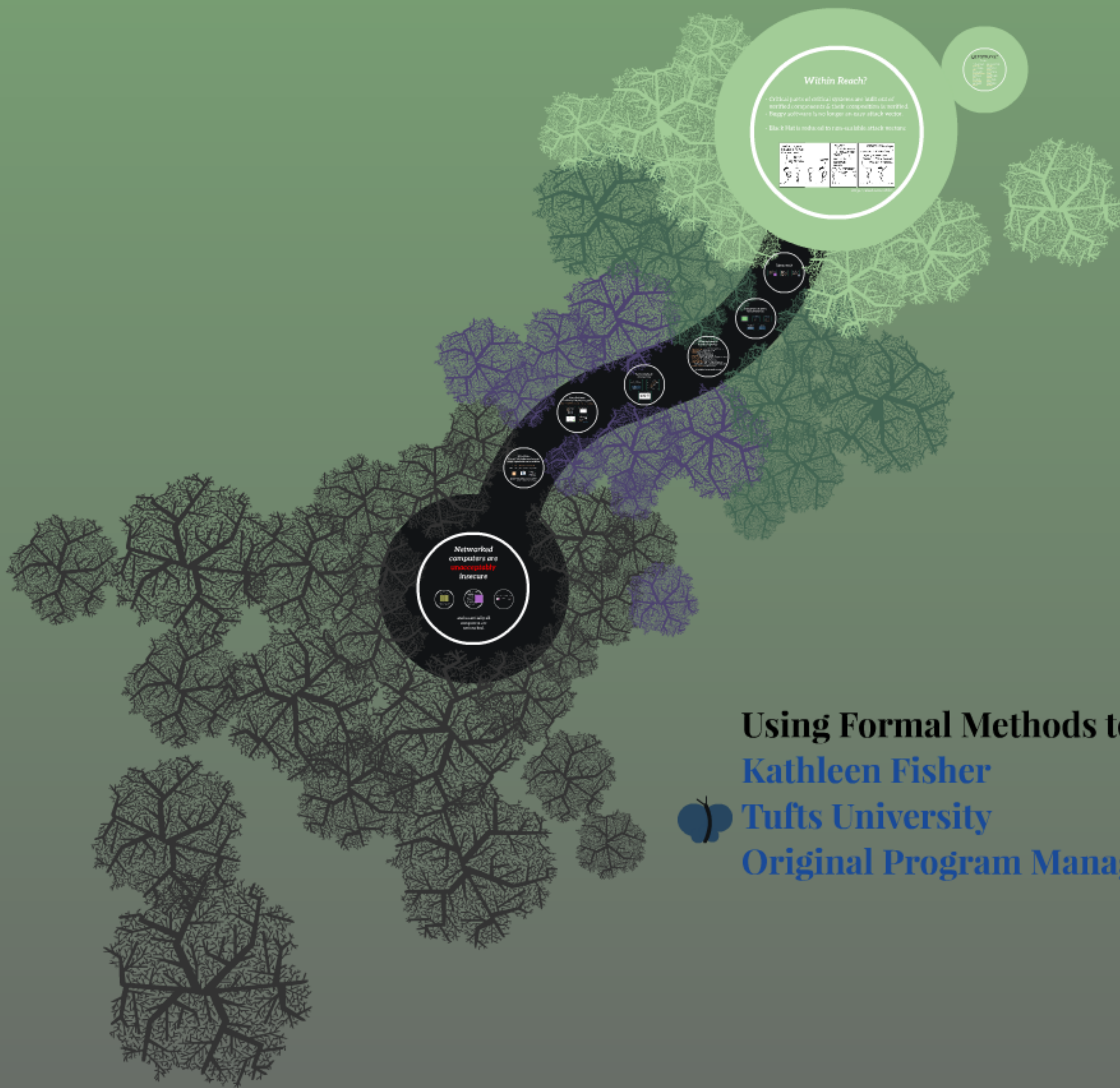
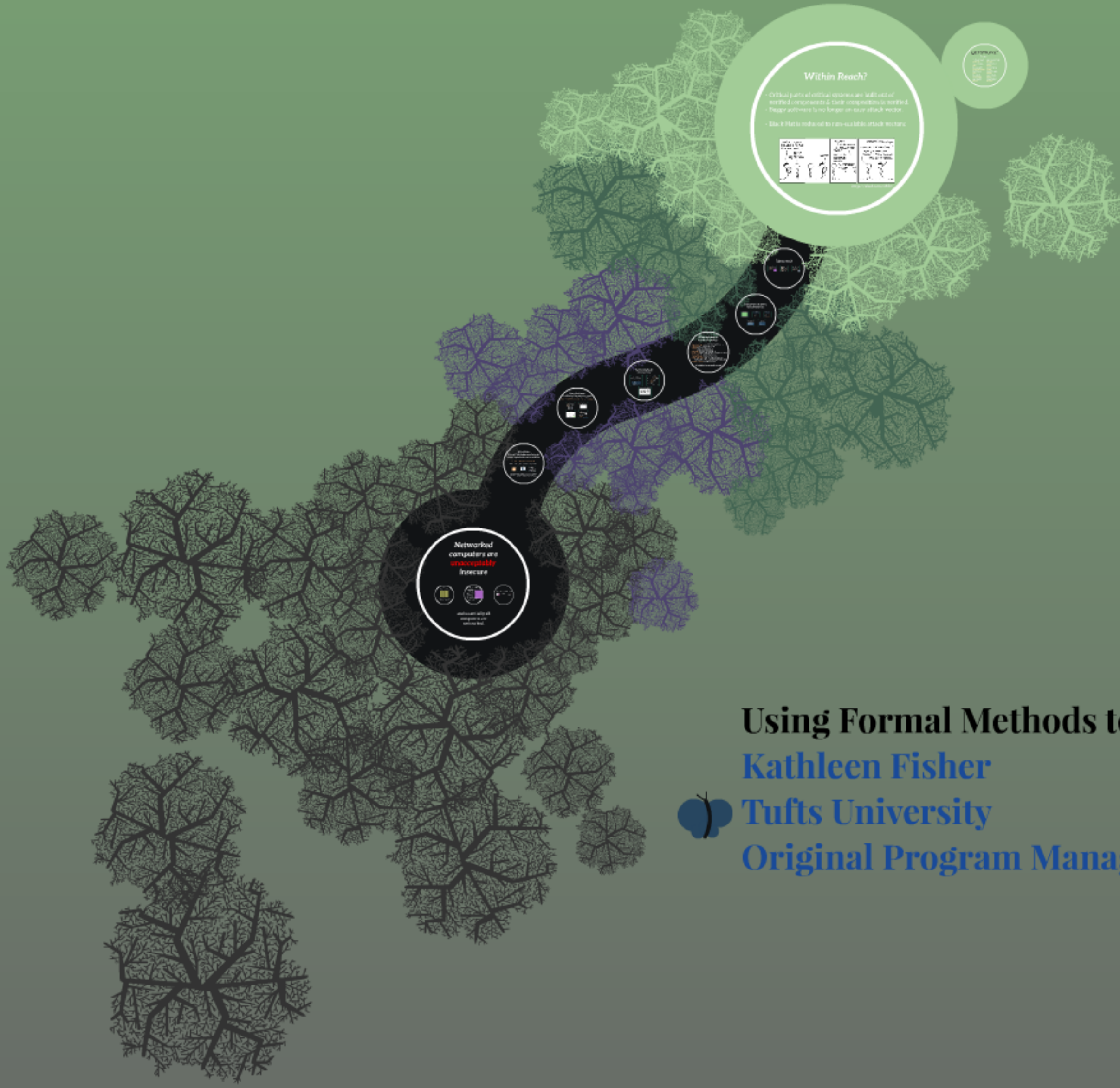


Using Formal Methods to Eliminate Exploitable Bugs

Kathleen Fisher
Tufts University
Original Program Manager, DARPA's HACMS program

Using Formal Methods to Eliminate Exploitable Bugs

Kathleen Fisher
Tufts University
Original Program Manager, DARPA's HACMS program



Using Formal Methods to Eliminate Exploitable Bugs

Kathleen Fisher
Tufts University
Original Program Manager, DARPA's HACMS program

Networked computers are ***unacceptably*** *insecure*

and essentially all computers are networked.

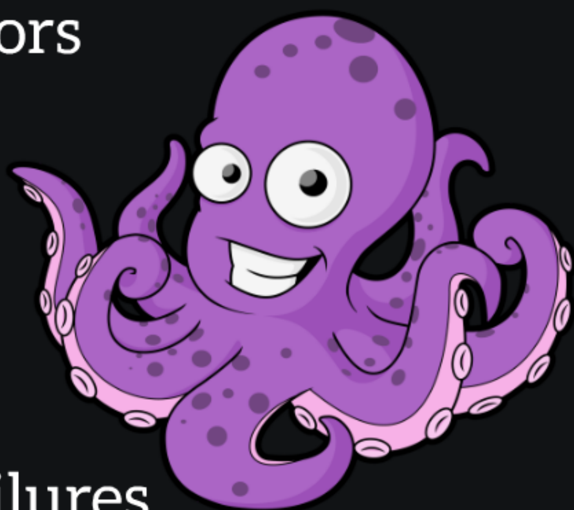
Today
many things are
networked computers.

http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

And attackers take control
with distressing ease.

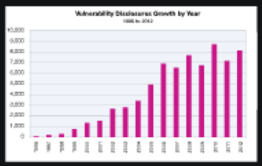
Why so insecure?

- Faulty Design
- Buggy Specifications
- Implementation Errors
- Side-channel leaks
- Misconfiguration
- Gullible users
- Weak passwords
- Malicious insiders
- Physical security failures
- Reliance on 3rd party software
- Faulty/malicious hardware
- And the list goes on...



- Faulty Design
- Buggy Specifications
- Implementation Errors
- Side-channel leaks
- Misconfiguration
- Gullible users

Many Exploitable Vulnerabilities



In 2012, 3,436 vulnerabilities had public exploits available.
42% of total number of vulnerabilities.

Ubiquitous and Pernicious

Microsoft Security Bulletin MS15-078 – Critical

Remote code execution on *all Windows platforms* (July 20, 2015).
Caused by a buffer underflow in the Adobe Type Manager Library.

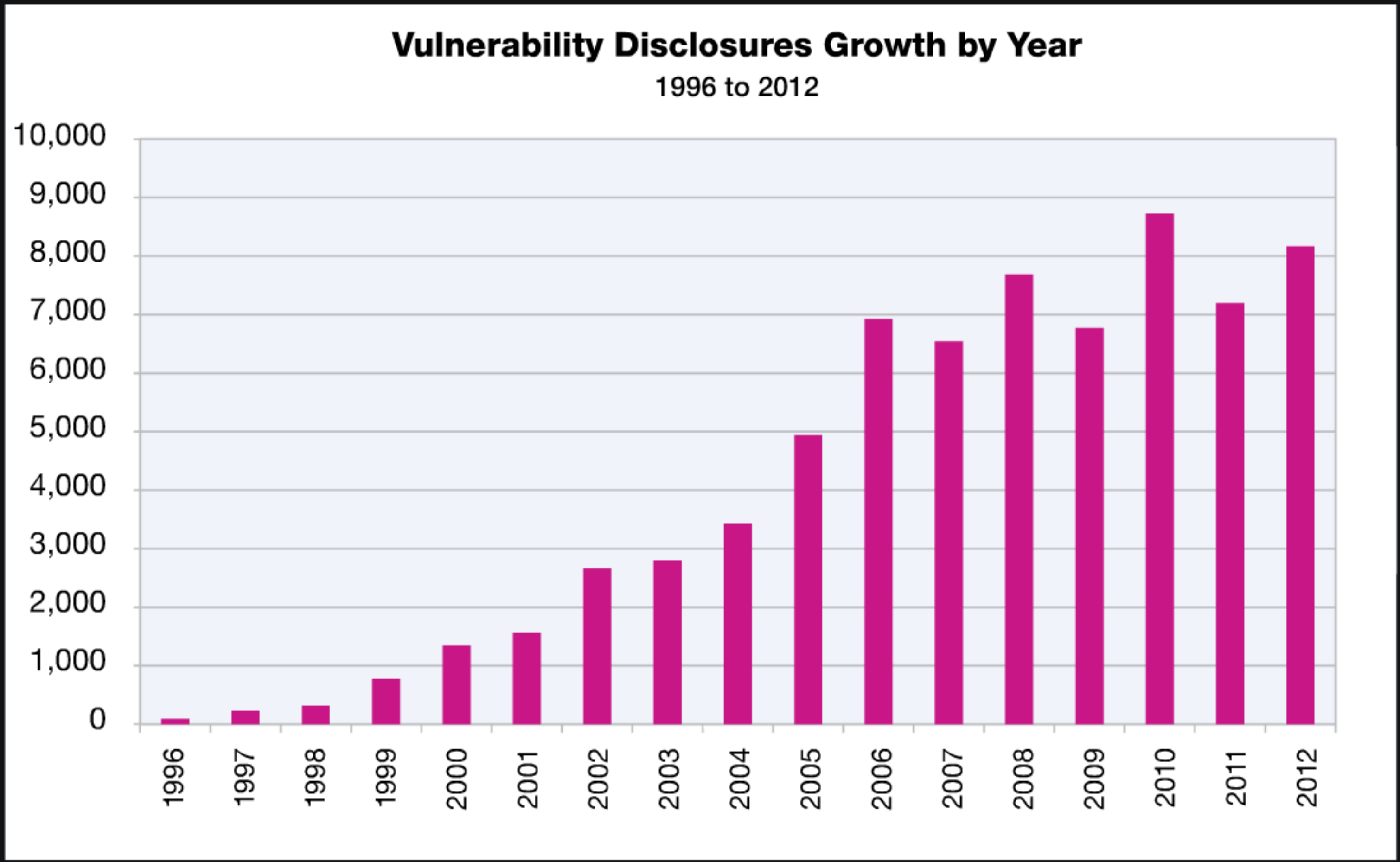
Heartbleed: Missing bounds check in OpenSSL allows theft of secret keys, user names and passwords, instant messages, emails, documents without leaving a trace.

Exploit Kits leverage software bugs

Phoenix Exploit Kit, 2010

CVE-2009-0836: Buffer Overflow
CVE-2009-0927: Missing Bounds Check
CVE-2009-1869: Integer Overflow
CVE-2010-0188: Unspecified
CVE-2010-0840: Unspecified
CVE-2010-0842: Invalid Array Index
CVE-2010-1297: Memory corruption
CVE-2010-1818: Unmarshalling bad ptr
CVE-2010-1885: Mishandling escapes
CVE-2010-2883: Buffer overflow

Enabling the masses to launch pernicious attacks at scale.



In 2012, 3,436 vulnerabilities had public exploits available.
42% of total number of vulnerabilities.

Source: IBM X-Force 2012 Trend and Risk Report

Ubiquitous and Pernicious

Microsoft Security Bulletin MS15-078 – Critical

Remote code execution on *all Windows platforms* (July 20, 2015).
Caused by a buffer underflow in the Adobe Type Manager Library.

https://technet.microsoft.com/en-us/library/security/ms15-078.aspx

http://heartbleed.com

Heartbleed: Missing bounds check in OpenSSL allows theft of secret keys, user names and passwords, instant messages, emails, documents without leaving a trace.

Exploit Kits leverage software bugs

Phoenix Exploit Kit, 2010

CVE-2009-0836: Buffer Overflow
CVE-2009-0927: Missing Bounds Check
CVE-2009-1869: Integer Overflow
CVE-2010-0188: Unspecified
CVE-2010-0840: Unspecified
CVE-2010-0842: Invalid Array Index
CVE-2010-1297: Memory corruption
CVE-2010-1818: Unmarshalling bad ptr
CVE-2010-1885: Mishandling escapes
CVE-2010-2883: Buffer overflow

Enabling the masses to launch pernicious attacks at scale.

Source: The Exploit Intelligence Project, Dan Guido

Networked computers are ***unacceptably*** insecure

Today
many things are
networked computers.

And attackers take control
with distressing ease.

Why so insecure?

- Faulty Design
- Buggy Specifications
- Implementation Errors
- Side-channel leaks
- Misconfiguration
- Gullible users
- Weak passwords
- Malicious insiders
- Physical security failures
- Reliance on 3rd party software
- Faulty/malicious hardware
- And the list goes on...

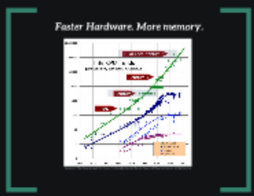
Many Exploitable Vulnerabilities

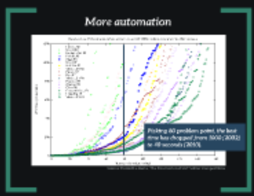
and essentially all
computers are
networked.

Hypothesis: Formal Methods can eliminate many exploitable vulnerabilities.

Been there, tried that, wasn't impressed.

Why is now a good time to revisit hypothesis?

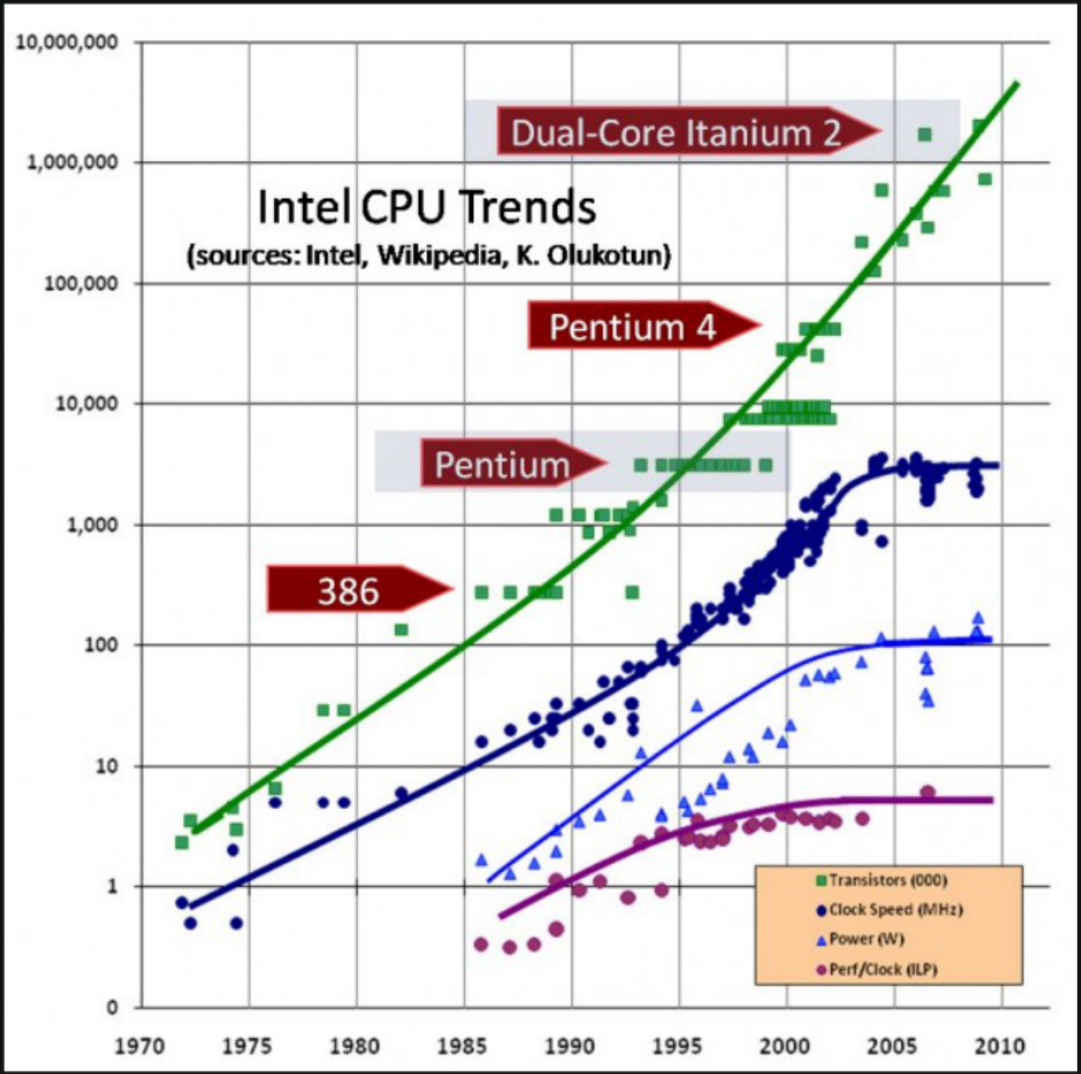




We have found that testing the code is inadequate as a method to find subtle errors in design, as the number of reachable states of the code is astronomical. So we looked for a better approach.

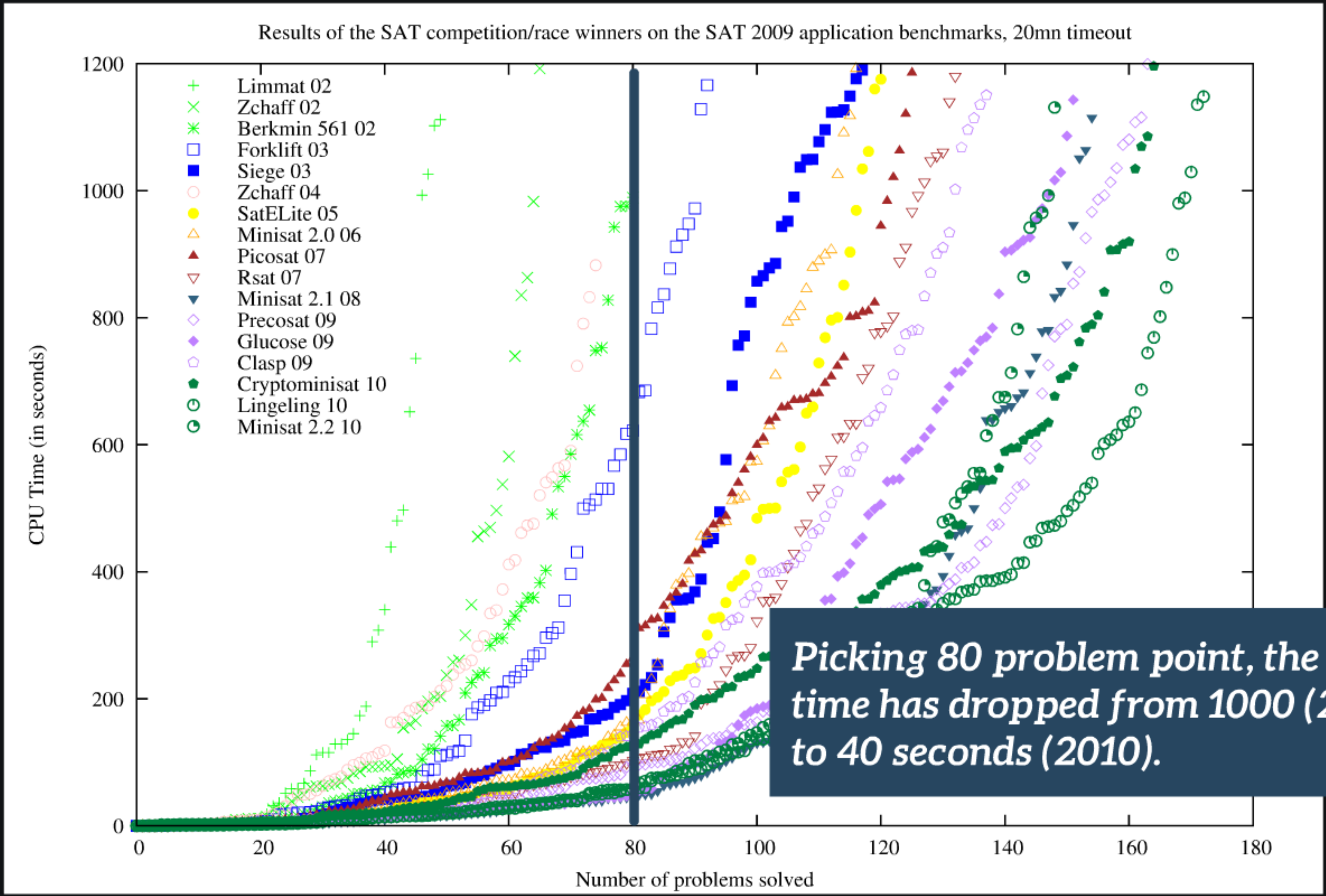
Use of Formal Methods at Amazon Web Services, 2014

Faster Hardware. More memory.



Source: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software

More automation



Picking 80 problem point, the best time has dropped from 1000 (2002) to 40 seconds (2010).

Source: Daniel Le Berre, The International SAT Solver Competitions

More infrastructure

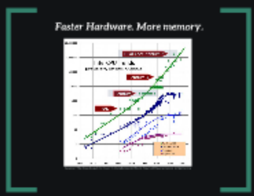
[A] significant part of the effort in existing projects was spent on the further development of verification tools, on formal models for low-level programming languages and paradigms, and on general proof libraries... *Future efforts will be able to build on these tools and reach far-ranging verification goals faster, better, and cheaper.*

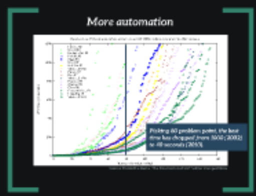
Gerwin Klein, Formal OS Verification—An Overview

Hypothesis: Formal Methods can eliminate many exploitable vulnerabilities.

Been there, tried that, wasn't impressed.

Why is now a good time to revisit hypothesis?





We have found that testing the code is inadequate as a method to find subtle errors in design, as the number of reachable states of the code is astronomical. So we looked for a better approach.

Use of Formal Methods at Amazon Web Services, 2014

Some Evidence: DARPA's HACMS Program

Using Formal Methods to produce more secure vehicles.

The Setup

Program thesis:
FM can yield vehicles less susceptible to remote cyber attack.

Threat model:
No physical access, full knowledge of system and source code.

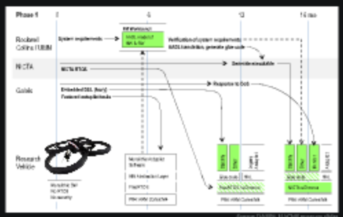
Out-of-scope:
Hardware assumed to be correct.

Experimental Platforms:
Arducopter and Boeing's Unmanned Little Bird (ULB)

Baseline Security Assessment

Red Team: Attacker could crash legitimate ground control station & hijack quadcopter in flight.

The Evolving SMACCMCopter



The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.
- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACCMCopter radio will reach the motor controller.
- Red Team:
  - *Found no security flaws in six weeks with full access to source code.*
- Penetration Testing Expert:
  The SMACCMCopter is probably "the most secure UAV on the planet."

Open source autopilot and tools available from http://smaccmpilot.org

The Setup

Program thesis:
FM can yield vehicles less susceptible to remote cyber attack.

Threat model:
No physical access, full knowledge of system and source code.

Out-of-scope:
Hardware assumed to be correct.

Experimental Platforms:
Arducopter and Boeing's Unmanned Little Bird (ULB)

Source: DYI Drones

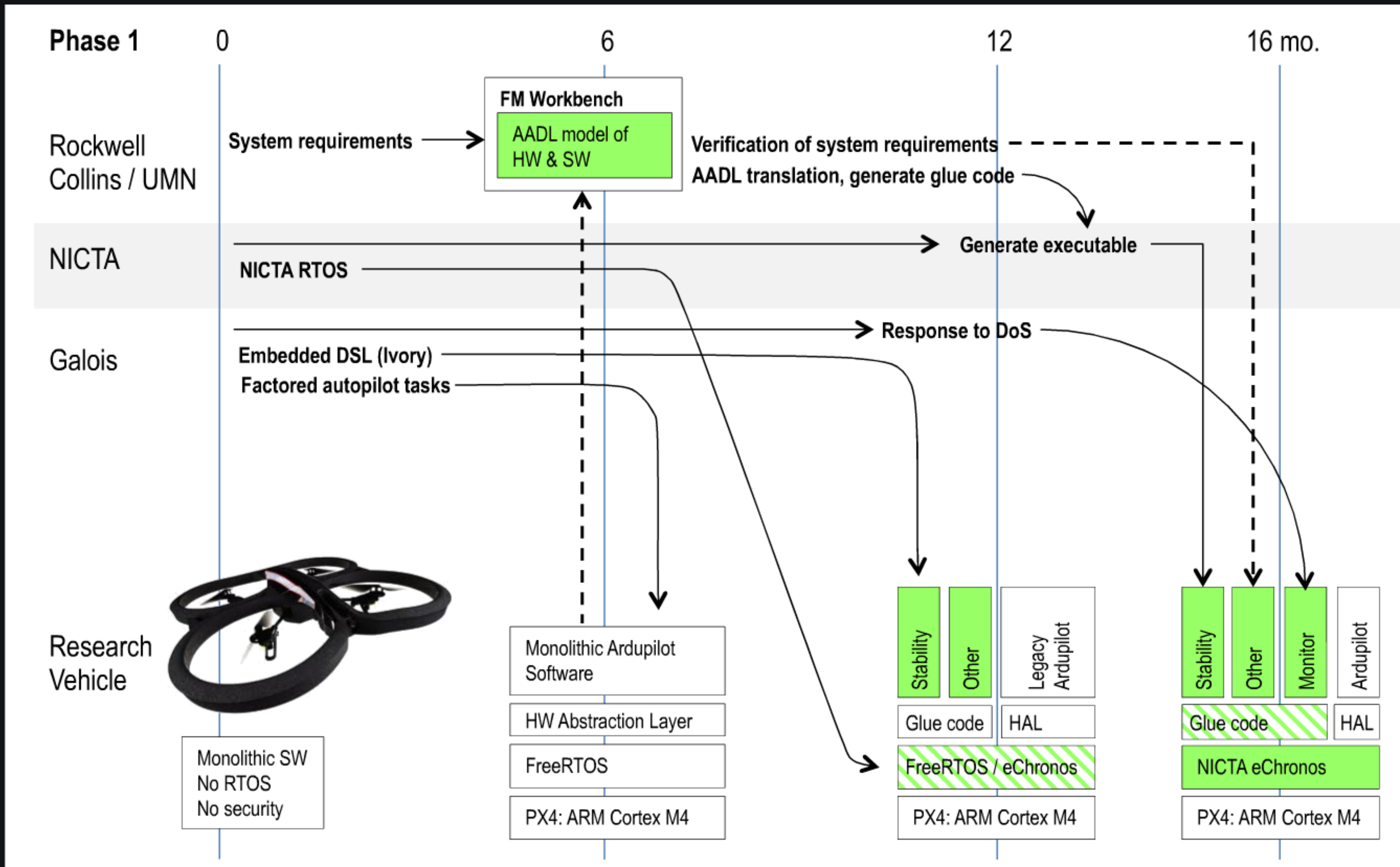
Source: Boeing

Baseline Security Assessment

Red Team: Attacker could crash legitimate ground control station & hijack quadcopter in flight.

Source: Darren Cofer, Rockwell Collins

The Evolving SMACCMCopter



Source: DARPA HACMS program slides

The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.

- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All “good” messages received by SMACCMCopter radio will reach the motor controller.

- Red Team:
  - *Found no security flaws in six weeks with full access to source code.*

- Penetration Testing Expert:
The SMACCMCopter is probably “the most secure UAV on the planet”

Open source: autopilot and tools available from http://smaccmpilot.org

Some Evidence: DARPA's HACMS Program

Using Formal Methods to produce more secure vehicles.

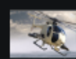
The Setup

Program thesis:
FM can yield vehicles less susceptible to remote cyber attack.

Threat model:
No physical access, full knowledge of system and source code.

Out-of-scope:
Hardware assumed to be correct.

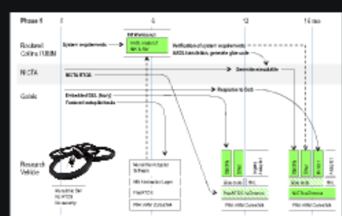
Experimental Platforms:
Arducopter and Boeing's Unmanned Little Bird (ULB)



Baseline Security Assessment

Red Team: Attacker could crash legitimate ground control station & hijack quadcopter in flight.

The Evolving SMACCMCopter



The SMACCMCopter: 18-Month Assessment

- The SMACCMCopter flies:
  - Stability control, altitude hold, directional hold, DOS detection.
  - GPS waypoint navigation 80% implemented.
- Air Team proved system-wide security properties:
  - The system is memory safe.
  - The system ignores malformed messages.
  - The system ignores non-authenticated messages.
  - All "good" messages received by SMACCMCopter radio will reach the motor controller.
- Red Team:
  - *Found no security flaws in six weeks with full access to source code.*
- Penetration Testing Expert:
  The SMACCMCopter is probably "the most secure UAV on the planet."

Open source: autopilot and tools available from http://smaccmpilot.org

Formal Methods: An Overview

What are "Formal Methods"?

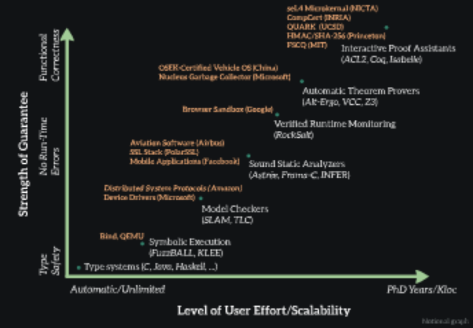
Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals ... to problems in software and hardware specification and verification.
Understanding Formal Methods, Jean-Francois Monin, 2003

"I know it when I see it" — *Justice Potter Stewart*

Characteristics:

- Based on math
- Machine-checkable
- Capable of proving properties of code and models
  - But, read the fine print!
  - Assumptions may be unreasonable.
  - Guarantees may be too weak.

Formal Method Based Tools



Survey Results on Using Formal Methods

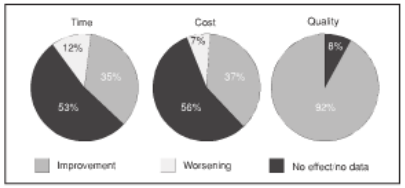


Fig. 6. Did the use of formal techniques have an effect on time, cost, and quality?

Source: Formal Methods: Practice and Experience, ACM Computing Surveys, October 2009

What are "Formal Methods"?

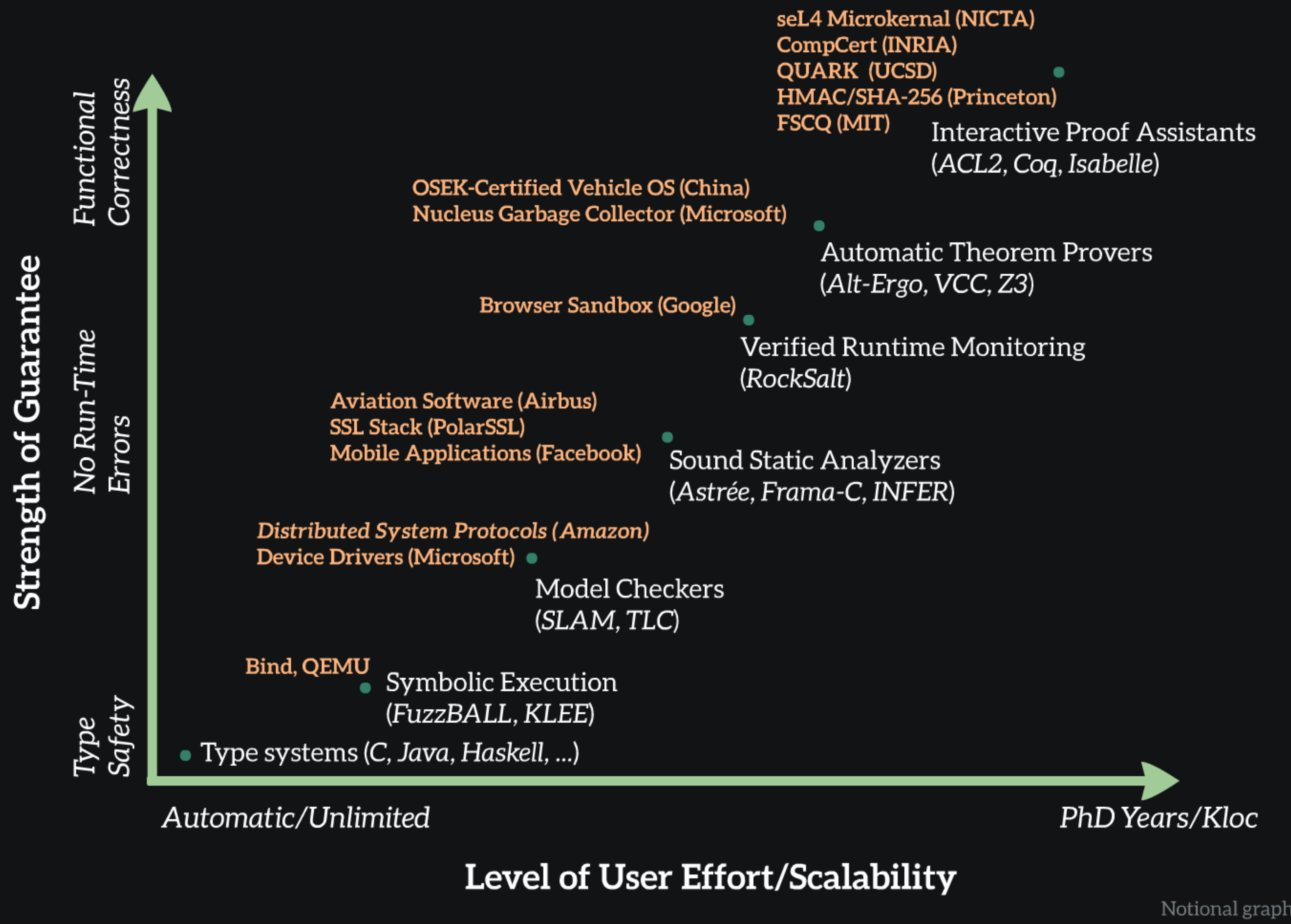
Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals ... to problems in software and hardware specification and verification.
Understanding Formal Methods, Jean-Francois Monin, 2003

"I know it when I see it" — *Justice Potter Stewart*

Characteristics:

- Based on math
- Machine-checkable
- Capable of proving properties of code and models
  - But, read the fine print!
  - Assumptions may be unreasonable.
  - Guarantees may be too weak.

Formal Method Based Tools



Survey Results on Using Formal Methods

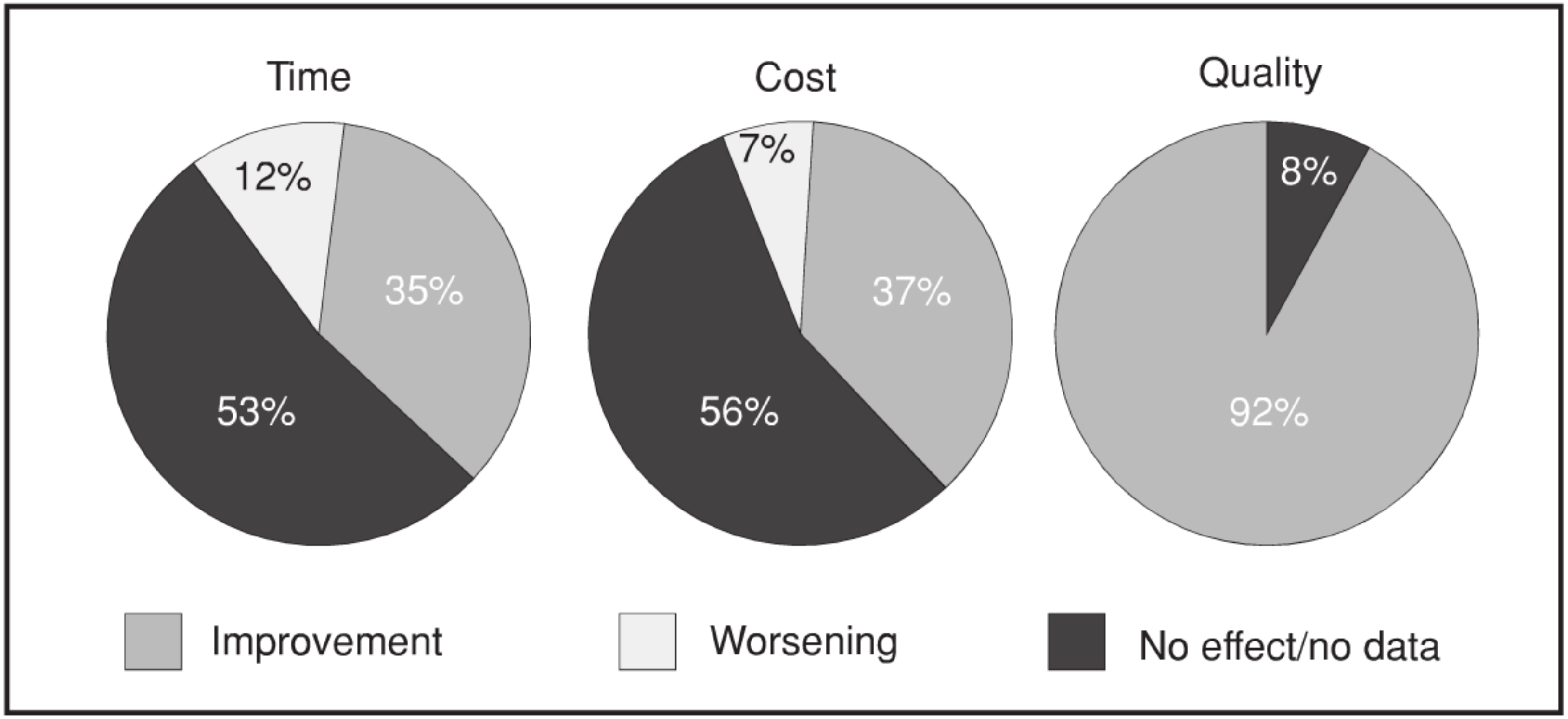


Fig. 6. Did the use of formal techniques have an effect on time, cost, and quality?

Source: Formal Methods: Practice and Experience, ACM Computing Surveys, October 2009

Formal Methods: An Overview

What are "Formal Methods"?

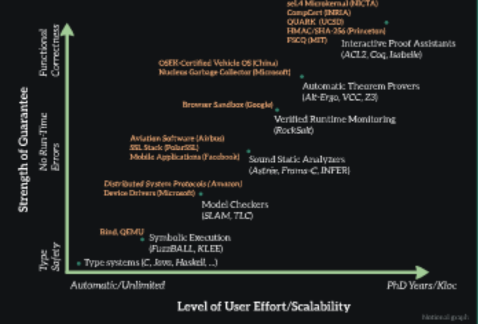
Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals ... to problems in software and hardware specification and verification.
Understanding Formal Methods, Jean-Francois Monin, 2003

"I know it when I see it" — *Justice Potter Stewart*

Characteristics:

- Based on math
- Machine-checkable
- Capable of proving properties of code and models
  - But, read the fine print!
  - Assumptions may be unreasonable.
  - Guarantees may be too weak.

Formal Method Based Tools



Survey Results on Using Formal Methods

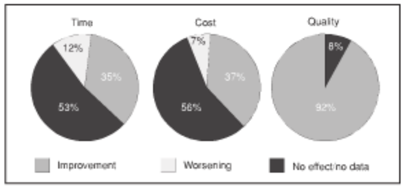


Fig. 6. Did the use of formal techniques have an effect on time, cost, and quality?

Source: Formal Methods: Practice and Experience, ACM Computing Surveys, October 2009

What software is worth verifying?

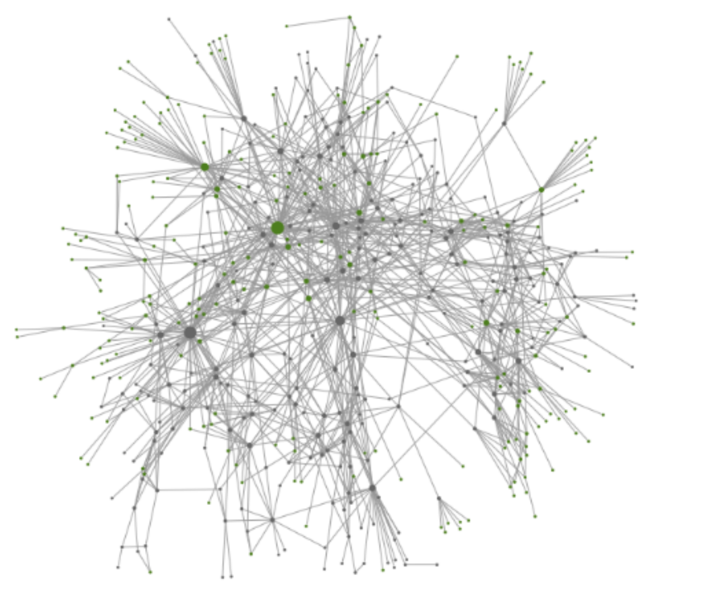
- **Separation Kernel:** seL4 [SOSP 2009, ToCS 2014] 
- **Hypervisor:** mCertiKOS [POPL 2015]
- **RTOSes:** eChronos [echronos.org]
  ORIENTIAS [ICECCS 2012]
- **C Compiler:** CompCert [POPL 2006] 
- **File Systems:** FSCQ [SOSP 2015], BilbyFS [OpSysRev 2014]
- **Web Browser:** QUARK [USENIX 2012]
- **Browser Sandbox:** RockSalt [PLDI 2012]
- **Crypto Algorithms:** SHA-256, HMAC [USENIX 2015]
  AES-128, SHA-384, ECDSA(NIST P-384) [HILT 2013]
- **Garbage Collector:** Nucleus [PLDI 2010]

All verified to be functionally correct!

seL4 Microkernel

- General purpose, operating system microkernel.
- Implemented and proven correct in Isabelle/HOL.
- Size: 10K LoC, 480K LoP
- Time to build: 13 (8) person years
- Speed: 206 vs 227 cycles in 1-way IPC fastpath
- Machine-checked theorems include
  - Access-control enforcement
  - Non-interference
  - Compilation to binary
  - IPC fast-path correctness

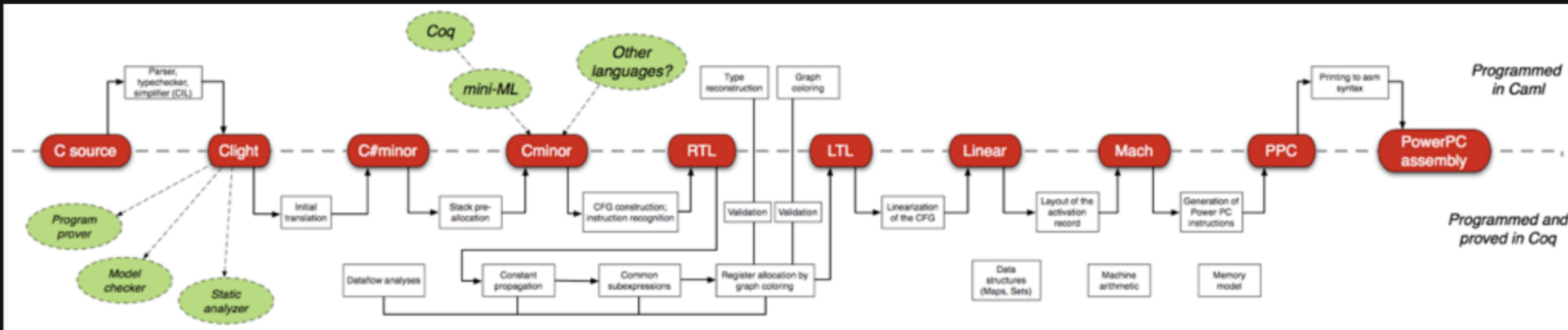
Available open source.



Call graph of the seL4 microkernel. Vertices represent functions, and edges invocations.

Comprehensive Formal Verification of an OS Microkernel,
ACM Transactions on Computer Systems, February 2014.

CompCert Verifying C Compiler



Xavier Leroy, Formal certification of a compiler back-end, POPL 2006

- Subset of C used by aviation industry
- Implemented and proven correct in Coq.
- Size: 42K LoC+P
- Time to build: 3 person years
- Speed: 2x speed of gcc -O0,
  7% slower than gcc -O1,
  12% slower than gcc -O2
- Poised to become the compiler for Airbus software.

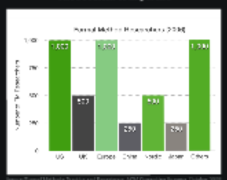
What software is worth verifying?

- **Separation Kernel:** seL4 [SOSP 2009, ToCS 2014] 
- **Hypervisor:** mCertiKOS [POPL 2015]
- **RTOSes:** eChronos [echronos.org]
  ORIENTIAS [ICECCS 2012]
- **C Compiler:** CompCert [POPL 2006] 
- **File Systems:** FSCQ [SOSP 2015], BilbyFS [OpSysRev 2014]
- **Web Browser:** QUARK [USENIX 2012]
- **Browser Sandbox:** RockSalt [PLDI 2012]
- **Crypto Algorithms:** SHA-256, HMAC [USENIX 2015]
  AES-128, SHA-384, ECDSA(NIST P-384) [HILT 2013]
- **Garbage Collector:** Nucleus [PLDI 2010]

All verified to be functionally correct!

Impediments to Using Formal Methods

The Problem of Expertise



The Required Level of Effort

Significant overhead in terms of lines of code/proof, but level of effort is becoming a reasonable investment for certain kinds of software.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  10K LoC, 480K LoP; 13(?) person years
- CompCert Verifying C compiler [POPL 2006]
  42K LoC+P; 3 person years
- FSCQ File System [SOSP 2015]
  24K LoC+P; <5 person years
- certiKOS Hypervisor [POPL 2015]
  3K LoC, 18.5K LoP; 1 person year
- SHA-256/HMAC [USENIX 2015]
  407 LoC, 14.4K LoP; not reported
- Rocksalt Sandbox [PLDI 2012]
  100 LoC, 10K LoP; <2 person years
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  6K LoP+C; 0.75 person years
- QUARK Web Browser [USENIX 2012]
  5.5K LoP+C; 0.5 person years

Performance Consequences

Verified code is not intrinsically slower, but verifying faster code can be more time consuming.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  206 vs. 227 cycles in 1-way IPC fastpath
- CompCert Verifying C compiler [POPL 2006]
  2x speed of gcc -O0, 7% slower than gcc -O1, 12% slower than gcc -O2
- FSCQ File System [SOSP 2015]
  Performance roughly 80% of xv6 file system.
- certiKOS Hypervisor [POPL 2015]
  <2x slowdown on most lmbench benchmarks
- SHA-256/HMAC [USENIX 2015]
  Performance equal to OpenSSL 0.9.1c (March 1999)
- Rocksalt Sandbox [PLDI 2012]
  1M instructions/second; faster than Google's checker
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  "competitive performance on macro-benchmarks when compared to native garbage collection."
- QUARK Web Browser [USENIX 2012]
  24% overhead wrt WebKit baseline on the top 10 Alexa Web sites

Keeping Up

Time to produce a proof can be serious impediment to adoption.

(Many of the artifacts we've considered lack features.)

Facebook's INFER sound static analyzer processes millions of lines of code and thousands of diffs per day.
- 4 hours for full Android/iOS code base
- <10 minutes on average for single diffs

But, proves only the absence of null pointer exceptions and resource leaks.

The Fine Print

All proofs come with assumptions.
Violate those assumptions and all bets are off.

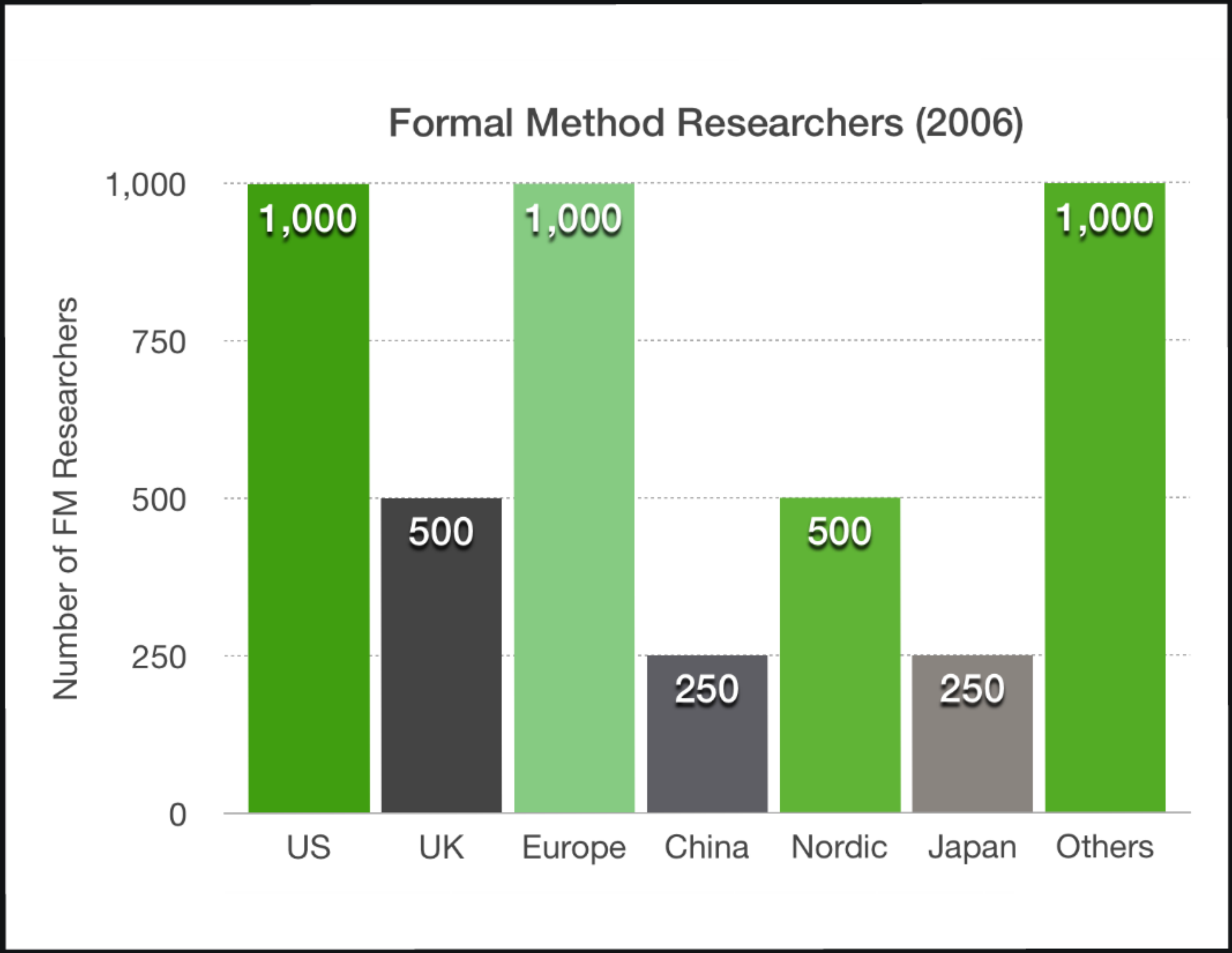
John Regehr et al found 325 bugs in various C compilers, including CompCert [PLDI 2011].

CompCert bugs were in:
- the unverified front end (subsequently fixed & verified)
- a hardware model

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. ... *CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.* This is not for lack of trying: we have devoted about six CPU-years to the task."

Source: Finding and Understanding Bugs in C Compilers, PLDI 2011

The Problem of Expertise



Source: Formal Methods: Practice and Experience, ACM Computing Surveys, October 2009

The Required Level of Effort

Significant overhead in terms of lines of code/proof, but level of effort is becoming a reasonable investment for certain kinds of software.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  10K LoC, 480K LoP; 13(8) person years
- CompCert Verifying C compiler [POPL 2006]
  42K LoC+P; 3 person years
- FSCQ File System [SOSP 2015]
  24K LoC+P; <5 person years
- certiKOS Hypervisor [POPL 2015]
  3K LoC, 18.5K LoP; 1 person year
- SHA-256/HMAC [USENIX 2015]
  407 LoC, 14.4K LoP; not reported
- Rocksalt Sandbox [PLDI 2012]
  100 LoC, 10K LoP; <2 person years
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  6K LoP+C; 0.75 person years
- QUARK Web Browser [USENIX 2012]
  5.5K LoP+C; 0.5 person years

Performance Consequences

Verified code is not intrinsically slower, but verifying faster code can be more time consuming.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  206 vs 227 cycles in 1-way IPC fastpath
- CompCert Verifying C compiler [POPL 2006]
  2x speed of gcc-O0, 7% slower than gcc -O1, 12% slower than gcc -O2
- FSCQ File System [SOSP 2015]
  Performance roughly 80% of xv6 file system.
- certiKOS Hypervisor [POPL 2015]
  <2x slowdown on most lmbench benchmarks
- SHA-256/HMAC [USENIX 2015]
  Performance equal to OpenSSL 0.9.1c (March 1999)
- Rocksalt Sandbox [PLDI 2012]
  1M instructions/second; faster than Google's checker
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  "competitive performance on macro-benchmarks when compared to native garbage collection."
- QUARK Web Browser [USENIX 2012]
  24% overhead wrt WebKit baseline on the top 10 Alexa Web sites

Keeping Up

Time to produce a proof can be serious impediment to adoption.

(Many of the artifacts we've considered lack features.)

Facebook's INFER sound static analyzer processes millions of lines of code and thousands of diffs per day.

- 4 hours for full Android/iOS code base
- <10 minutes on average for single diffs

But, proves only the absence of null pointer exceptions and resource leaks.

Source: Moving Fast with Software Verification, NASA Formal Method Symposium, 2015

The Fine Print

All proofs come with assumptions.
Violate those assumptions and all bets are off.

John Regehr et al found 325 bugs in various C compilers, *including CompCert* [PLDI 2011].

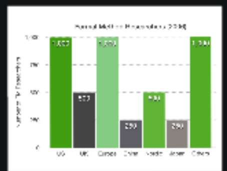
CompCert bugs were in:
- the unverified front end (subsequently fixed & verified)
- a hardware model

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. ... *CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.* This is not for lack of trying: we have devoted about six CPU-years to the task."

Source: Finding and Understanding Bugs in C Compilers, PLDI 2011

Impediments to Using Formal Methods

The Problem of Expertise



The Required Level of Effort

Significant overhead in terms of lines of code/proof, but level of effort is becoming a reasonable investment for certain kinds of software.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  10K LoC, 480K LoP; 13(?) person years
- CompCert Verifying C compiler [POPL 2006]
  42K LoC+P; 3 person years
- FSCQ File System [SOSP 2015]
  24K LoC+P; <5 person years
- certiKOS Hypervisor [POPL 2015]
  3K LoC, 18.5K LoP; 1 person year
- SHA-256/HMAC [USENIX 2015]
  407 LoC, 14.4K LoP; not reported
- Rocksalt Sandbox [PLDI 2012]
  100 LoC, 10K LoP; <2 person years
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  6K LoP+C; 0.75 person years
- QUARK Web Browser [USENIX 2012]
  5.5K LoP+C; 0.5 person years

Performance Consequences

Verified code is not intrinsically slower, but verifying faster code can be more time consuming.

- seL4 Separation Kernel [SOSP 2009; ToCS 2014]
  206 vs 227 cycles in 1-way IPC fastpath
- CompCert Verifying C compiler [POPL 2006]
  2x speed of gcc -O0, 7% slower than gcc -O1, 12% slower than gcc -O2
- FSCQ File System [SOSP 2015]
  Performance roughly 80% of xv6 file system.
- certiKOS Hypervisor [POPL 2015]
  <2x slowdown on most lmbench benchmarks
- SHA-256/HMAC [USENIX 2015]
  Performance equal to OpenSSL 0.9.1c (March 1999)
- Rocksalt Sandbox [PLDI 2012]
  1M instructions/second, faster than Google's checker
- Nucleus Allocator/Garbage Collector [PLDI 2010]
  "competitive performance on macro-benchmarks when compared to native garbage collection."
- QUARK Web Browser [USENIX 2012]
  24% overhead wrt WebKit baseline on the top 10 Alexa Web sites

Keeping Up

Time to produce a proof can be serious impediment to adoption.

(Many of the artifacts we've considered lack features.)

Facebook's INFER sound static analyzer processes millions of lines of code and thousands of diffs per day.
- 4 hours for full Android/iOS code base
- <10 minutes on average for single diffs

But, proves only the absence of null pointer exceptions and resource leaks.

The Fine Print

All proofs come with assumptions.
Violate those assumptions and all bets are off.

John Regehr et al found 325 bugs in various C compilers, *including CompCert* [PLDI 2011].

CompCert bugs were in:
- the unverified front end (subsequently fixed & verified)
- a hardware model

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. ... *CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors.* This is not for lack of trying: we have devoted about six CPU-years to the task."

Source: Finding and Understanding Bugs in C Compilers, PLDI 2011

Taking Stock

Formal Methods: Not just for implementation bugs!

Formal methods can help with other security challenges as well:

- Faulty design
  Ex: Amazon Web Services use of TLA+
- Buggy specifications
- Ex: Rockwell Collins detecting unencrypted comm channel
- Side-channel information leaks
  Ex: NICTA analysis of seL4
- Dependence on 3rd party software

Lessons Learned

- Don't verify existing code artifacts
  seL4, CompCert, FSCQ, certiKOS, Rocksalt, Nucleus, QUARK
- Use static analyzers to eliminate obvious bugs before starting formal verification.
- Don't verify all code:
  Secure essential infrastructure & contain the rest.
  QUARK, Verve/Nucleus, Rocksalt
- Use DSLs to generate code and correctness proofs
  Rocksalt, Ivory/Tower, SpiralGen, BilbyS
- Composition enables scaling
  Facebook INFER; UC-KLEE, Rockwell Collins Workbench; mCertiKOS
- Automation is essential:
  Tactic libraries and SMT/SAT solver integration
  FSCQ, Nucleus, mCertiKOS

On-Going Research & Challenges

- Producing and validating models of real systems
  x86, LLVM, Linux, Browser APIs, POSIX interfaces
- Increasing automation
- Scaling
- Proof engineering
- Integrating with normal development processes
- Getting buy-in/adoption & training
  "Exhaustively testable pseudo-code"
  [*Using Formal Methods at Amazon Web Services*]
- Handling concurrency

Formal Methods:
Not just for implementation bugs!

Formal methods can help with other security challenges as well:

- Faulty design
  Ex: Amazon Web Services use of TLA+
- Buggy specifications
- Ex: Rockwell Collins detecting unencrypted comm channel
- Side-channel information leaks
  Ex: NICTA analysis of seL4
- Dependence on 3rd party software

Lessons Learned

- Don't verify existing code artifacts
  seL4, CompCert, FSCQ, certiKOS, Rocksalt, Nucleus, QUARK
- Use static analyzers to eliminate obvious bugs before starting formal verification.
- Don't verify all code:
  Secure essential infrastructure & contain the rest.
  QUARK, Verve/Nucleus, Rocksalt
- Use DSLs to generate code and correctness proofs
  Rocksalt, Ivory/Tower, SpiralGen, BilbyS
- Composition enables scaling
  Facebook INFER; UC-KLEE, Rockwell Collins Workbench; mCertiKOS
- Automation is essential:
  Tactic libraries and SMT/SAT solver integration
  FSCQ, Nucleus, mCertiKOS

On-Going Research & Challenges

- Producing and validating models of real systems
  x86, LLVM, Linux, Browser APIs, POSIX interfaces
- Increasing automation
- Scaling
- Proof engineering
- Integrating with normal development processes
- Getting buy-in/adoption & training
  "Exhaustively testable pseudo-code"
  [*Using Formal Methods at Amazon Web Services*]
- Handling concurrency

Taking Stock

Formal Methods: Not just for implementation bugs!

Formal methods can help with other security challenges as well:

- Faulty design
  Ex: Amazon Web Services use of TLA+
- Buggy specifications
- Ex: Rockwell Collins detecting unencrypted comm channel
- Side-channel information leaks
  Ex: NICTA analysis of seL4
- Dependence on 3rd party software

Lessons Learned

- Don't verify existing code artifacts
  seL4, CompCert, FSCQ, certiKOS, Rocksalt, Nucleus, QUARK
- Use static analyzers to eliminate obvious bugs before starting formal verification.
- Don't verify all code:
  Secure essential infrastructure & contain the rest.
  QUARK, Verve/Nucleus, Rocksalt
- Use DSLs to generate code and correctness proofs
  Rocksalt, Ivory/Tower, SpiralGen, BilbyS
- Composition enables scaling
  Facebook INFER; UC-KLEE, Rockwell Collins Workbench; mCertiKOS
- Automation is essential:
  Tactic libraries and SMT/SAT solver integration
  FSCQ, Nucleus, mCertiKOS

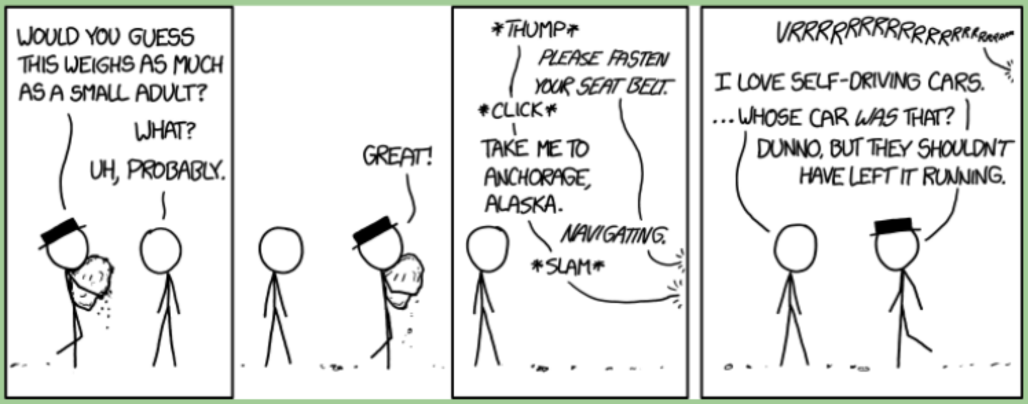
On-Going Research & Challenges

- Producing and validating models of real systems
  x86, LLVM, Linux, Browser APIs, POSIX interfaces
- Increasing automation
- Scaling
- Proof engineering
- Integrating with normal development processes
- Getting buy-in/adoption & training
  "Exhaustively testable pseudo-code"
  [*Using Formal Methods at Amazon Web Services*]
- Handling concurrency

Within Reach?

- Critical parts of critical systems are built out of verified components & their composition is verified.
- Buggy software is no longer an easy attack vector.

- Black Hat is reduced to non-scalable attack vectors:

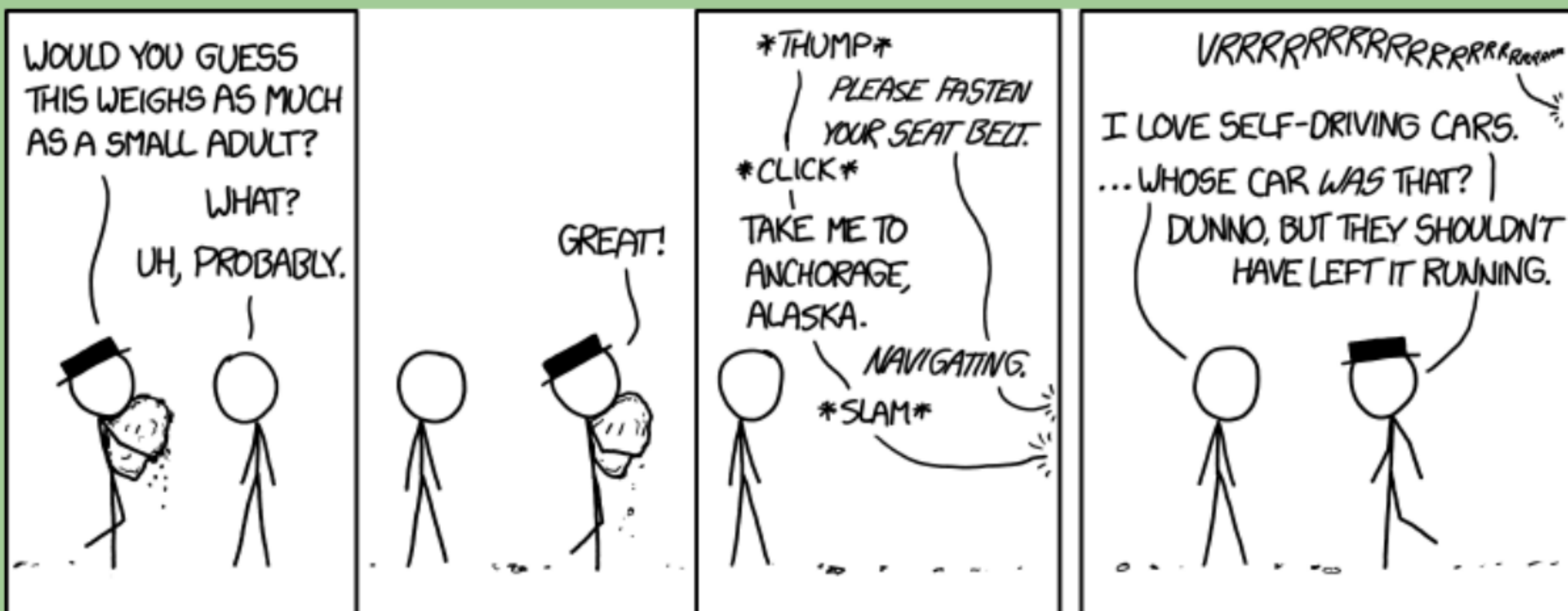


http://xkcd.com/1559/

verified components & their composition is verified.

- Buggy software is no longer an easy attack vector.

- Black Hat is reduced to non-scalable attack vectors:



http://xkcd.com/1559/

QUESTIONS?

Thanks to:

Andrew Appel
Princeton
Adam Chlipala
MIT
Darren Cofer
Rockwell Collins
Drew Dean
SRI
Dan Guido
Trail of Bits
Joe Hendrix
Galois

John Launchbury
DARPA
Gerwin Klein
NICTA
Gary McGraw
Cigital
Greg Morrisett
Cornell
Aaron Tomb
Galois
Mike Walker
DARPA