

# Methods for Approximating Memory Safety in C

Greg Singer  
CSE 814  
Michigan State University

# From SPARK to... ?

- SPARK was created with formal methods in mind
- Can prove (partial) correctness, but only if:
  - You allocate all memory statically
  - You avoid recursion
  - You provide extensive annotations throughout the program
- What if you would prefer not to do that?
  - Is it possible to get SPARK-like guarantees in a less restrictive language?

# Memory Safety and C

- C gives the programmer direct control over memory
  - Can allocate or free memory at any time
  - Can access memory at programmer-defined addresses
  - Can cast data between types whenever necessary
- C is very flexible
- C is also very dangerous
  - Mistakes can lead to crashes and/or security vulnerabilities
- Is it possible to verify that a C program makes none of these mistakes?

# Memory Safety and C

- Short answer: “No”
- Longer answer: “Not really”
- Finding every error without also throwing some false positives is *undecidable*
  - You can, however, try to come close
- Conducted a survey of proposed analyses
  - Found four categories into which most can be grouped
  - Also tested a few briefly to see their actual performance

# Lexical Analyzers

- The simplest approach
  - Break up source code into tokens, and look for trouble
  - Example: `strcpy()`
    - Copies characters between buffers until `'\0'` is read
    - No check is done to avoid overflowing output buffer
  - Goal is to find a few common errors as fast as possible
- Examples: ITS4, Flawfinder
  - All use a database of “bad” tokens like `strcpy`

# Compiler-like Tools

- Try a bit harder
  - Parse the program, rather than tokenizing it
  - Look for potential errors in the program's semantics
  - Try to avoid running much more slowly than a compiler
    - Often simplify potentially time-consuming statements like branches, loops, and function calls
    - May allow programmer to improve analysis with annotations
- Examples: Clang Static Analyzer, Splint
  - Splint in particular tries to be very conservative when annotations are not present

# Verification-based Tools

- Try to prove correctness
  - Some map the program into an instance of a classic problem (e.g. satisfiability), then solve the problem to check safety
  - Others act as model checkers, and trace the program's execution to look for paths to unsafe states
  - Accuracy is usually a greater priority than speed
    - Approximations are still made: C is hard!
- Examples:
  - Problem-solving: Saturn, CSSV
  - Model-checking: BLAST

# Program Transformation Tools

- Demonstrate safety the easy way
  - Analyze the program to see where errors might occur
  - Add runtime checks to stop the program whenever a memory violation is about to be committed
    - Often involves storing extra state information in memory
  - Potential downside: is stopping the program acceptable?
  - May also degrade performance due to extra code
- Examples: CCured, Cyclone

# Evaluation

- Wanted an example of real analysis results
- Picked three free analysis tools, and used them to check for null pointer dereferences
  - Two compiler-like tools: Splint, Clang Static Analyzer
  - One verification-based tool: Saturn
- Wrote a small program for initial testing
- Also tested against zlib
  - Widely-used open-source compression library

# Evaluation: Test Program

```
#include <stdlib.h>

void risky(int* arg) { *arg = 1; }
void safer(int* arg) { *arg = 2; }

int main() {
    int* ptr;
    int value;
    ptr = NULL;

    value = *ptr;
    if (1 == 1) value *= *ptr;
    if (0 == 1) value += *ptr;
    risky(ptr);

    ptr = &value;
    safer(ptr);
    value = *ptr + 1;

    return 0;
}
```

Crash program

Unreachable

Safe

# Evaluation: Test Program

```
#include <stdlib.h>

void risky(int* arg) { *arg = 1; }
void safer(int* arg) { *arg = 2; }

int main() {
    int* ptr;
    int value;
    ptr = NULL;

    value = *ptr;
    if (1 == 1) value *= *ptr;
    if (0 == 1) value += *ptr;
    risky(ptr);

    ptr = &value;
    safer(ptr);
    value = *ptr + 1;

    return 0;
}
```

- Saturn made no mistakes
- Splint flagged this line (false positive)
- Clang missed this error (false negative)

# Evaluation: zlib

Utility	Splint	Clang	Saturn
Reported errors	19	0	12
Relevant errors	0	0	2
Running time (s)	0.94	30.35	1073.58

- Clang found nothing
- Splint found effectively nothing
- Saturn found two actual bugs:
  - A partially-initialized object was misused in an error case
  - A user-visible function did not check its arguments
  - *But* Saturn's analysis was glacial, and timed out on six functions

# Discussion

- For analyzing C programs, there are many options
  - Want to check a few things very quickly? Try Flawfinder
  - Want to find many bugs with a reasonable wait? Try Splint
  - Want to conduct a really thorough analysis while you are away on a weeklong vacation? Try Saturn
  - Want to avoid all buffer overflows even if it means killing your program suddenly? Try CCured
  - None are perfect, but most are better than nothing
- Safety-critical software should avoid C if possible

# Discussion

- One other issue...
  - Many free analysis tools started as research prototypes
    - ...and then the lead developer graduated...
  - Getting these to work on a large project can be a challenge
    - Build settings have to be handled for code to parse
    - Some simply will not work with more than a few simple files
  - Working within an existing compiler framework may be necessary to ensure usability