NIST Workshop on Software Measures and Metrics to Reduce Security Vulnerabilities

Measuring Software Analyzability

Andrew Walenstein

Center for High Assurance Computer Excellence

July 12, 2016
Gaithersburg, MD

The views and opinions expressed in this presentation are those of the author and do not necessarily reflect the official policy or position of BlackBerry.

POSITION

We need to better measure the analyzability of software

Because

We need to measure the security of software better.

Motivation at BlackBerry

- Center for High Assurance Computing Excellence
  - Security assurance research (collaborative)
  - Have been exploring CBMC (with Oxford University)
  - CBMC = bounded model checker
  - Turns checks into Boolean satisfiability problem

    Read code → generate SAT formula → search for solution

- Can be applied to find vulns due to integer overflow

Model checking for integer overflows

```
char* stagefrt( char* buffer, unsigned int count, unsigned int size)
{
  unsigned int i;
  unsigned int alloc_size = size * count;

  char* copy = malloc( alloc_size );

  for( i=0 ; i<count ; ++i )
    strncpy( copy + i*size, buffer + i*size, size );
  return copy;
}
```

Checkable – analyzable

```
void calls() {
  char buffer[1024];
  unsigned int over = UINT_MAX/2 + 1;

  stagefrt( buffer, 2, 2 );
  stagefrt( buffer, 2, over );
}
```

Verifies successfully using model checker

Finds overflow

Still analyzable

incl.h

```
#define lt void
#define lk strncpy
#define li const
#define lc char
#define ld unsigned
#define la int
#define ll stagefrt
#define lm malloc
#define lu for
#define lq return
#define lo main
#define lp UINT_MAX
```

main.c

```
#include “incl.h“

lc*ll(lc*lf,ld la lg,ld la le
){ld la lb;ld la
lj=le*lg;lc*lh=lm(lj);lu(lb=0;l
b<lg;++lb)lk(lh+lb*le
,lf+lb*le,le);lq lh;}la lo(){lc
lf[1024];ll(lf,2,lp/2+1);}
```

```
void stagefrt2( char* buffer, unsigned int count, unsigned int size) {
  unsigned int i;
  unsigned int alloc_size;
  if ( count < size && ( size > 12 || count < 32 ) ) {
    if ( size > 32 ) {
      if ( count < 3 )
       alloc_size = count * size; }
    else {
      alloc_size = size;
      count = 1; }}
  else if ( size > 1024 || (count < 42 && size > 2 ) ) {
     alloc_size = size;
     count = 1; }
  else {
    alloc_size = size;
    count = 1; }

  char* copy = malloc( alloc_size );
  for ( i=0 ; i<count ; ++i )
    strncpy( copy + i*size, buffer + i*size, size );
}
```

Actual problems for BMC

```
void calls() {
  extern unsigned int unstated;
  char buffer[1024];
  unsigned int over = UINT_MAX/2 + 1;

  stagefrt( buffer, 2, unstated );

  stagefrt( buffer, 2, encrypt(msg,pw)==res ? 2 : over );
}
```

FP?

Hard…

Measures/metrics approach

> "We can't hope to raise the cybersecurity bar if we don't know how to measure its height"
>
> **David Kleidermacher, CSO BlackBerry**

Theory / approach

- Measuring drives improvement and investment – objective function
- What kind of improvement do we expect?

Goals: height of the bar

Economically

- "sustainably secure systems development and operation" – economic viability question, *not* feasibility

Fantastically

- "reduce the number of vulnerabilities in software by **orders of magnitude**"

Urgently:

- A **3-7 year** goal

Goals → Automation

Automation is the key

- We want *sustainability*
  - *How can costly humans be the answer?*
- We seek *orders of magnitude* improvement
  - *How can we do this without mobilizing orders of magnitude better automation?*

Security assurance automation

- Assurance = level of confidence that software functions as intended and is free from vulnerabilities (Mitre)
- Focus: checking security properties – the root of all confidence

Tool limitations

On formal methods:

"the applicability of these techniques is currently limited to modest programs with tens-of-thousands of lines of code. Improvements in efficacy and efficiency may make it possible to apply formal methods to systems of practical complexity"

2016 **Federal Cybersecurity R&D Strategic Plan**

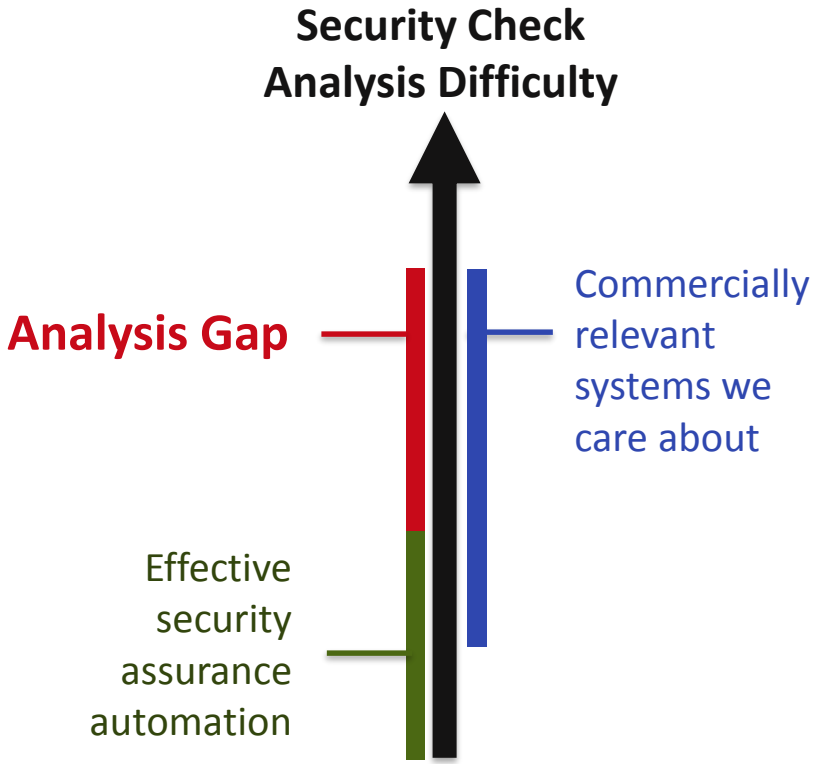
On static analysis coverage:

"Static tools only see code they can follow, which is why modern frameworks are so difficult for them. Libraries and third-party components are too big to analyze statically, which results in numerous `lost sources' and `lost sinks' – toolspeak for "we have no idea what happened inside this library." Static tools also silently quit analyzing when things get too complicated."

Jeff Williams: **Why It's Insane to Trust Static Analysis**

Analysis gap

- Gap between what we can automatically check and what we need to
- We need to reduce that gap
- Common focus: height of green -- need better checking tools
- But what about asking if we can make software more analyzable?



Making software analyzable

Gerard Holzman proposed 10 coding guidelines for safety critical code.

> When it really counts, …, it may be worth going the extra mile and living within stricter limits.. In return, we should be able to demonstrate more convincingly that critical software will work as intended

Example Rule

Restrict all code to very simple control flow constructs – do not use goto statements, … and direct or indirect recursion.

Rationale

Simpler control flow translates into stronger capabilities for verification…Without recursion, …, we are guaranteed to have an acyclic function call graph, which can be exploited by code analyzers, and can directly help to prove that all executions that should be bounded are in fact bounded.

Gerard J. Holtzman – NASA/JPL – The Power of 10: Rules for Developing Safety Critical Code

Approach 1: heuristic metrics

Recipe

1. Identify properties of code that make it hard to analyze by "typical" analyzers
2. Define metrics that relate to those code properties

Example

- Holzmann: *do not use goto statements, ..., and direct or indirect recursion*
- Measure: based on observing gotos, direct and indirect recursion

Tradeoffs

- Easy to generate; might be pretty tool-independent
- No theory to guide and assess – we don't want under- **or** over-restrictive

Approach 2: empirically measure

Recipe

1. For a given tool, identify ways in which analysis is weakened by code
2. Modify analysis tools to provide measures of analyzability

Example

- Tool can find data exfiltration tracking taint through some pointers, but not all
- Modify tool to output measures relating to its success in following the taint

Tradeoffs

- Should be possible for many (all?) tools
- How usable / actionable are the reports?

Approach 3: new software metrics?

- Problem: code not modularized well for the purposes of checking using CBMC
  - Difficult to set up a small checking "environment" or calling context
  - End up writing complex "drivers" and "stubs" and even modify the code
  - Notorious problem in model checking community
- Essentially a modularity problem – the wrong modularity?
  - The code might be considered nicely modular in terms of "ordinary" modularity metrics
  - But from the point of analyzing the code with CBMC, it was a tangled mess
  - *Is it possible to define new modularity metrics that ease CBMC-analyzabilty?*
- Not yet sure – ongoing research

Approach 4: adapt obfuscation theory?

Theory

- Obfuscation = transformations that make analyzers break
- Making software more analyzable = deobfuscation
- Approach: define metrics using available theories of obfuscation potency

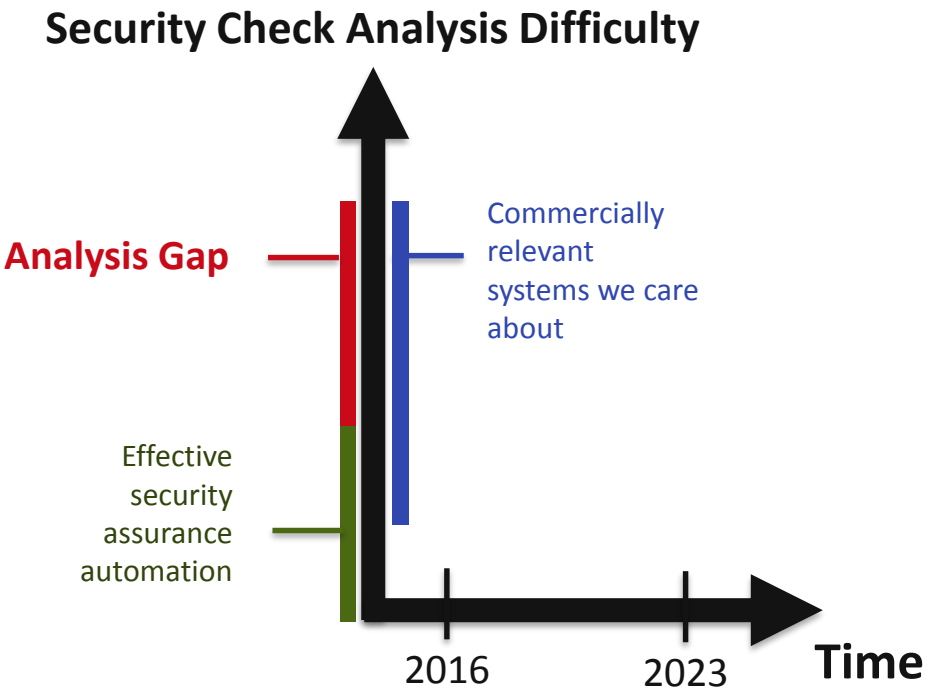
Example

- Giacobazzi and Dalla Preda use Abstract Interpretation to define obfuscation in terms of transformations that make analyzers *incomplete*
  - Yields a theoretical model for defining potency and comparing potency
  - Can we use this approach to define metrics on code?

Analysis gap and the future

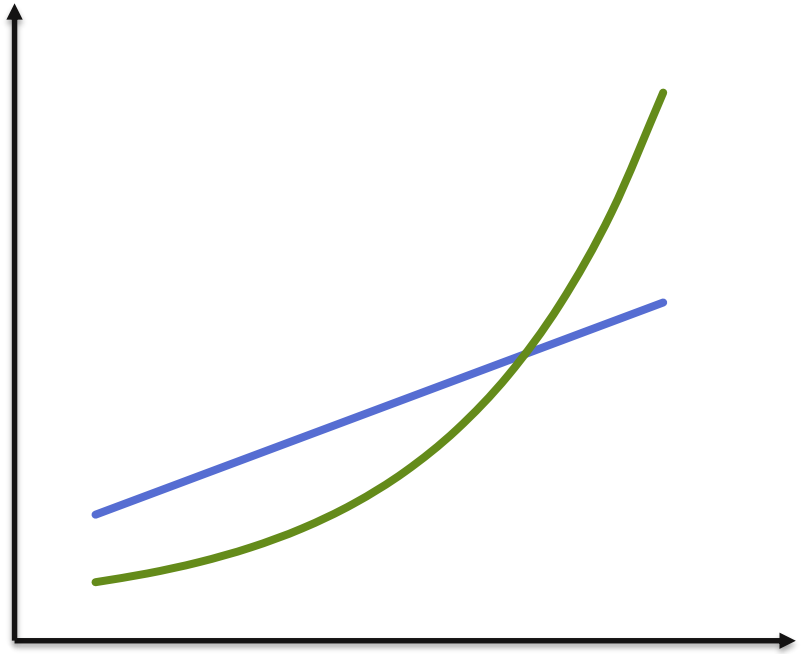
- What does the future hold for automation of security analysis?
- Where should we place our bets for making ***orders of magnitude*** improvement?

Imaging charting the green and blue peaks into the future….



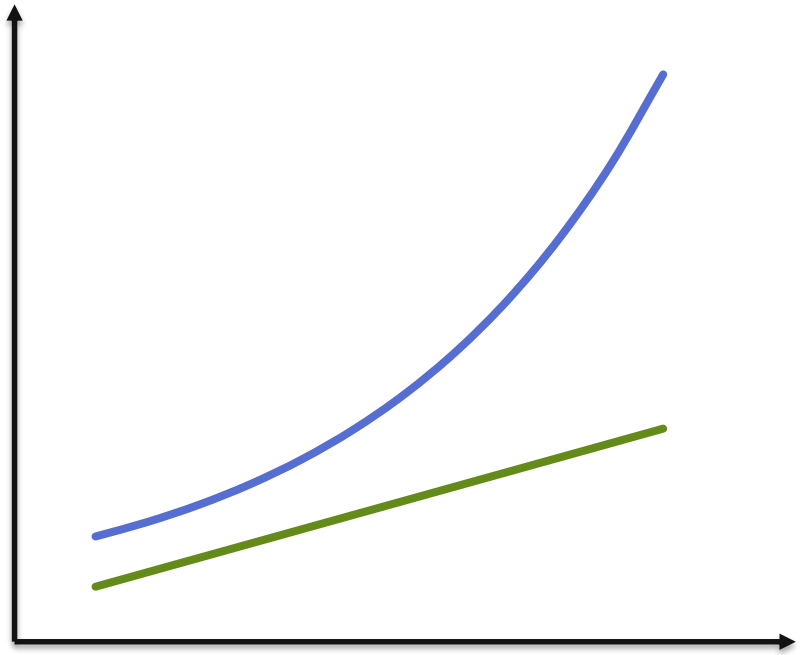
Automation revolution (closed universe)

- Fantastic improvement in automated security assurance
  (Henry Gordon Rice is astounded)
- Catch up to and surpass current needs
- Effective elimination of classes of vulnerabilities



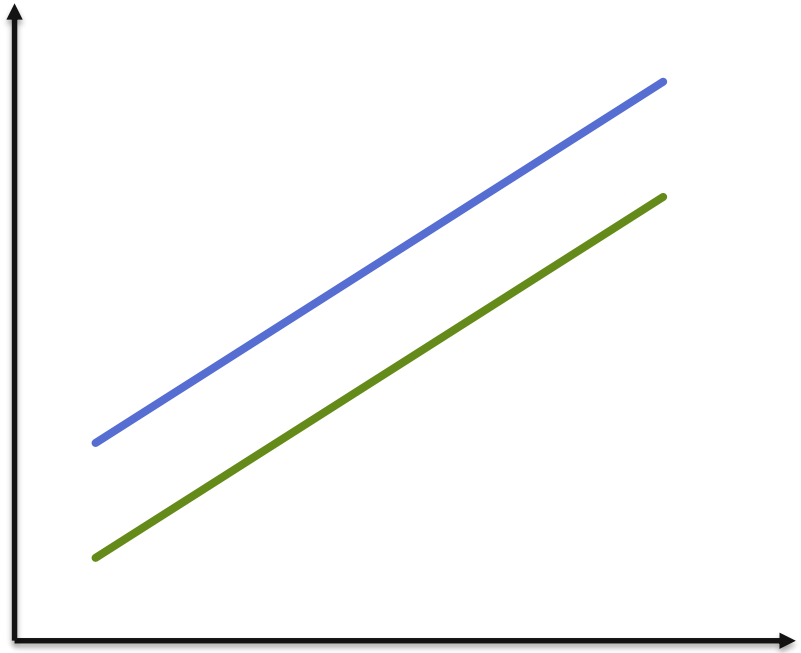
Regression (open universe)

- Analysis loses ground
- Not promising for the future



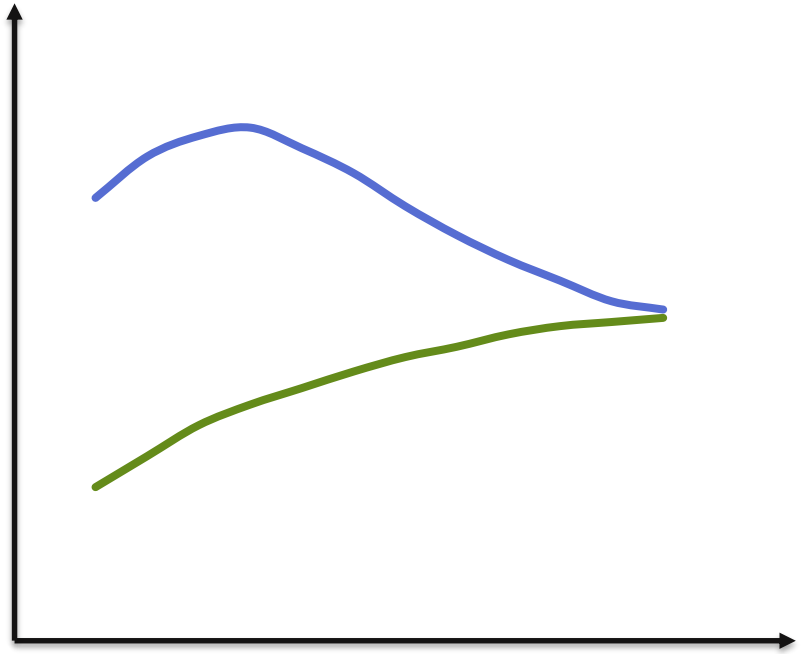
Stasis (flat universe)

- Analyzability of software rises at same rate as our tool abilities
- Huge improvement possible for some systems
- But gap is same = same lack of assurance…have we succeeded on our goals?



Rendezvous model

- Automation improves slowly
- Analyzability is measured and slowly improves analyzability of code



Bets?

Sustainable orders of magnitude increase in security in 5-7 years

- *Where is your bet how it will most likely come to pass?*

A. Non-automation

- Humans, processes, standards, …

B. Improvements in automation

- Improved & cheaper formal methods, program analysis, test generation…

C. Improvements in software analyzability

- Processes and tools that generate more analyzable code

Possible steps forward?

1. Defining new measures, metrics
   - We can start defining as best we can and measure their utility.
2. Modifying tools to support analyzability improvements
   - Reporting loss of completeness/precision – and highlight problem code features?
   - Automated de-obfuscators?
3. Language / framework design
   - Can we make analyzability a key design feature?
4. Process change
   - Analyzability as a quality?
   - Analyzability gap as a type of maintenance debt?