Trustworthy Software Systems

Greg Morrisett
Cutting Professor of Computer Science
School of Engineering & Applied Sciences
Harvard University

HARVARD
School of Engineering
and Applied Sciences

Little about me…

- Research & Teaching
  - Compilers, Languages, Formal Methods
  - Software Security
  - Harvard Center for Research on Computation & Society
- Number of security-oriented advisory boards
  - Microsoft Trustworthy Computing Board (& MSR TAB)
  - Intel-Berkeley SCRUB Lab
  - Fortify (bought by HP)
  - DARPA ISAT
  - National Academy Study on "Science of Cybersecurity"

All too familiar headlines…

SECURITY security, encryption, heartbleed

Devastating 'Heartb
was unknown befor
disclosure, study fir

September 26, 2014

19 million Windows PCs still vulnerable to Stuxnet zero-day

In Cyberattack on Saudi Firm, U
Back

Security researcher says many of his iOS 'backdoor' vulnerabilities are fixed in iOS 8 GM, but not all

**The Stuxnet Attack On Iran's N
Was 'Far More Dangerous' Tha
Thought**

MICHAEL B KELLEY

NOV. 20, 2013, 12:58 PM 37,528 11

ENTERPRISE

The Internet Is Broken, and Shellshock Is Just the Start of Our Woes

BY ROBERT MCMILLAN 09.29.14 | 6:30 AM | PERMALINK

From DARPA's Cyber Analytic Framework…

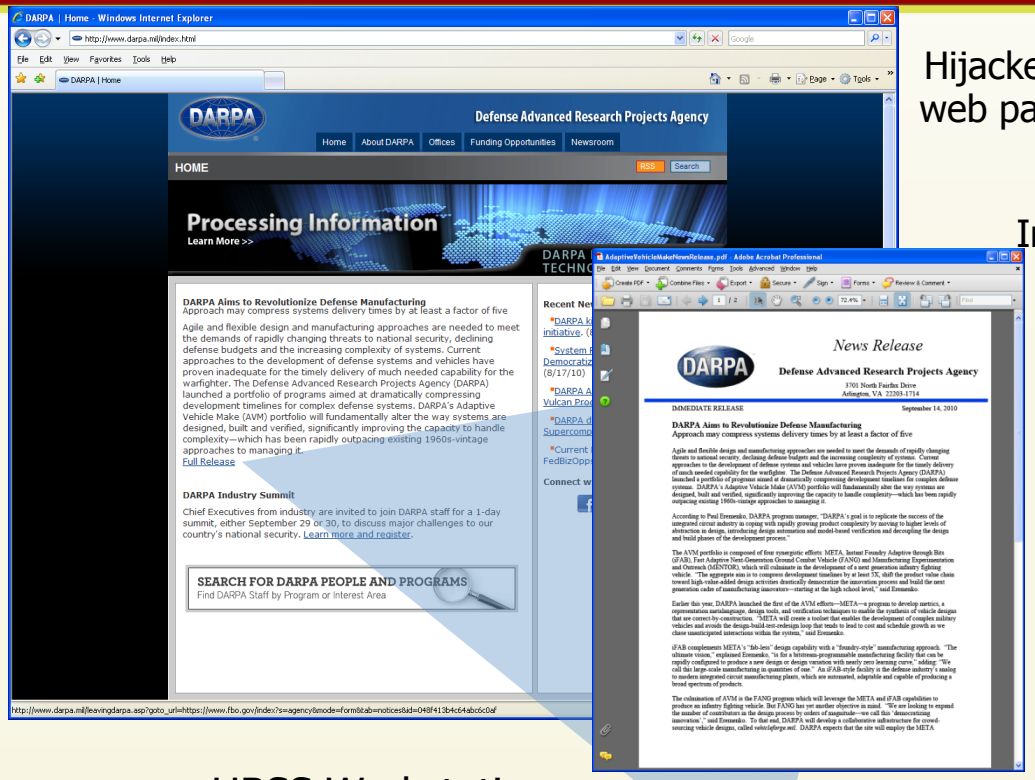
Attackers penetrate the architecture easily…

Goal

- Demonstrate asymmetric ease of exploitation of DoD computer versus efforts to defend.

Result

- Multiple remote compromises of fully security compliant and patched HBSS‡ computer within days:
  - 2 remote exploits
  - 25+ local privilege escalation exploits
  - Undetected by defenses



Hijacked web page

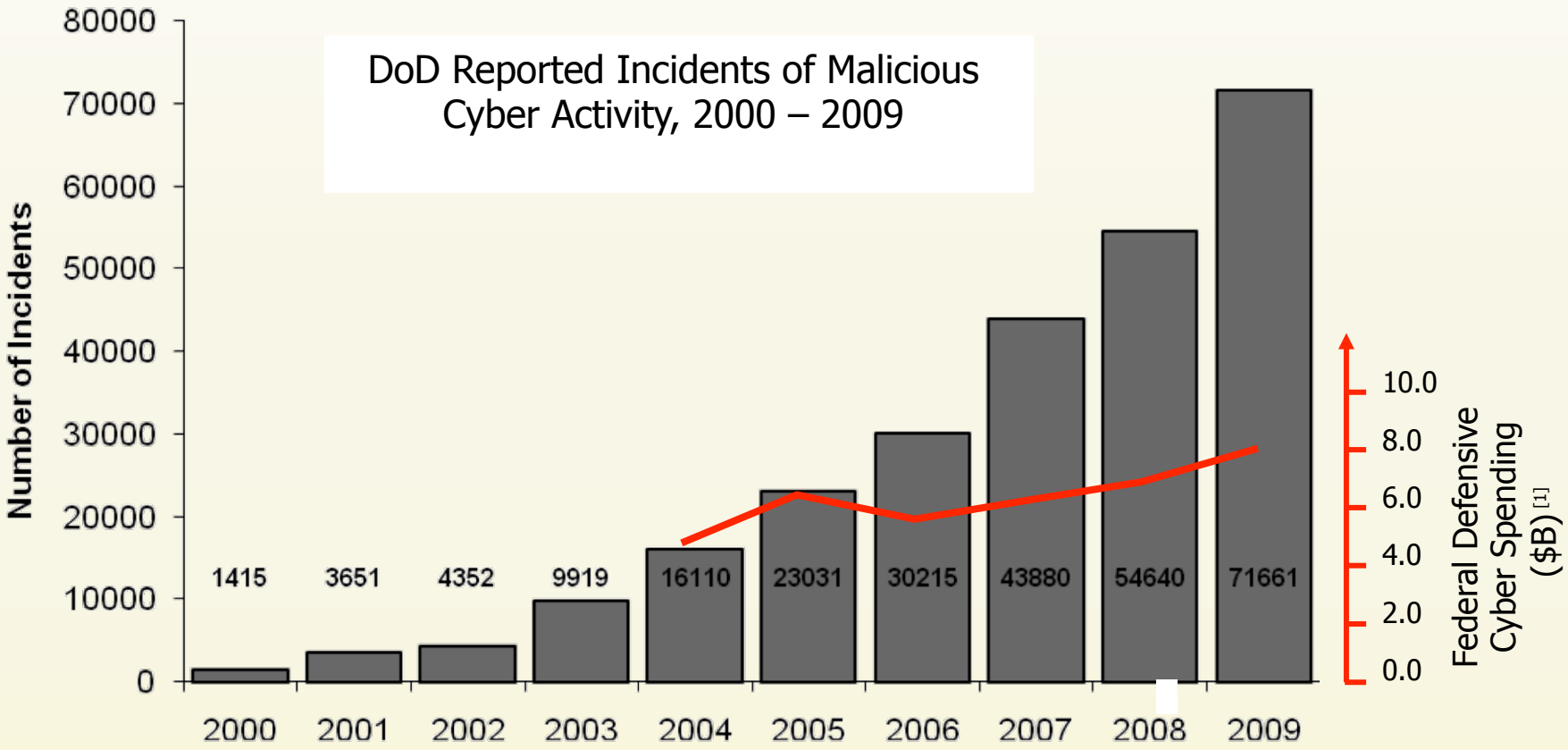
Infected .pdf document

HBSS Workstation Penetration Demonstration

Total Effort: 2 people, 3 days, Total cost = $18K

HBSS Costs: Millions of dollars a year for software and licenses alone (not including man hours)

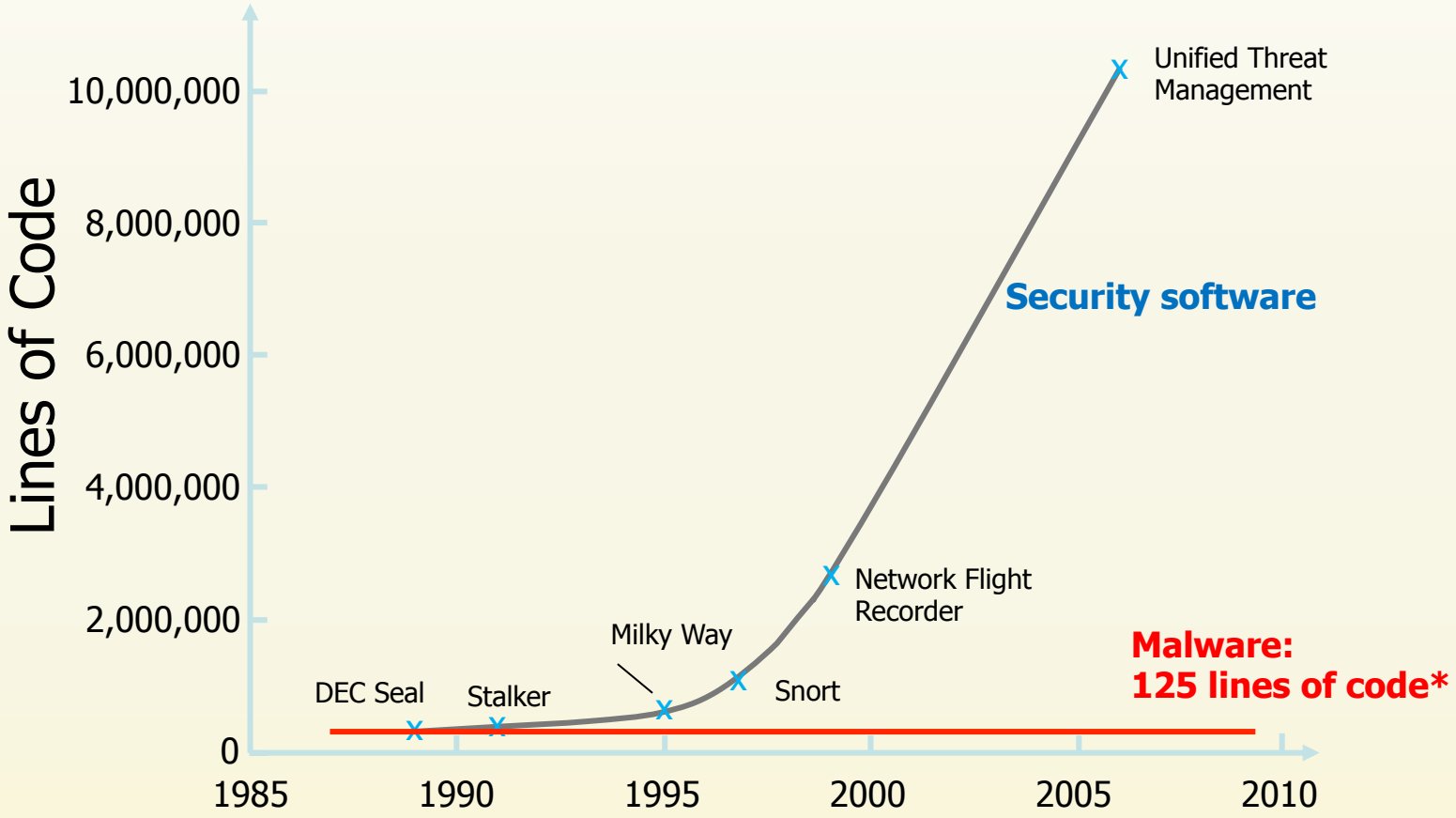
Ground truth…



2010 REPORT TO CONGRESS *of the* U.S.-CHINA ECONOMIC AND SECURITY REVIEW COMMISSION

[1] INPUT reports 2004 – 2009

We are divergent with the threat…



* Malware lines of code averaged over 9,000 samples

Many Things Need to be Fixed

- User interfaces (and users)
- Underlying Architecture
- Underlying Protocols
- Configuration & Operation tools

But one huge issue dominates right now:

- The code we depend upon is full of bugs.

What's going wrong?

- Development processes are ineffective.
  - Human code review doesn't work.
- Certification processes are ineffective.
  - Based on who authored, not the code itself.
- Current automated defenses are worse than ineffective.
  - Based on *syntax* or *provenance*, not semantics.
  - Introduce new classes of vulnerabilities.

Additional security layers often create vulnerabilities…

Current vulnerability watchlist

Vulnerability Title	Fix Avail?	Date Added
XXXXXXXXXXX XXXXXXXXXXX Local Privilege Escalation Vulnerability	No	8/25/2010
XXXXXXXXXXX XXXXXXXXXXX Denial of Service Vulnerability	Yes	8/24/2010
XXXXXXXXXXX XXXXXXXXXXX Buffer Overflow Vulnerability	No	8/20/2010
XXXXXXXXXXX XXXXXXXXXXX Sanitization Bypass Weakness	No	8/18/2010
XXXXXXXXXXX XXXXXXXXXXX Security Bypass Vulnerability	No	8/17/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Security Vulnerabilities	Yes	8/16/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Code Execution Vulnerability	No	8/16/2010
XXXXXXXXXXX XXXXXXXXXXX Use-After-Free Memory Corruption Vulnerability	No	8/12/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Code Execution Vulnerability	No	8/10/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Buffer Overflow Vulnerabilities	No	8
XXXXXXXXXXX XXXXXXXXXXX Stack Buffer Overflow Vulnerability	Yes	8
XXXXXXXXXXX XXXXXXXXXXX Security-Bypass Vulnerability	No	8
XXXXXXXXXXX XXXXXXXXXXX Multiple Security Vulnerabilities	No	8
XXXXXXXXXXX XXXXXXXXXXX Buffer Overflow Vulnerability	No	7/29/2010
XXXXXXXXXXX XXXXXXXXXXX Remote Privilege Escalation Vulnerability	No	7/28/2010
XXXXXXXXXXX XXXXXXXXXXX Cross Site Request Forgery Vulnerability	No	7/26/2010
XXXXXXXXXXX XXXXXXXXXXX Multiple Denial Of Service Vulnerabilities	No	7/22/2010

6 of the vulnerabilities are in security software

Color Code Key: Vendor Replied – Fix in development | Awaiting Vendor Reply/Confirmation | Awaiting CC/S/A use validation

Market Failures

This problem won't be solved by startups:

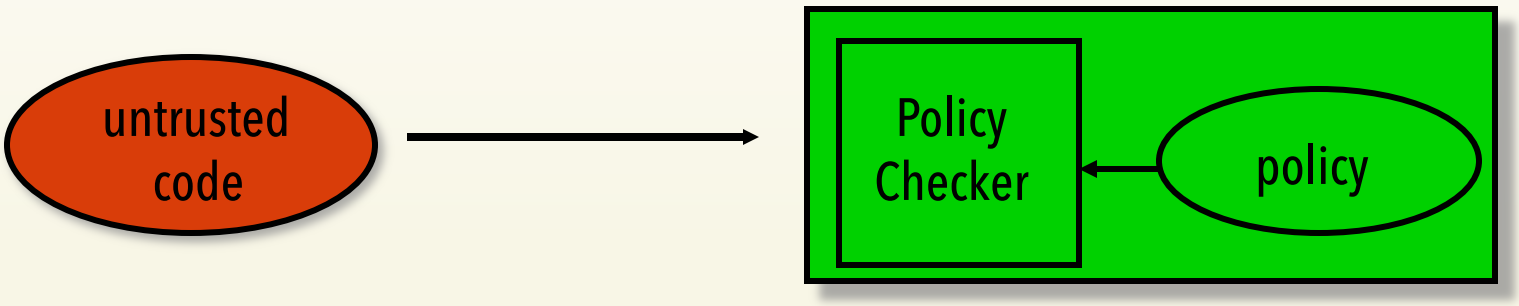
- Developers are stingy.
- Developers make money/fame by adding features, not by doing security audits or fixes.

Contrast with attackers.

- They make money by doing careful audits…

So how do we dig ourselves out of this mess?

Ideal Architecture:



Policies capture *behavior.*

The checker automatically rules out any code that will violate the policy.

The checker is small, simple, trustworthy, and automatic.

Unfortunately

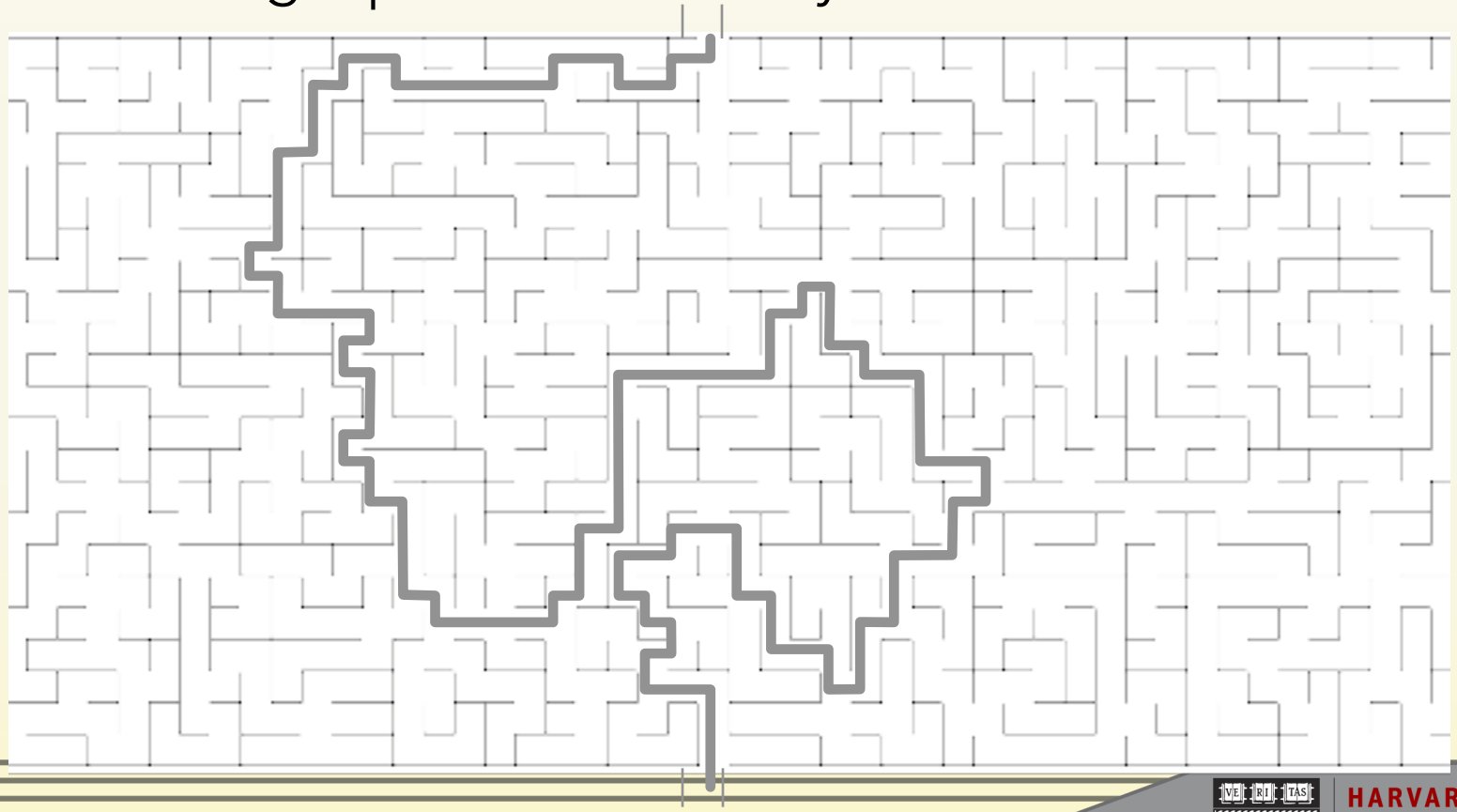
- Even simple policies are undecidable.
  - e.g., Does the code have a buffer overflow?
  - So any checker is either incomplete or unsound.
    - Incomplete: rules out programs that meet the policy
    - Unsound: allows a program that fails to meet the policy

- Analyzing machine code is *hard*.
  - It's hard enough to analyze real source code for simple policies.
  - Any machine-level analysis requires a big, complicated checker.
  - So how can we trust that it's doing its job correctly?

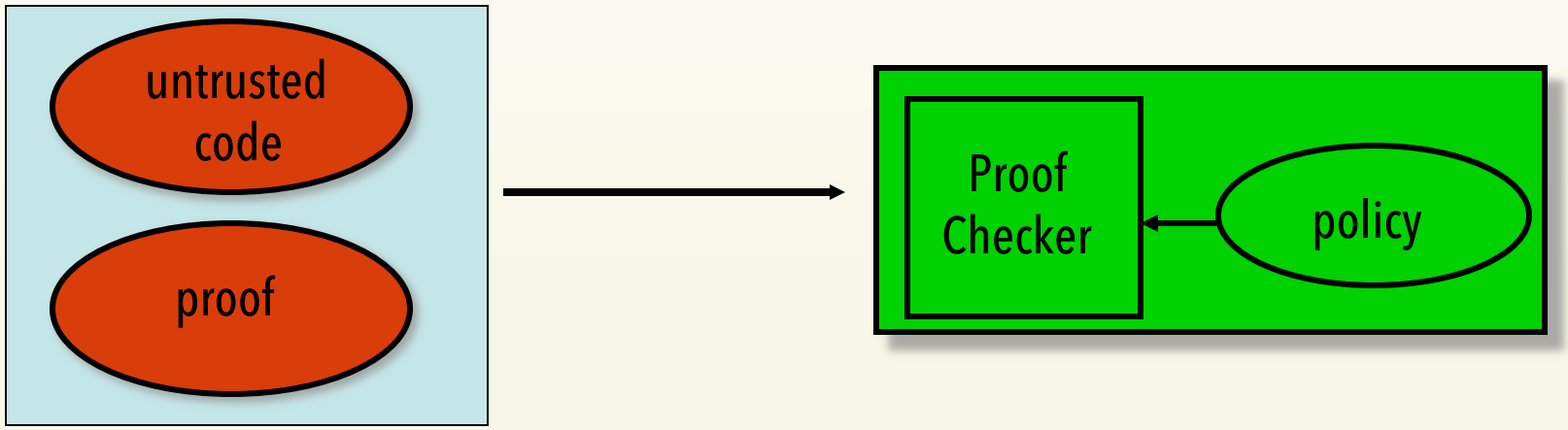
So *shift the burden*.

Key Observation

- Finding a proof is hard.
- Checking a proof can be easy.



Proof-Carrying Code [Necula & Lee '97]



Code comes with a *proof* that it satisfies the policy.

The proof checker ensures that:

a) the proof is valid

b) the conclusion says "this code respects the policy"

Good Properties of PCC

- We can build a trustworthy proof checker.
  - ~1K lines of code.
- The coupling is tamper-proof.
  - Change code: verifier will discover that the proof no longer talks about the same code.
  - Change proof: verifier will discover if it's no longer valid.
  - No secrets to have stolen.
- Relative completeness.
  - Any policy that can be formalized.
  - Any code that provably respects the policy.
- Enables integration
  - No longer matters who produced the code.

PCC is No Silver Bullet

- How to get proofs?
  - Policies of interest are hard to prove.
  - Manual proof construction is order(s) of magnitude harder to write than code.
- What policies should we enforce?
  - How do you formalize "nothing bad"?
- Proofs are relative to models.
  - How do we model the real world?

How to get proofs?

1. Use high-level code & a certifying compiler.
2. Use automatic analysis.
3. Insert checks that make it easier to build proof.
4. Change the policy so it's easier to build proof.
5. Get the programmer to help.

In reality, we have to do all of these…

Rest of this Talk

Research investments that can help us build proofs of safety & security for real code.

- Compiler verification
- Static analysis
- In-lined reference monitors
- Proof engineering & automation
- Domain-specific languages & logics

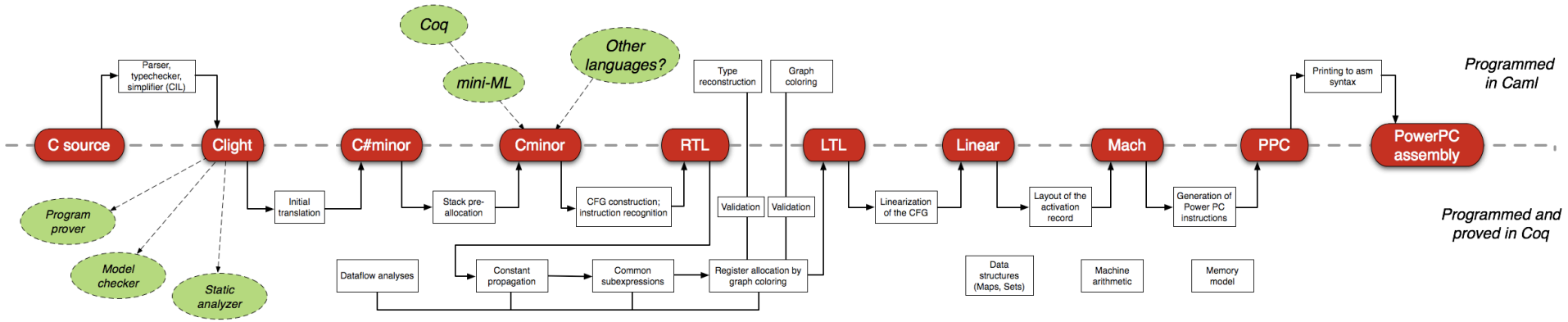
Reasoning About Machine Code

- Building proofs about machine code is *hard.*
- Prefer to construct proofs about a high-level language.
- But then there is a gap…
  - A bug in the compiler can lead to an exploit.
  - Most browser vulnerabilities are due to bugs in Java or Javascript implementations.
  - See *Yang et al.'s* 2011 PLDI paper.

Proven Correct Compilers

- CompCert [http://compcert.inria.fr]
  - Optimizing C compiler
  - Back-ends for x86, Arm, PPC
  - Competitive with `gcc –O1`
  - Proof of correctness:
    - C code has same I/O behavior of generated machine code.
    - Means we can reason about the source code instead of the machine code for most policies.
    - See for instance, Andrew Appel's program logic.

Proof Engineering



Still Many Challenges

- From -01 to -03; From WAT to JIT.
- Higher-level languages than C.
  - c.f., core-ML compiler out of Cambridge.
- Issues reasoning about multi-core programs.
  - c.f., Sewell & Batty's work on C++11.
- Proof of correctness ~10x the size of code.
- Some policies not preserved by refinement.

Proved Correct Compilers

Shift reasoning from machine to source code.

But we still need to produce a proof for the source code…

Static Analysis

- Static analysis tools are now viable for detecting a wide class of common bugs in source code.
  - Prefast, Coverity, HP/Fortify, …
  - Based on foundational research in program analysis from the 1980's-2000's.
- However, for legacy code:
  - Generate too many "false" positives.
    - For that matter, too many "true" positives as well.
  - Today's commercial tools are (purposefully) unsound.

An alternative:

In-lined Reference Monitors (IRMs)

- Formulate a *safety* policy.
  - e.g., will not access the network.
- Insert run-time checks into the code to enforce the policy.
  - Needed at security-critical events.
  - But also must insert checks to protect the monitor!
  - Makes it easy to prove that the compiler respects the policy.
  - Importantly: *avoids false positives of static analyses.*
- However, we can use static analysis, to optimize checks.
  - Only eliminate a check if you can prove it's safe to do so.
  - So the role of analysis is purely for optimization.

Some Example Policies

- SFI: Software Fault Isolation [Wahbe et al.]
- CFI: Control-Flow Isolation [Abadi et al.]
- XFI: Extended Flow Isolation [Erlingsson et al.]
- SafeCode [Dhurjati et al.]

These policies attempt to stop various forms of control-flow hijacking and/or data corruption attacks in legacy C/C++ code.

Policy Tradeoffs

1. What vulnerabilities are mitigated?
2. How much legacy code do we break?
3. How much overhead do we incur?
4. How hard is it to get the implementation *right?*

Some Example Policies

- SFI: Software Fault Isolation [Wahbe et al.]
  - Forces code to execute in a sandbox.
  - Low overhead (~5% on 32-bit x86), easy to enforce.
  - But doesn't stop hijacking code or data *within* the sandbox.
- CFI: Control-Flow Isolation [Abadi et al.]
  - Forces code to follow a control-flow graph.
  - Not as lightweight as SFI (~10-20%?), harder to implement.
  - Stops most (but not all) control hijacks such as ROP attacks; no data.
- XFI: Extended Flow Isolation [Erlingsson et al.]
  - Extends CFI with stack-protection, even more expensive.
- SafeCode [Dhurjati et al.]
  - Enforces a type-safety discipline (code + data).
  - Overheads range from 20-150%, very hard to implement well.
  - Stops all control hijacking, many data integrity attacks.

Zooming in on one of these…

- Google wanted to use SFI to provide a sandbox for their "Native Client" extension to the Chrome Browser.

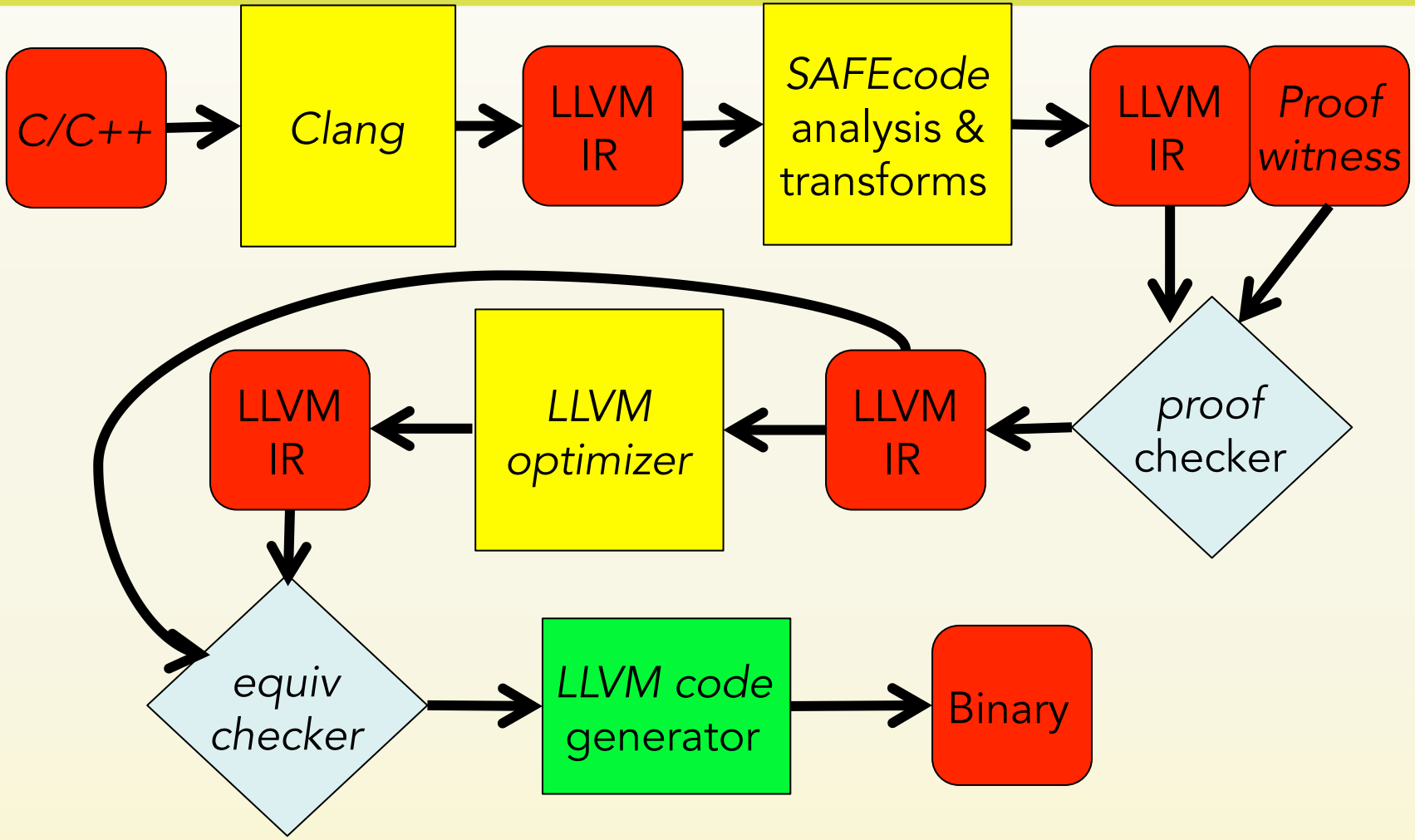
- We built a checker that allows Google to verify that a binary will respect the policy.
  - Specialized to this policy: 80 lines of code!
  - We proved that the checker is correct.
  - [See Morrisett et al., PLDI 2012].

Another IRM Example [Adve]

Based on the SAFEcode compiler [PLDI'06]:

- Compiles C, C++, Java, Haskell, etc.
- Enforces a much stronger policy than SFI.
- Works by instrumenting the LLVM intermediate representation + some runtime support.
- Has been used to compile Linux 2.4.22 & NetBSD
  - For Linux, prevented 4 of 5 known vulnerabilities
  - ~20% - 50% overhead

Certification for SAFEcode



Summary Thus Far…

- Formulate safety policy as an in-lined reference monitor.
  - Automates policy enforcement; simplifies proof.
- Technologies from analysis used to cut overheads of monitoring.
- Technologies from proof-preserving compilation eliminate the need to trust the tools.

Modeling

- A major challenge for both the Google and SAFECode efforts was constructing formal models of the underlying machines.
  - x86 model has thousands of instructions.
  - Building & validating such models is crucial for *any* real-world application of formal methods.
- UK researchers have taken the lead here: C++, x86, ARM, TCP, Javascript, …

Richer Policies…

- IRM's make it possible to automate basic safety properties.

- But for safety & security-critical software, we need policies that cover confidentiality, availability, and functional correctness.
  - Much harder to get proofs.
  - Example: SEL4 micro-kernel + proof of correctness ~20 person years of effort.

Proof Automation

- Some of this is alleviated by advances in automatic theorem proving technology.
  - SAT solvers; SMT provers.
  - Both have seen dramatic improvements.
  - Still have many hard challenges here.
- Much more is alleviated by using domain-specific languages, logic, & decision procedures.

Examples: Confidentiality

- Jif, LIO:
  - Information flow tracking through types
  - Ensure information from private fields in data do not flow to public channels.
  - Dually, public data cannot influence integrity.
- EasyCrypt, FCF:
  - Domain-specific languages & logics for reasoning about cryptographic schemes (e.g., TLS.)
  - Connect cryptographer-level proofs to actual code.

Wrapping it up:

- Proof-carrying code (PCC) enables trust.
  - Doesn't matter who wrote the code.
  - Can verify with small trusted computing base.
  - Important for scaling software, where components are brought in from 3rd parties, open source, etc.
- Certifying compilers help produce PCC:
  - prove properties at the source level.
  - no need to trust compiler or reveal the source.

Getting Proofs:

- Today:
  - Safety policies enforced by in-lined reference monitors.
  - Stop a wide range of common attacks.
- Tomorrow:
  - New languages let us capture a range of policies: integrity, confidentiality, availability, correctness.
  - New analysis techniques & decision procedures help automate proof construction.

Thanks!

Questions? Comments?