths through a function, evaluating each instruction and updating memory state. It identifies defects such as uninitialized memory and invalid pointer dereferences during simulation, while memory leaks are detected during end-of-path analysis.

- **Memory State Summarization**: After simulating each path, PREfix summarizes the final memory state. These summaries are then combined into a comprehensive model for the function.

Introduction

```
while (there are more paths to simulate)
    {
        initialize memory state
        simulate the path, identifying inconsistencies and updating the memory state
        perform end-of-path analysis using the final memory state,
            identifying inconsistencies and creating per-path summary
    }
    combine per-path summaries into a model for the function
```

Figure 5. Pseudo-code for function simulation.

Per-path simulation (Paths)

- Covers all possible execution scenarios within a function
- Each path through a function begins at the function entry point and continues through various statements until the function exits
- Evaluates statements and updates memory state

Per-path simulation (Paths)

```
1    #include <stdlib.h>
2    #include <stdio.h>
3
4    char *f(int size)
5    {
6         char *result;
7
8         if (size > 0)
9             result = (char *)malloc(size);
10        if (size == 1)
11            return NULL;
12        result[0] = 0;
13        return result;
14   }
```

```
example1.c(12) : warning 10 : dereferencing uninitialized pointer 'result'
    problem occurs in function 'f'
    example1.c(6) : variable declared here
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size <= 0' here
    Path includes 3 statements on the following lines: 8 10 12
```

Figure 3. Second error message.

Per-path simulation (Memory)

- The memory used by a function being simulated is tracked as accurately as possible.

- Each piece of memory can have an exact value, be initialized without a known exact value, or be uninitialized.

- Predicates are associated with these values to express various conditions and relationships.

- Unary predicates represent language semantics.
- Binary predicates cover relational operators.
- Ternary predicates describe more complex relations, such as a = b + c.

Per-path simulation (Memory)

•Operations on memory during simulation include:

- Evaluating expressions: Involves a recursive descent of expressions and returns a new memory location.
- Testing conditions: Evaluates conditions with a three-valued logic (TRUE, FALSE, DON'T KNOW).
- Setting values: Assigns values to memory locations. For example, if a > 3 and b > 4, the simulator can infer that a + b > 7.

•These operations ensure that the memory state reflects the current execution context accurately

Per-path simulation (Conditions, assumptions and choice points)

•**Conditions**: Correspond to specific states or values that trigger warnings if violated

•**Assumptions**: Made when variable values are unknown, guiding the simulation along plausible paths. (Ex : size)

•**Choice points** occur when the simulation encounters multiple possible paths

•The simulator chooses one path and explores the others subsequently

Per-path simulation (End-of-path)

•At the end of each path, PREfix summarizes the final memory state.

•It performs additional analysis to detect errors such as memory leaks or invalid pointer dereferences.

•This end-of-path analysis is crucial for identifying defects that may not be apparent during the simulation of individual statements.

•By examining the final state of memory, PREfix ensures that all potential inconsistencies are captured and reported.

•These summaries are then combined into a comprehensive model for the function.

Multiple Paths

- Selects a representative sample of paths for practical analysis.
- User-configurable maximum number of paths
- Identifies defects in multiple execution scenarios efficiently.
- Optimized random selection maximizes coverage.
- Detects errors manifesting under specific conditions.

Models

Definition: represents an abstraction of a function's behavior during software execution.

Let’s break it down:

- Structure and Functionality
- Concepts: Guard, Constraints, and Results
- Model Emulation
- Handling Dynamic Memory and Resources
- Model Generation

Models: Structure and Functionality

- **Function Models:** Each function in the program is represented by a model that describes its behavior
- **Outcomes:** Each model consists of different outcomes, which are potential results of the function execution, dependent on input conditions
- **Externals:** Models include details about external interactions of the function
- **Operations:** Outcomes in a model are defined by operations

Models: Guard, Constraints, and Results

- **Guards:** conditions that determine whether a specific outcome of a model is applicable based on the input values to the function

- **Constraints:** preconditions that must be satisfied for the model to consider its execution valid

- **Results:** postconditions or effects of executing the function, which change the state of the system or output specific values.

Models: Example

Let’s break it down...

```
int deref(int *p)
{
        if (p==NULL)
                return NULL;
        return *p;
}
```

Function

```
(deref
        (param p)
    (alternate return_0
        (guard peq p NULL)
        (constraint memory_initialized p)
        (result peq return NULL)
        )
    (alternate return_X
        (guard pne p NULL)
        (constraint memory_initialized p)
        (constraint memory_valid_pointer p)
        (constraint memory_initialized *p)
        (result peq return *p)
        )
    )
```

Model

Models: Example

With this, we can conclude

Outcome 1:

- **guard:** p equals NULL;
- **constraint:** p must be initialized;
- **result:** the return value is NULL.

Outcome 2:

- **guard:** p does not equal NULL;
- **constraint:** p must be initialized;
- **constraint:** p must be a valid pointer;
- **constraint:** the memory pointed to by p must be initialized;
- **result:** the return value is equal to the memory pointed to by p.

Model Emulation

Definition: simulating the behavior of functions based on predefined models during software analysis

- **Execution Path:** When a function is called, the model corresponding to that function is retrieved and used to simulate the function call

- **Evaluation of Guards and Constraints:** Guards are evaluated to filter applicable outcomes, and constraints are checked to ensure they are met before proceeding

- **Result Application:** Once an applicable outcome is determined and constraints are verified, the results are applied to simulate the function's effect on the system

Handling Dynamic Memory and Resources

- Models include operations for dynamic memory allocation and deallocation
- The model ensures that memory and resource management operations adhere to correct programming practices

Model Generation

- Automatic Model Generation
- Merging States

Incomplete knowledge and conservative assumptions

Sometimes PREfix…

There are some implications:

- **Pointer Usage:** all pointers are used, implying that pointer aliasing occurs
- **Return Values:** return value of a function can be anything, which allows for correct handling of the return value in any subsequent context
- **Memory Modifications:** any function can modify any memory to which it has access

04.2

DEVELOPMENT AND USE OF THE ANALYZER

Development Process

- The development of PREfix followed an iterative improvement process.
- Tested on over 100 million lines of code from various styles and real-world scenarios.
- Extensive testing provided continuous feedback from commercial sites.
- Feedback was crucial for refining the tool.
- Focused on reducing false positives and improving performance.

Use Model

- Inspired by Purify: Modeled after the user-friendly debugging tool Purify.
- Test Case Free
- Seamless Integration: Designed to seamlessly integrate into existing build environments.
- Effortless Error Detection: Effortlessly detects errors in extensive source bases.
- Real-world Deployment: Successfully deployed at commercial sites, enhancing code quality.

Practical Application

- Focus on large commercial programs
- Parameters for customization (path exploration limits and analysis time per function)
- Examples: Mozilla and Apache

05

RESULTS

Metrics

Some metrics were evaluated:

- Performance Metrics
- Detection Capability (Warning Categories / Statistics)
- Comparative Analysis

Results: Performance Metrics

Notes:

- Time to parse is 2-5x the build time
- Ratios of 2–4 are typical for C code
- 3–5 for C++ code

Table I. Performance on sample public domain software.

Program	Language	Number of files	Number of lines	PREfix parse time	PREfix simulation time
Mozilla	C++	603	540 613	2 h 28 min	8 h 27 min
Apache	C	69	48 393	6 min	9 min
GDI Demo	C	9	2655	1 s	15 s

Results: Detection Capability – Warnings

Warning	Mozilla	Apache	GDI
Using uninitialized memory	26.14%	45%	69%
Dereferencing uninitialized pointer	1.73%	0	0
Dereferencing NULL pointer	58.93%	50%	15%
Dereferencing invalid pointer	0	5%	0
Dereferencing pointer to freed memory	1.98%	0	0
Leaking memory	9.75%	0	0
Leaking a resource (such as a file)	0.09%	0	8%
Returning pointer to local stack variable	0.52%	0	0
Returning pointer to freed memory	0.09%	0	0
Resource in invalid state	0	0	8%
Illegal value passed to function	0.43%	0	0
Divide by zero	0.35%	0	0
Total number of warnings	1159	20	13

Results: Detection Capability - Warnings

Notes:

- Mozilla has much more warnings found by PREfix
- The number of warnings identified by PREfix ranges from 1 per 200 LOC (lines of code) to 1 per 2000 LOC
- PREfix occasionally incorrectly reports a warning. In these examples, the percentage of such messages ranges from under 10% (GDI demo, Apache) to roughly 25% (Mozilla).

Results: Comparative Analysis

Notes:

- No modules: no leaks can be detected because there are no memory allocations.
- only system models: the only leaks that are detected are situations involving direct calls to malloc and free

Model set	Execution time (minutes)	Statement coverage	Branch coverage	Predicate coverage	Total warning count	Using uninit memory	NULL pointer deref	Memory leak
None	12	90.1%	87.8%	83.9%	15	2	11	0
System	13	88.9%	86.3%	82.1%	25	6	12	7
System & auto	23	73.1%	73.1%	68.6%	248	110	24	124

06

Conclusions and Observations

Effectiveness of PREfix

- Highly effective at detecting dynamic errors
- Capable of analyzing complex, real-world codebases
- Valuable in commercial software environments

Key Observations

- Most errors involve interaction between multiple functions
- Errors frequently occur off the main code paths
- Older code tends to be more reliable due to extensive testing

Continuous Improvement and Adaptation

- Continuous refinement based on user feedback
- Adaptations to enhance usability and performance
- Integration with modern development environments

Future Directions

- Expanding detection capabilities (ex: race conditions and deadlocks in multi-threaded applications)
- Enhancements in result presentation and filtering tools
- Extending support to more programming languages

07

Appendix: The Modelling Language

Appendix

- Detailed syntax and operations
- Defines constraints, guards, and results
- LISP-like syntax for automatic generation
- Captures function behavior
- Enhances analysis accuracy
- Crucial for automatic model generation
- Refer to appendix for details
- Key part of PREfix’s functionality

Thanks!

Do you have any questions?

Software Testing and Validation

Henrique Vaz	98938
Rui Santos	98966
Daniel Costa	98930