



INSTITUTE FOR DEFENSE ANALYSES

## **If It Works, It's Legacy: Analysis of Legacy Code**

David A. Wheeler

June 27, 2018

Approved for public  
release; distribution is  
unlimited.

IDA Non-Standard  
NS D-9161

INSTITUTE FOR DEFENSE  
ANALYSES  
4850 Mark Center Drive  
Alexandria, Virginia 22311-1882



*The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.*

#### About This Publication

This work was conducted by the Institute for Defense Analyses (IDA). The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

#### For more information:

David A. Wheeler, Project Leader  
dwheeler@ida.org, 703-845-6662

Margaret E. Myers, Director, Information Technology and Systems Division  
mmyers@ida.org, 703-578-2782

#### Copyright Notice

© 2018 Institute for Defense Analyses  
4850 Mark Center Drive, Alexandria, Virginia 22311-1882 • (703) 845-2000.

This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (a)(16) [Jun 2013].

INSTITUTE FOR DEFENSE ANALYSES

IDA Non-Standard NS D-9161

**If It Works, It's Legacy:  
Analysis of Legacy Code**

David A. Wheeler



## **Executive Summary**

The “Sound Static Analysis for Security (SSAS) Workshop” was held on June 27-28, 2018, at the National Institute of Standards and Technology (NIST) facility in Gaithersburg, MD. This two-day workshop focused on decreasing software security vulnerabilities by orders of magnitude, using the strong guarantees that only sound static analysis can provide. The workshop was aimed at developers, managers and evaluators of security-critical projects, as well as researchers in cybersecurity. The program featured experts on sound static analysis applied to security around three theme topics: analysis of legacy code, use in new development, and accountable software quality. Dr. Wheeler presented this keynote presentation to kick off the discussion on analysis of legacy code.

This presentation provides an overview of how sound static analysis approaches can facilitate the development of higher quality and more secure versions of existing software. Because technical fads and requirements change all the time, if a system works, it’s a legacy system. Rewriting an existing system from scratch is almost always a foolish decision, so ways of dealing with existing “legacy” systems are needed. Existing systems have problems that testing and non-sound static analysis cannot fully address, so there are reasons to use sound static analysis on existing systems. Unfortunately, legacy systems are challenging for sound static analysis tools, e.g., because they are often large, not designed to be analyzed, and written in languages that are hard to analyze.

To address this, an engineering mindset, rather than a scientific or mathematical mindset, is indicated. Examples of approaches that can help include incremental localizable improvement, sound analysis of high-level models (particularly of protocols), lightweight formal methods (which emphasize partial specification and focused application), easing the combination of sound analysis with other approaches (e.g., strengthening tests, human review, and start-up checks), analysis of prevention/hardening/detection/response mechanisms, improving tools to handle scale and real constructs (not subset), improving tools to handle larger scales, providing useful feedback on “how to change the code to be analyzable,” and improving the interactions between developers and makers of sound tools. Releasing sound analysis tools as open source software (OSS) enables users to improve the tools so that they can be purposed or repurposed to a specific use. In short, we need sound static analysis tools that can work with and improve legacy systems, and we need an engineering mindset to help us do so.





# Institute for Defense Analyses

4850 Mark Center Drive • Alexandria, Virginia 22311-1882



# If It Works, It's Legacy: Analysis of Legacy Code

Keynote for Analysis of Legacy Code Session,  
Sound Static Analysis for Security Conference

June 27, 2018

by

David A. Wheeler

PhD, CISSP

[dwheeler @ ida.org](mailto:dwheeler@ida.org)

## IDA | Common definition

- A legacy system is:
  - An old method, technology, computer system, or application program “of, relating to, or being a previous or outdated computer system.”
  - Often a pejorative term.
  - Can also imply that the system is out of date or in need of replacement.



## IDA | If it works, it's legacy

- Current technical fad changes all the time
  - “Hype driven development”
  - Any technology soon becomes legacy
- Requirements change all the time
- If a system works, it's a legacy system

If your job is to make working software,  
then your job is to  
**create & maintain legacy systems**

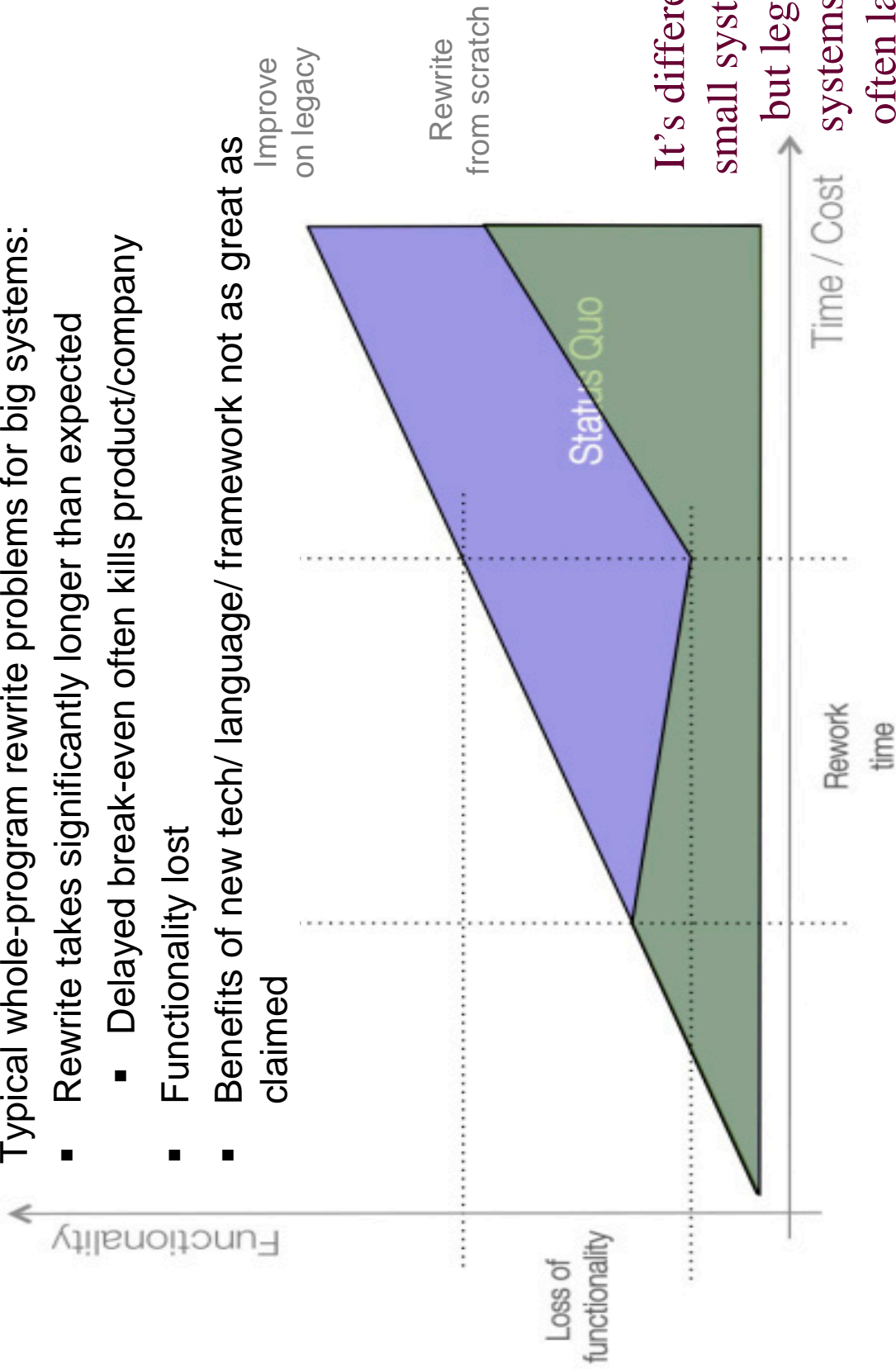
# Joel Spolsky “Things you should never do...” for big systems

- “[Netscape made] the **single worst strategic mistake** that any software company can make: They decided to **rewrite the code from scratch**.”
- “The idea that new code is better than old is patently absurd.”
  - “Old code has been used. It has been tested.”
  - “Lots of bugs have been found, and they’ve been fixed.”
- “When you throw away code and start from scratch...”
  - “You are throwing away... knowledge... bug fixes. Years of ... work.”
  - “You are throwing away your market leadership....”
  - “You are putting yourself in an extremely dangerous position... unable to make any strategic changes or react...”
  - “You are wasting an outlandish amount of money...”
- “When you start from scratch there is absolutely no reason to believe that you are going to do [better than before].”
- “[throwing away] is dangerous [for] large scale commercial applications... [replacing an experimental function is fine. Refactoring a class is fine.]”
- **“throwing away the whole program is a dangerous folly.”**

# Why a total rewrite is usually the worst idea for big systems

Typical whole-program rewrite problems for big systems:

- Rewrite takes significantly longer than expected
  - Delayed break-even often kills product/company
- Functionality lost
- Benefits of new tech/ language/ framework not as great as claimed

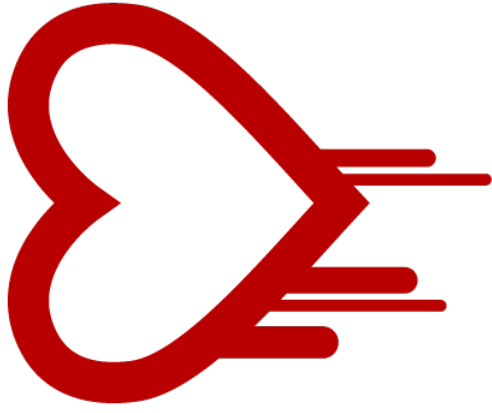


# IDA | So you want to replace the Linux kernel...

- Linux kernel mainline v4.17 released 2018-06-03
  - Analyzed using SLOCCount version 2.26
  - Used COCOMO model, kernel-specific parameters
    - Semi-detached, -- effort 4.64607 1.12 -- schedule 2.5 0.35
  - Data "generated using David A. Wheeler's 'SLOCCount'"
- Outrageously expensive to redevelop from scratch
  - Total Physical Source Lines of Code (SLOC) = 16,974,012
  - Development Effort Estimate, Person-Years = 21,147.70
  - Schedule Estimate, Years = 16.23
  - Estimated Average Number of Developers = 1,303
  - Total Estimated Cost to Develop = \$ 2,856,766,709\*
- Whether or not you believe the model, it's expensive & time-consuming to rewrite large software component
- Kernel is just 1 of MANY system components

\* In year 2000 dollars, given average salary = \$56,286/year, overhead = 2.40. For more about the method used, see "The Linux Kernel: It's Worth More!" by David A. Wheeler, <https://www.dwheeler.com/essays/linux-kernel-cost.html> Linux 4.17 removed a lot of code; see Larabel's "Linux Set To Shed Nearly 500k Lines Of Code By Dropping Old CPUs", [https://www.phoronix.com/scan.php?page=news\\_item&px=Linux-4.17-Gutting-Old-CPU](https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.17-Gutting-Old-CPU)

# IDA | But our legacy systems have problems...



Credit: Heartbleed logo from <http://heartbleed.com/> - "Heartbleed logo is free to use, rights waived via [CC0](https://creativecommons.org/licenses/by/4.0/)." Stagefright logo via <https://medium.com/threat-intel/bug-branding-heartbleed-14ef1a64047f>. Shellshock logo from [https://en.wikipedia.org/wiki/Shellshock\\_\(software\\_bug\)](https://en.wikipedia.org/wiki/Shellshock_(software_bug)).

# ... and alternatives to sound static analysis tools are weak

- Can't test our way out
  - Dynamic analysis only reports what it tests
  - Can't test real-world programs with useful fraction of its inputs
    - “Add two 64-bit integers” has  $2^{64}$  possible inputs
    - 1.7 quadrillion years (1 million 8-core 4GHz CPUs, 5 cycles/test)
  - If a test finds a vulnerability, we know we're vulnerable, but if it doesn't find a vulnerability, we know relatively little
  - In short: dynamic analysis methods (including testing) can only show whether something is insecure – they cannot show whether it's secure
- Unsound static analysis
  - Can more easily cover code, and can be very helpful
  - Limited confidence in its results – heuristic, not “guaranteed”
- Sound static analysis tools let us state software does or doesn't do something with very high confidence
  - In many cases, this info (“is it secure”) would be *valuable*
  - Especially if we're trusting the code for something important

# IDA | Heartbleed & unsound static analysis



- OpenSSL routinely analyzed by unsound SA tools
  - None in use found Heartbleed vulnerability
  - One (CQual++) could, but difficult to use/interpret
- Heartbleed could be found in various ways, incl. sound SA:
  1. Thorough negative testing in test cases (DA)
  2. Fuzzing w/ address checking & standard allocator (DA)
  3. Compiling w/ address checking & standard allocator (HA)
  4. Focused manual spotcheck w/ validation of every field (SA)
  5. Fuzzing with output examination (DA)
  6. Context-configured source code weakness analyzers (SA)
  7. Multi-implementation 100% branch coverage (HA)
  8. Aggressive run-time assertions (DA)
  9. Safer language (SA)
  10. Sound (“complete”) static analyzer (**sound SA**)
  11. Thorough human review / audit (SA)
  12. Formal methods (**specific type of sound SA**)

Key:

DA = Dynamic analysis  
HA = Hybrid analysis  
SA = Static analysis

# IDA | What is “sound static analysis”?

- Static analysis: “Examines... system/software without executing it, examining source code, bytecode, and/or binaries” [SOAR2016]
  - Includes formal methods and some other kinds of tools
- Sound: “**Every tool report is correct**” (given its assumptions)
  - *Not* based on heuristics
- Issue: Is tool’s purpose *finding bugs* or *verifying correctness*?
- Finding bugs: (NIST) SATE V Ockham Sound Analysis Criteria:
  - “A site is a location in code where a weakness might occur. A buggy site is one that has an instance of the weakness [i.e., an input [will] cause a violation...
  - A finding is a definitive report about a site. In other words, that the site has a specific weakness (is buggy) or that the site does not have a specific weakness (is not buggy).
  - Sound means every finding is correct. The tool need not produce a finding for every site; that is completeness.”
- Verifying correctness: If tools says item is correct, it’s correct

Sources: (1) “SATE V Ockham Sound Analysis Criteria” <https://samate.nist.gov/SATE5OckhamCriteria.html>

(2) State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016. David A. Wheeler & Amy E. Henninger. <https://www.acq.osd.mil/se/docs/P-8005-SOAR-2016.pdf>



# Legacy programs hard for static analysis tools

- Tend to be large & complex (tool scaling problems)
  - No, users will **not** give up functionality to make them smaller and thus easier to analyze
  - Size/complexity solves real-world requirements
- Requirements and design not fully understood
- Not designed to be analyzed (a key reason Heartbleed wasn't found)
- Often written in languages that are difficult to analyze
  - Non-strict typing & manual garbage collection (C,C++)
  - Many undefined parts (C,C++)
  - Dynamically-typed (can't infer types) (JavaScript, Python, Ruby)
  - Arrays, pointers, assignment, loops, inheritance, etc.
  - Semantics not rigorously defined
  - Often mixed languages
- Often depend on poorly-understood components
  - E.g., reused software with no source, different language...
  - Often must “understand” framework for good static analysis
  - Transitive dependencies

## IDA | Common “wisdom”

- To use sound static analysis (e.g., to prove a program correct), the program must be written to be analyzable/provable (due to limitations of the tools)
- Most programs are *not* written that way
- Recreating (big) programs is usually folly
- Therefore, sound static analysis (e.g., proving programs correct) is usually folly

**But is this really the case?**

## IDA | Need engineering mindset

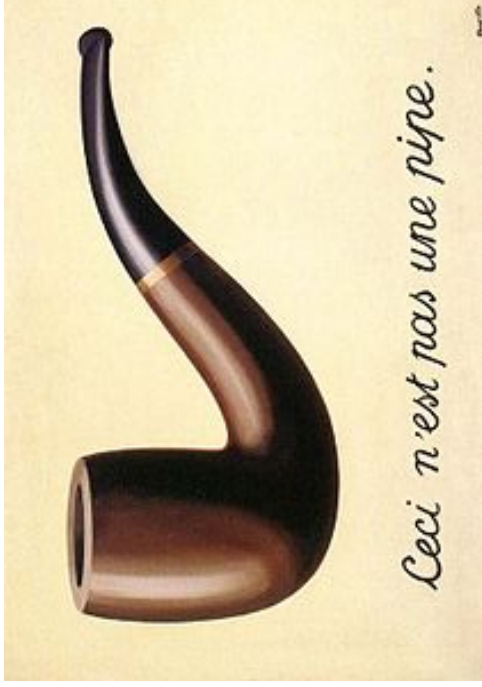
- Don't confuse these (my definitions):
  - **Mathematics:** Rigorous deduction of conclusions from assumptions (not necessarily real world)
  - **Science:** Induction of general principles, rigorously tested via experiments and observations
  - **Engineering:** Developing systems that solve human problems within constraints and using available knowledge
- Developing software to solve human problems is fundamentally **engineering**

# Common confusion: Software ≠ Mathematics

- Mathematics deduces conclusions from models/assumptions
  - Assumptions may not be realistic!
  - Conclusions may not match reality either
- SATE V Ockham Criteria acknowledges this:
  - “All reasoning is based on models, assumptions, definitions, etc. (collectively, “models”)... we will decide if an unexpected finding results from a reasonable model difference...”

# IDA | The map is not the territory

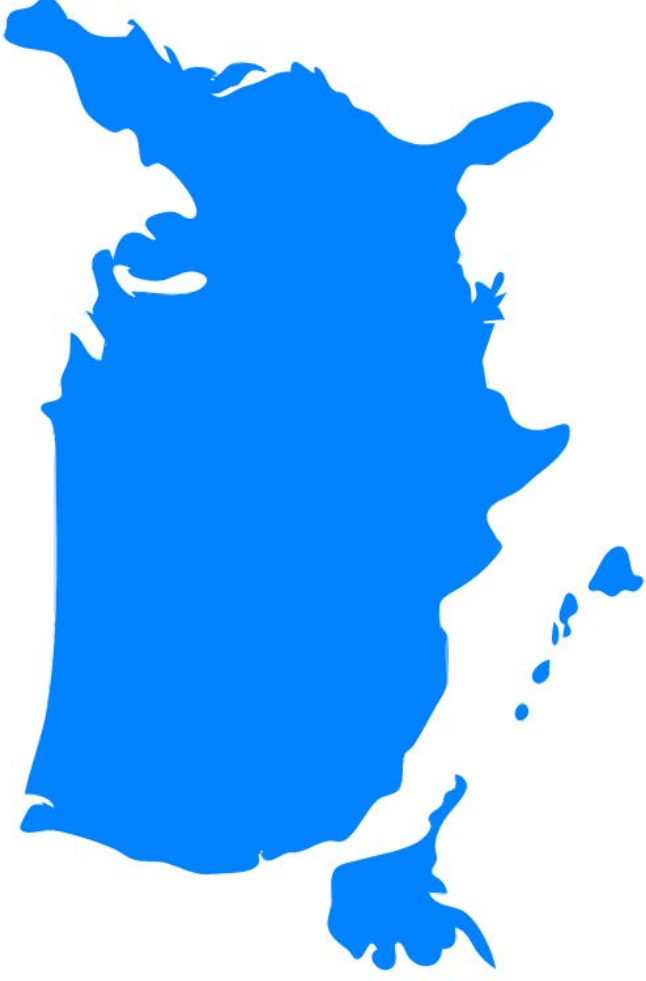
- Mathematics is used to *model* the world
- Don't confuse the **map** with the **territory**
  - Easy to do with software and math because both are abstract



“The Treachery of Images”  
(1928–29) by René Magritte

“The famous pipe. How people reproached me for it! And yet, could you stuff my pipe? No, it's just a representation, is it not? So if I had written on my picture ‘This is a pipe’, I'd have been lying!” – René Magritte

**IDA** | And yet... there are good reasons we use maps!



“I have a map of the United States... actual size. It says, ‘Scale: 1 mile = 1 mile.’ I spent last summer folding it.” – Steven Wright (comedian)

Source: <http://www.wright-house.com/steven-wright/steven-wright-Be.html>

Credit: <https://openclipart.org/detail/202446/usa-map-silhouette>

“All models are wrong,  
but some are useful.”  
— George Box

See [https://en.wikipedia.org/wiki/All\\_models\\_are\\_wrong](https://en.wikipedia.org/wiki/All_models_are_wrong)

E.g., Box, G. E. P. 1979. "Robustness in the strategy of scientific model building". In *Robustness in Statistics*, edited by R. L. Launer & G. N. Wilkinson, pp. 201–236. Cambridge, MA: Academic Press.

## IDA | **Sound analysis makes assumptions, and that's ok**

- **Sound analysis must make assumptions**
  - Programming language spec accurate?
  - OS kernel correct?
  - Compiler correct?
  - CPU correct? (Meltdown, Spectre)
  - Transistors working? (Gamma rays, EMP, quantum mechanics is probabilistic)
  - Sound analysis tool correct?
- **Sound analysis can still be very useful**
  - It can often provide a much *higher* level of confidence than alternatives (within constraints)



# IDA | Engineering constraints

- Many constraints, e.g.:
  - Functional requirements
  - Cost (~labor hours)
  - Calendar time
  - Quality (reliability, security, safety, ...)
  - Available (reusable) components
  - Environment / existing systems
  - Risks (probability & impact)
- Many trade-offs (decisions to be made)
  - Each decision is a trade-off between constraints
  - Want perfect quality? No risk? Can't afford it
  - If “quality” is in trade space, *can* afford to apply sound static analysis approaches in at least some contexts



# IDA | Key: Incremental localizable improvement

- Make it *easy* to start using the tool incrementally
- *Do not* require massive widespread changes
  - Don't require big-bang "flag day" changes where everything has to change at the same time
- Support incremental spec-writing
  - Ideally, don't require a specification to be written at all
  - Allow users to add small specific specifications that provide small additional value
- Support incremental/inferred typing for dynamic languages that don't require typing
  - Provide value when additional optional information provided
- It's okay to require some changes
  - You want some changes (fix bugs)
  - But must be scalable

# Learn lessons from the Python 2->3 transition (1)



- Python version 3.0 released on 3 December 2008
  - Significant but not massive changes: “print” became a function, string semantics changed, some library renaming
- Original transition plans unrealistic
  - Python 2 and 3 interpreters separate codebases
  - Each couldn’t run code in the other version (incompatible) - MISTAKE
  - “2to3” auto-translated but didn’t/couldn’t work reliably on real programs
  - Projects had to hand-translate an *entire* program from 2 to 3 all at once, and *all* its transitive dependencies had to transition first (not just 99%)
  - Impossible for every library to simultaneously switch to Python 3
  - Version 3 mostly ignored for many years

# Learn lessons from the Python 2->3 transition (2)



- Transition can only happen when it's easy/practical
  - V3 modified, added libraries so code can run on both v2 and v3
  - Tools created to help modify v2 code so it also works on v3
  - Allowed incremental transition of program fragments and libraries
- 10 years later, transition much more successful (though incomplete!)
  - Mercurial requires v2
  - Google *working* on v3 in App Engine Standard Environment (announced June 2018)

# IDA | Approaches system developers can use

- Sound analysis of high-level models (instead of code)
  - Particularly useful with protocols
- Lightweight formal methods
  - Emphasize partial specification, focused application, and analysis of models
- Ease combining sound analysis with other approaches
  - Use sound analysis to determine what it affordably can
  - Use other approaches where it stops working: e.g., strengthening tests, human review, and start-up checks
  - Requires sound tools to *tell* you where it's not working as narrowly as possible (not just “can’t do that” or hang)
- Analysis of prevention/hardening/detection/response mechanisms (vs. system itself)
  - Perhaps you can analyze the mechanism instead of the system
- Designate local champion & start small

## IDA | Approaches toolmakers can apply\* (1)

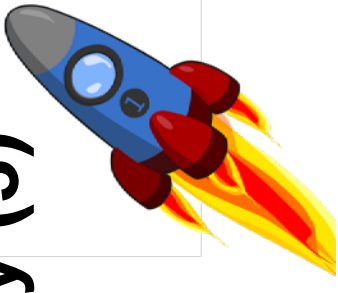
- Improve tools to handle (larger) scale
  - Support remote/lots of CPUs where that makes sense
- Provide useful feedback on “how to change the code to be more analyzable”
  - Otherwise, users won’t know what info the tool needs
- Good documentation (including quick tutorials)
- Provide more/better examples
- Improve interactions between developers and makers of sound tools
  - Releasing sound analysis tools as open source software (OSS) enables users to improve the tools so that they may be repurposed or repurposed to a specific use

\* If you’re a tool user, ask toolmakers to strive for these

## IDA | Approaches toolmakers can apply (2)

- Improve tools to handle full programming languages as used in the real world
  - Not a subset, not just the formal spec, but as actually used (as much as you can)
  - Arrays, pointers, assignment, loops, inheritance, manual garbage collection, etc.
  - e.g., SPARK work on borrow-checked pointers
- Identify/support common frameworks
  - Greatly simplify analysis and giving practical output
  - Make it easy for user to describe their framework (there are too many to support all)
- Ease integrating these tools into both developers' environments and in the development processes

# IDA | Approaches toolmakers can apply (3)



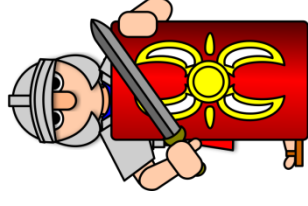
- *Easy & fast* “start to first win”
  - Make it *easy* to get started and have a win
  - 0 (or near-0) install, delay need for docs
  - Be able to do something useful win 0/near-0 configuration and no new info (e.g., spec)
  - Pre-canned focused solutions
    - Find all undefined constructs (C, C++)
      - Language standard and compiler authors have become adversaries – developers want tools to help!
      - Support non-standard extensions that prevent some
    - Find deadlocks or livelocks
  - Once a user gets *some* value from a tool, they’re more willing to invest time in it
    - Be a good date; commitment happens later



## IDA | Approaches toolmakers can apply (4)

- Ease refactoring
- Sometimes refactoring is needed to make a tool work or work better
  - Ideally it isn't, so try not to, but it's a reality
  - Clearly explain what's needed and the benefit
  - Provide/work with tools to maximally automate *trustworthy* refactoring
    - Make refactoring easy to do
    - Look for ways to give confidence that the refactoring hasn't broken anything
  - Limit the refactoring needed at any one time
  - Suggest priorities (what's most important?)
  - Listen to users, work with representative ones

# IDA | Protect the software from attack



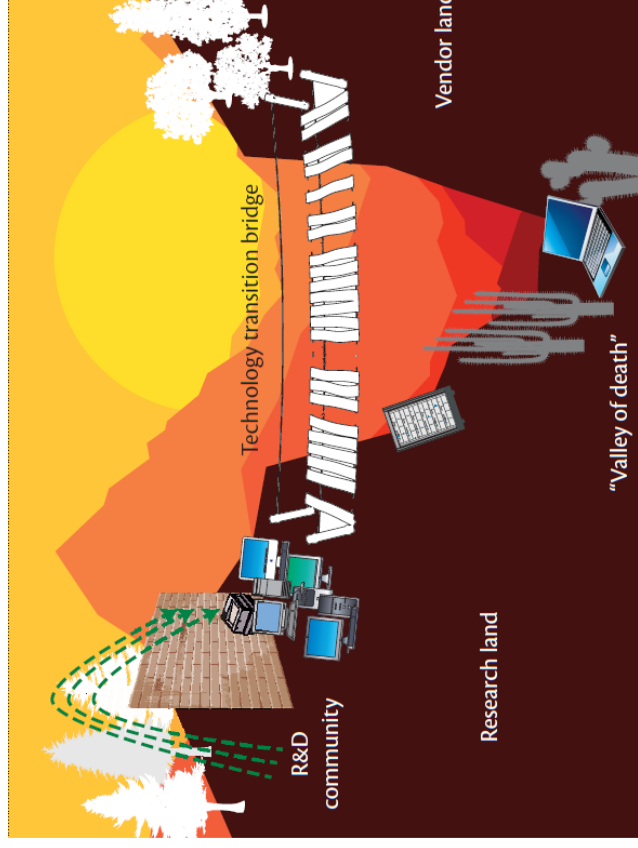
- Legacy systems often valuable/control valuable things
  - Confidentiality and/or integrity may be critical
  - Outsiders may try to break in
  - Developers may be insider threats (feel under-appreciated, want revenge, accept bribes, work for adversary, ...)
  - Tools may themselves become attack vectors
- Tools should:
  - Be usable while **disconnected** from the Internet
  - Be usable in a sandbox (limited privilege)
  - Run even where developers aren't fully trusted
  - NOT ask for more privileges than clearly needed
    - If you don't need to write, don't ask for the privilege
  - Allow repeatable re-execution (verify results) (time heuristics!)
  - Make it easy to check-in inputs and results (verify results)
  - Consider releasing the tools as OSS (reviewability)
- Users should demand such tools

## IDA | Long-term: Education and training

- Software developers often lack knowledge
- Often don't know first-order logic (FOL), the basis of nearly all spec languages
  - Need to “reduce the fear”
- Often unaware of sound analysis
- Often clueless about secure software development
- Need to improve their education level
- Many software developers don't have computer-related college education
- Need to do *more* than cover in college

# IDA | Technology transition

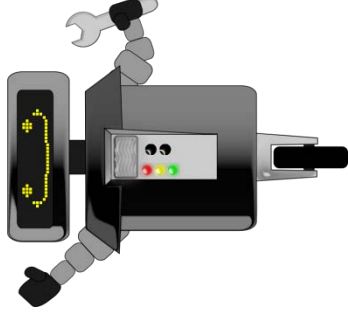
- Today, getting sound static analysis tools into widespread use is a technology transition problem
- Standard problem: Crossing the “Valley of death” from R&D to widespread real-world use
- Apply lessons learned from other successful transitions



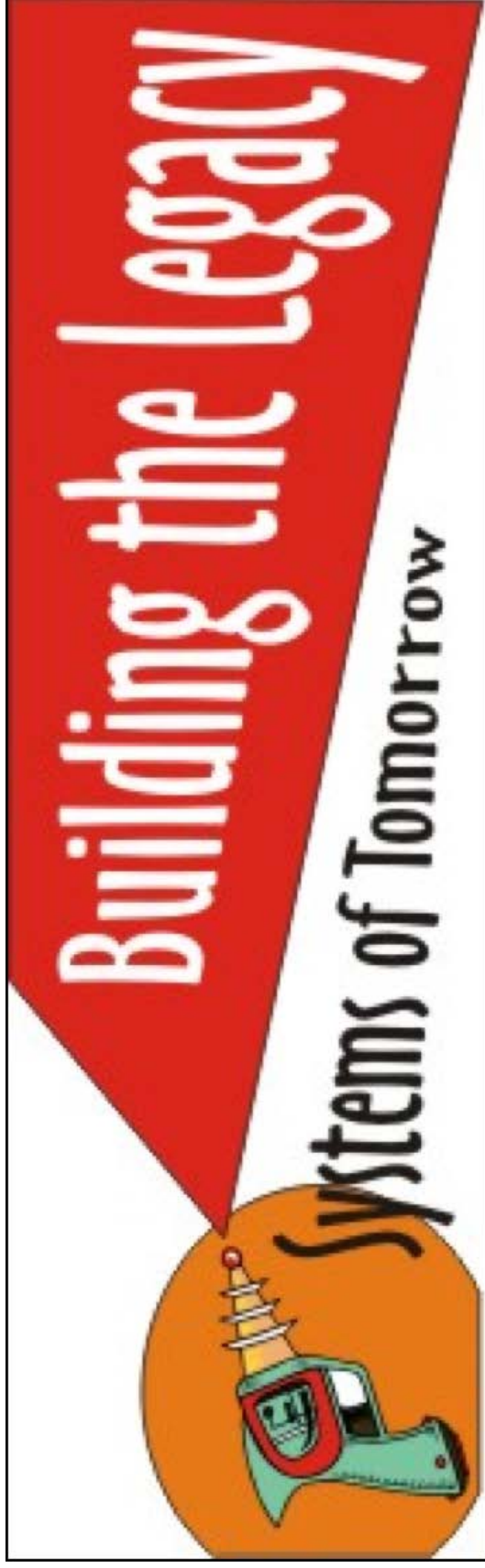
Source: “Crossing the “Valley of Death”: Transitioning Cybersecurity Research into Practice” by Douglas Maughan, David Balenson, Ulf Lindqvist, and Zachary Tudor, Published in IEEE Security and Privacy, Volume 11 Issue 2, March 2013, Pages 14-23.  
<https://dl.acm.org/citation.cfm?id=2498497>

# IDA | Conclusions (1)

- There will always be “legacy” systems
  - “Legacy” just means “it works”
  - Reasons to *not* rewrite big systems from scratch
- We need sound static analysis tools that *can* work with and improve legacy systems
  - There are a variety of ways to do that
  - Incremental improvement, easy first win, etc.
- Need an *engineering* mindset
  - Even sound tools aren’t *total* guarantees
  - There’s a trade space – make and apply tools so that their benefits exceed their costs



You are.....



Make future generations grateful  
for what they'll be using!

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188		
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. <b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b>					
1. REPORT DATE (DD-MM-YY) 00-06-18		2. REPORT TYPE Non-Standard		3. DATES COVERED (From – To)	
4. TITLE AND SUBTITLE If It Works, It's Legacy: Analysis of Legacy Code			5a. CONTRACT NUMBER HQ0034-14-D-0001		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBERS		
6. AUTHOR(S) David A. Wheeler			5d. PROJECT NUMBER ITSDPB		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Institute for Defense Analyses 4850 Mark Center Drive Alexandria, VA 22311-1882			8. PERFORMING ORGANIZATION REPORT NUMBER NS D-9161		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Institute for Defense Analyses			10. SPONSOR'S / MONITOR'S ACRONYM IDA		
			11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION / AVAILABILITY STATEMENT This draft has not been approved by the sponsor for distribution and release.					
13. SUPPLEMENTARY NOTES Project Leader: David A. Wheeler					
14. ABSTRACT This presentation provides an overview of how sound static analysis approaches can facilitate the development of higher quality and more secure versions of existing software. Because technical fads and requirements change all the time, if a system works, it's a legacy system. Rewriting an existing system from scratch is almost always a foolish decision, so ways of dealing with existing "legacy" systems are needed. Existing systems have problems that testing and non-sound static analysis cannot fully address, so there are reasons to use sound static analysis on existing systems. Unfortunately, legacy systems are challenging for sound static analysis tools, e.g., because they are often large not designed to be analyzed and written in languages that are hard to analyze. To address this, an engineering mindset, rather than a scientific or mathematical mindset, is indicated. Examples of approaches that can help include incremental localizable improvement, sound analysis of high-level models (particularly of protocols), lightweight formal methods (which emphasize partial specification and focused application), easing the combination of sound analysis with other approaches (e.g., strengthening tests, human review, and start-up checks), analysis of prevention/hardening/detection/response mechanisms, improving tools to handle scale and real constructs (not subset), improving tools to handle larger scales, providing useful feedback on "how to change the code to be analyzable," and improving the interactions between developers and makers of sound tools. Releasing sound analysis tools as open source software (OSS) enables users to improve the tools so that they be purposed or repurposed to a specific use. In short, we need sound static analysis tools that can work with & improve legacy systems, and we need an engineering mindset to help us do so.					
15. SUBJECT TERMS Sound static analysis; static analysis; software assurance; security; technology transition; vulnerability analysis; legacy; legacy software; engineering; engineering approach; trade-offs; mathematics; incremental improvement; open source software; lightweight formal methods; scale; large software systems					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT  Unlimited	18. NUMBER OF PAGES  34	19a. NAME OF RESPONSIBLE PERSON Institute for Defense Analyses
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include Area Code)

