Unifying Disparate Tools in Software Security

Greg Morrisett

When can you trust code?

galaga.exe

When can you trust code?

galaga.exe

Traditional Computer Security

The focus was on attacks from outside the perimeter.

Assumption:

An authorized user knows what they're doing when they execute a program.

The Context has Changed:

We constantly download and run code.

- Drivers, patches, games, applications
- Web pages, email, spreadsheets, postscript

Who knows what it can do?

Attackers can blow by
our firewalls, passwords, …

Attack Incentive?

- Steal confidential information:
  - social security number, credit card number
  - passwords for stores, banks, services
- Set up an illegal service:
  - copyrighted media sharing, child pornography
- Blackmail denial-of-service:
  - send us $1M or we flood your website with traffic
- Send spam:
  - buy v1agRa! I'm from Nigeria…
  - penny stock pump-and-dump

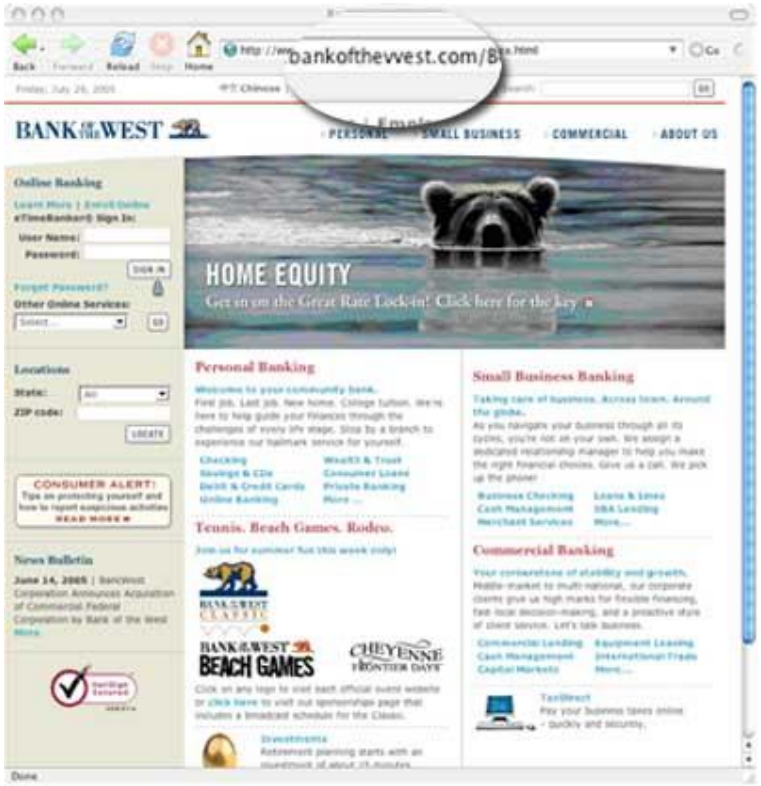
Primary Defenses Today:

- Users
  - don’t click on untrusted links
- Virus scanners
  - look for snippets of code known to be bad
- Digital signatures
  - check that signer is a trusted entity
  - check that code hasn’t been modified

Shortcomings:

Users:

- Attackers are *clever.*
- Hide code in useful applications.
- Use “social engineering” tricks to fool users.
- It’s difficult to tell what is dangerous, even for experts.



Shortcomings:

Virus scanners:

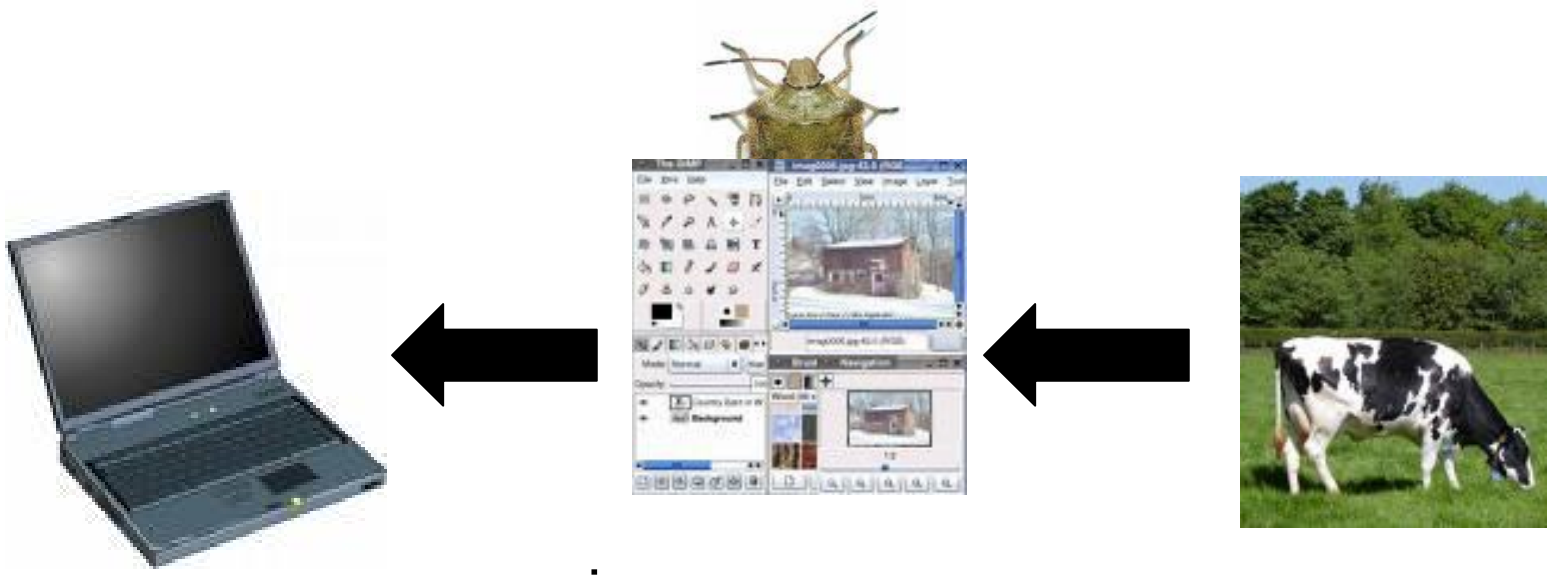
- only detect known attacks
- difficult to scale over time
  - old attacks do not go away
  - so database of virus snippets only grows
- easy to defeat by obfuscating the code
  - e.g., "if (c) then S_1 else S_2" $\rightarrow$ "if (not c) then S_2 else S_1"
  - can download toolkits to automate rewriting

Shortcomings:

Digital signatures:

- Do you trust noname.com?
  - No help when you don't know the signer.
- It only costs $250 to get a certificate
  - Certificate authorities have little incentive to do adequate checks to ensure signer is trustworthy.
- Do you trust microsoft? nvidia? ati? adobe? oracle? apple? valve? mozilla? sony?
  - All well-meaning companies…

“Good” Code Can Have Bugs:

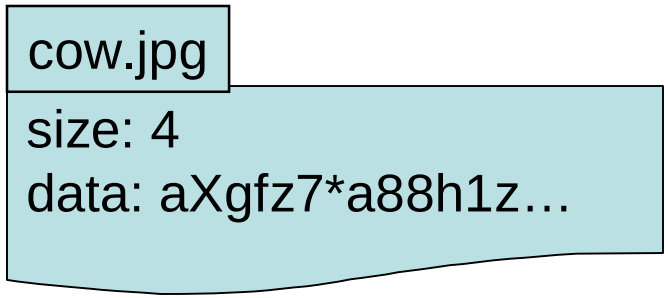


imagemax.exe

cow.jpg

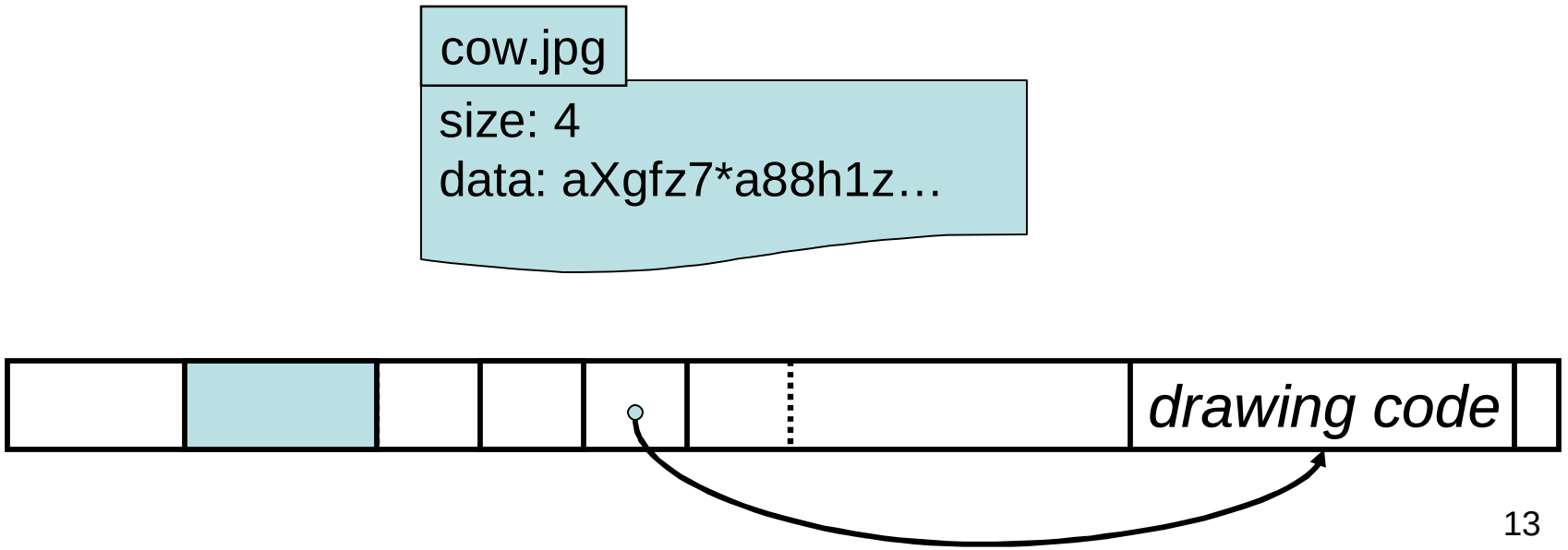
Example: Buffer overrun

```
int n = get_image_size(f);
char *buf = malloc(n);
get_image(f,buf);
```



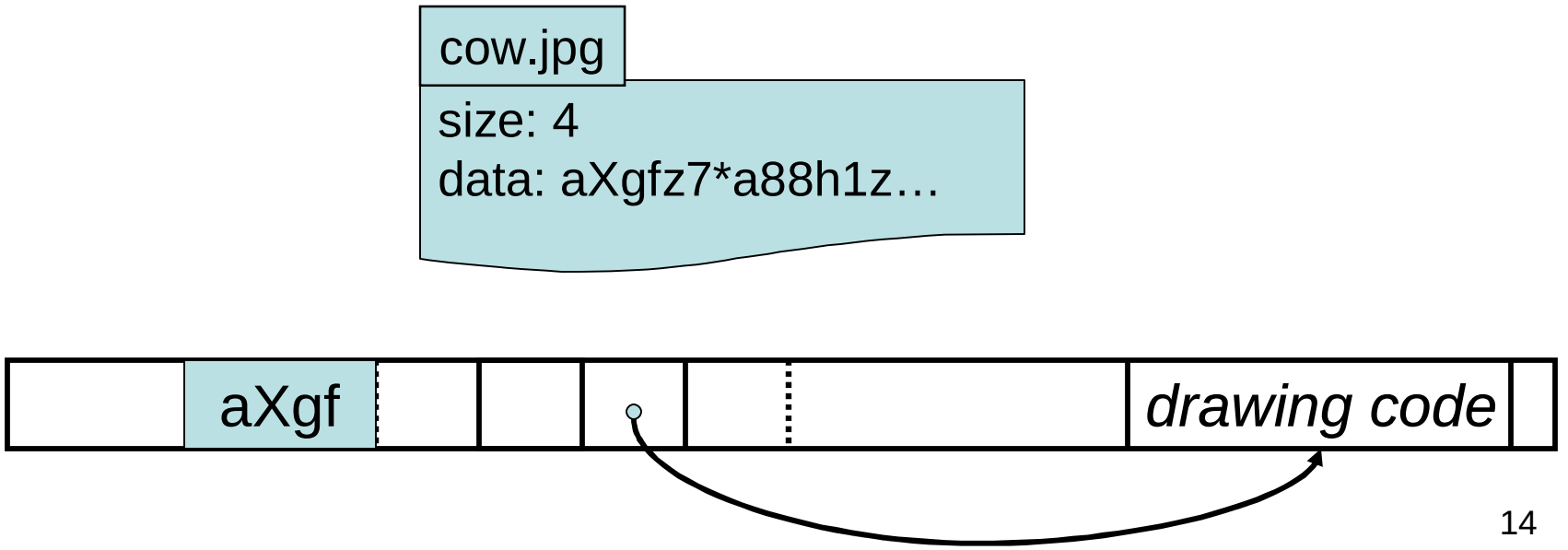
Buffer Overrun:

```
int n = get_image_size(n);
char *buf = malloc(n);
get_image(f,buf);
```



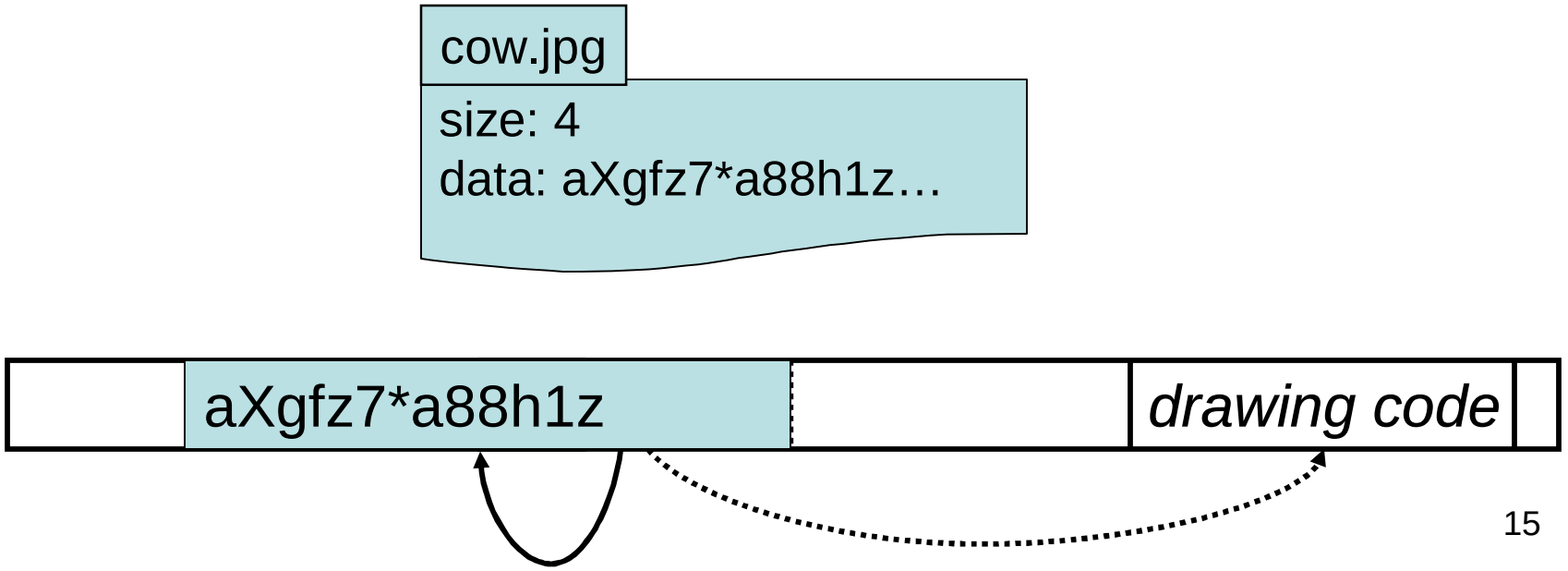
Buffer Overrun:

```
int n = get_image_size(n);
char *buf = malloc(n);
get_image(f,buf);
```



Buffer Overrun:

```
int n = get_image_size(n);
char *buf = malloc(n);
get_image(f,buf);
```



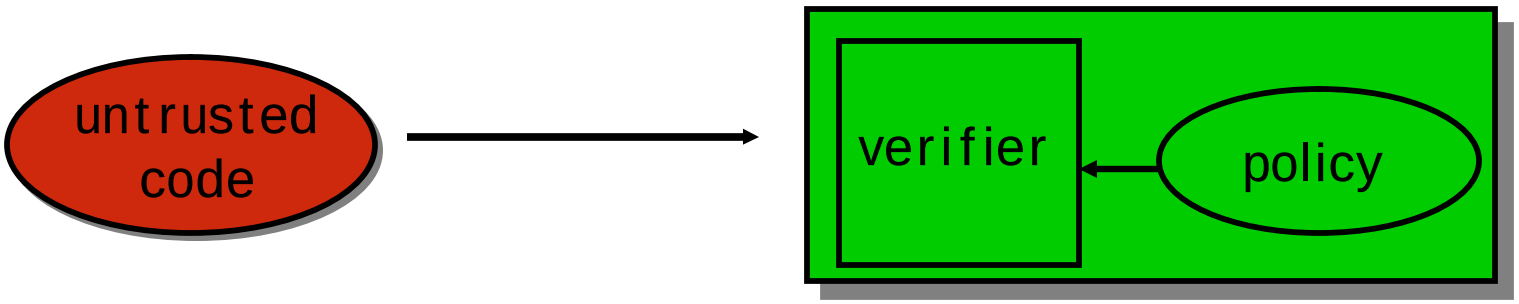
Syntax vs. Semantics

Checking the *syntax* or the *provenance* of code is too weak to ensure that the code is trustworthy.

What we want is to validate the *behavior* of the code.

This is the focus of *software security*.

Ideal Architecture:



Policies capture *behavior.*

Verifier automatically rules out any code that will violate the policy.

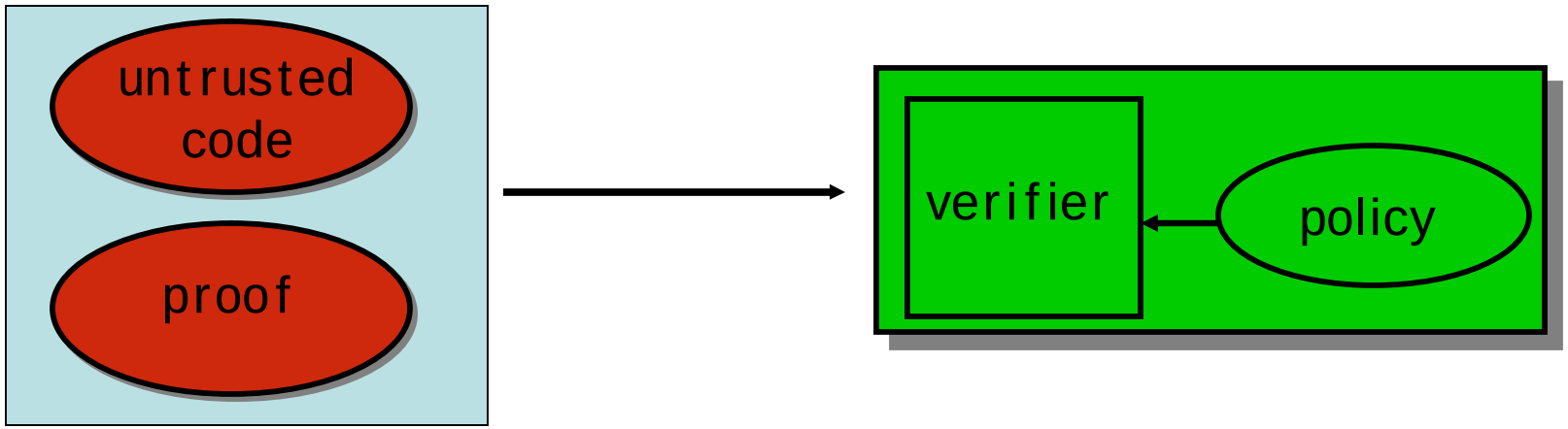
Verifier is small, simple, trustworthy, and automatic.

Unfortunately:

- Even simple policies are undecidable.
  - If a verifier can automatically determine whether a program has a buffer overrun, we can use it to solve the halting problem.
  - So any verifier is either incomplete or unsound.

- Analyzing machine code is *hard*.
  - Analyzing source or byte code is hard enough.
  - Good analyses for array bounds checks include things like ILP solvers, symbolic theorem provers, pointer analyses, etc.
  - So any useful verifier is big and complicated.
  - Can we trust it?
  - Do you trust your compiler?

Proof-Carrying Code: [Necula & Lee '97]



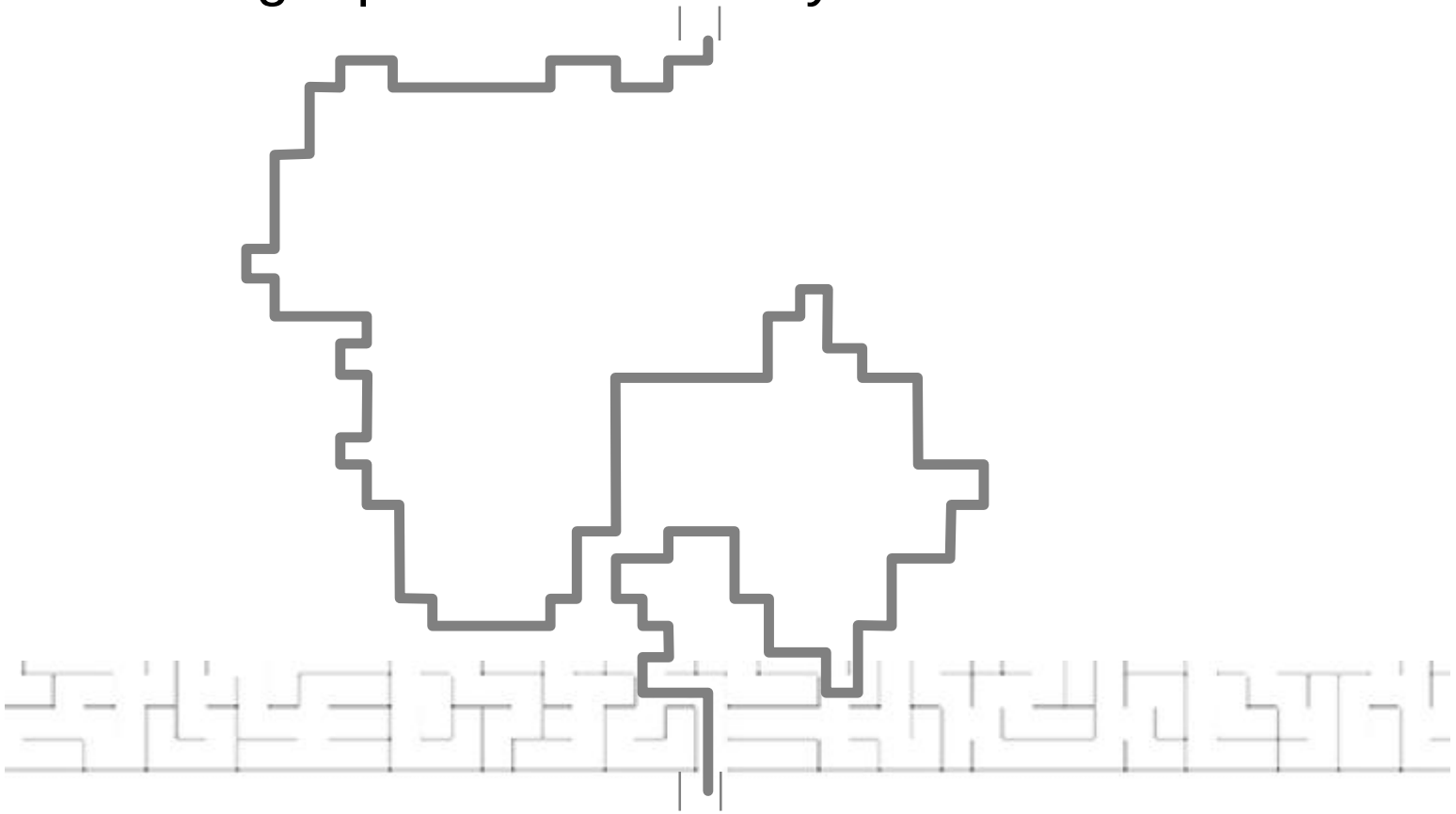
Code comes with a *proof* that it satisfies the policy.

The verifier checks that:

a) the proof is valid
b) the conclusion says "this code respects the policy"

Key Observation

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



PCC on paper:

- Simple and trustworthy verifier
  - about 1K lines of code.
  - within range of formal verification.
- The coupling is tamper-proof
  - change the code: verifier will discover that the proof no longer talks about the same code.
  - change the proof: verifier will discover if it's no longer valid.
- Relative completeness
  - accept any code that *provably* respects the policy.
- No need to trust compiler or other tools.

PCC is No Silver Bullet

Many low-level technical issues:

- e.g., how to represent proofs
- e.g., what logic, axioms to use

Key issue: How do good guys produce proofs?

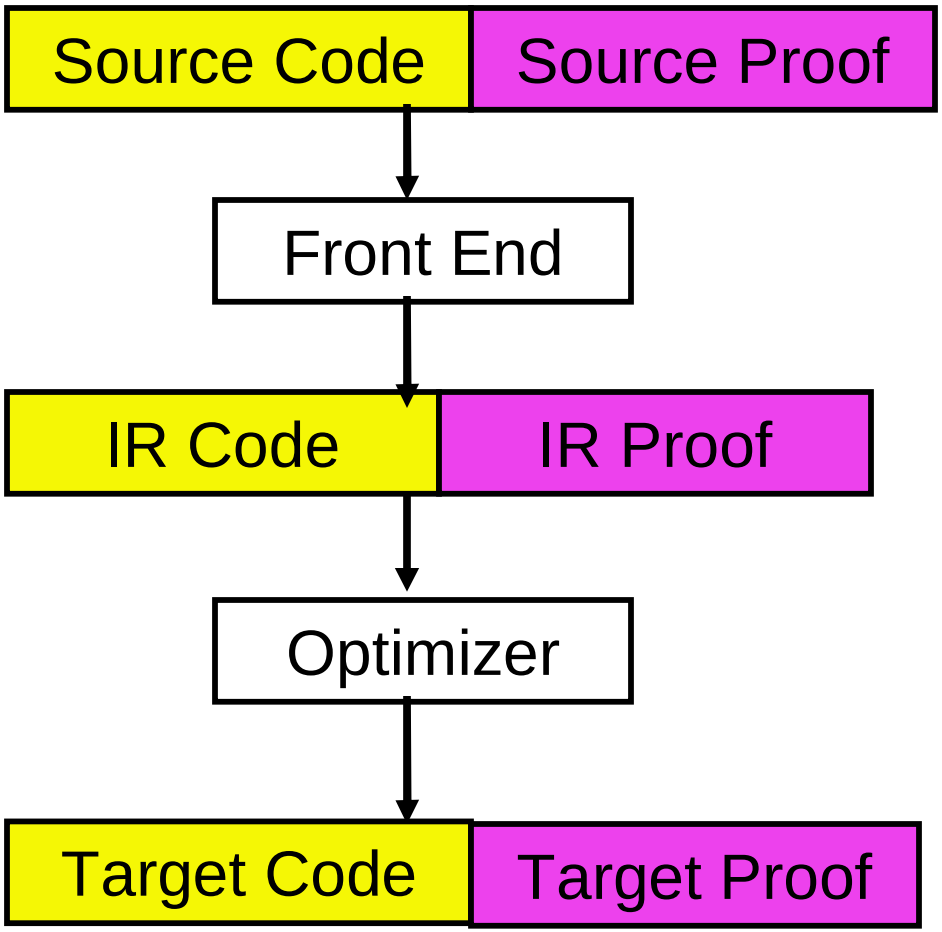
- PCC simply shifts the burden from the consumer to the producer of the code.
- The really hard problem of proving properties of code remains.

How to get the source proofs…

1. Restrict the code so it is easy to analyze.
2. Restrict the policy so it is easy to prove.
3. Use static analysis & theorem provers to synthesize the proof.
4. Rewrite the code so that it's easy to prove the code respects the policy.
5. Get the programmer to help construct the proof.

In reality, we have to do all of these…

Certifying Compilers



Takes as input source code and a proof that the source respects the policy.

Produces target code and proof by doing proof-preserving compilation.

Now we only have to prove properties at the source-level.

Example: Special-J [Colby et al. '00]

1. Limit the code to Java.
2. Limit the policy to type-safety.
3. Use type-inference to construct the proof.
4. Where the analysis is too conservative, insert run-time checks (e.g., downcasts, array-bounds checks).

Compiler automatically produces proof that x86 machine code respects type-safety.

Types and Proofs

When you interact with a type-checker, you're really doing a form of interactive theorem proving.

Today, the prover is pretty dumb, and the theorem you're proving is pretty weak.

- prevents buffer overruns
- but not higher-level policy issues

This is starting to change…

Beyond Simple Types: [Hamlen'06]

1. Extended type system for .NET.
2. Policy: object-level security automata
   - e.g., user-level input must have been validated before flowing to database as query.
3. Rewrite code to track states of objects and check states upon actions.
4. Dataflow analysis to eliminate state + checks where provably safe to do so.

Other Emerging Systems:

- ESC/Java, Spec#
  - pre/post-conditions, object invariants integrated into type system.
  - can rule out many errors at compile-time.
  - SMT-based theorem prover discharges proofs.
- Coq, Epigram, ATS, Ynot:
  - powerful program logic integrated into types.
  - can capture simpe errors up to full correctness.
  - programmers construct proofs with help of automated decision-procedures.

Realistic?

On the one hand, yes:

- [X.Leroy ‘06]
- built an optimizing compiler in Coq
- maps a subset of C to PowerPC code
- the types captured full *correctness*
  (i.e., input code behaves same as output)

On the other hand, not yet:

- proof constructed largely by hand
- an order of magnitude bigger than code…
- will advances in prover automation bring this down to something feasible for commercial code?

Legacy code?

What about existing C/C++ code?

- Can't afford to re-code Vista in .NET.
  - Vista is roughly 50 million lines of code.
  - Likely to introduce as many bugs as it kills.
- Many low-level services cannot be written in today's high-level languages.
  - e.g., the .NET garbage collector!

How do we play the PCC game here?

Today: Imperfect Tools

- C/C++ source code bug-finders:
  - Prefast, Fortify SCA, Coverity, …
  - very effective at finding bugs
  - tradeoffs: precision, false positives
- Hardware, compiler and run-time tricks:
  - Stackguard, NEX, address randomization, …
  - inject artificial “diversity” into code
  - harder for attackers to inject code
  - tradeoffs: breaking code, performance overhead

Effective, but for how long?

Emerging Research Tools

- Ccured [Necula et al '02]
  - rewrites code to check type safety.
  - adds meta-data to support checks.
  - optimizes checks & state using whole-program analysis.
- Cyclone [Jim et al '02]
  - similar, but advanced types let programmers have more control over data representations, and avoid more checks.

Wrapping it up:

- Proof-carrying code 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.
  - compiler transforms proof to target level.
  - no need to trust compiler or reveal the source.

But we still need proofs:

- Today:
  - limit the policy to type-safety.
  - a big challenge is legacy C/C++ code.
- Tomorrow:
  - new languages let us capture a range of policies from simple types to full correctness.
  - new analysis techniques & decision procedures help automate proof construction.
  - working in the lab, but still a long way to making the vision practical for commercial software.