

# Advances in Programming Languages

## Lecture 15: Certifying Correctness

Ian Stark

School of Informatics  
The University of Edinburgh

Thursday 8 November 2018  
Semester 1 Week 8



# Outline

- 1 Opening
- 2 Proving Programs Correct
- 3 Certifying Code
- 4 Verifying Complete Systems
- 5 Conclusion

# Outline

- 1 Opening
- 2 Proving Programs Correct
- 3 Certifying Code
- 4 Verifying Complete Systems
- 5 Conclusion

# Topic: Augmented Languages for Correctness and Certification

This block of lectures cover some language techniques and tools for improving program correctness:

- Specification and Verification
- Practical Java tools for Correctness
- Separation Logic
- **Certifying Correctness**

The focus here is not necessarily on changing what a program does, or making it do that thing faster, or using less memory, or less power. Instead we want to make sure that what it does is the right thing to do.

Following previous lectures on Hoare Logic, the Java Modeling Language and Separation Logic, this talk looks at how these can be taken beyond local checking of individual programs and into a larger whole-systems framework.

# Homework from Monday

## 1. Watch this

Peter O'Hearn: "Continuous Reasoning: Scaling the impact of formal methods"  
Plenary talk at the Federated Logic Conference 2018.

Video: <https://is.gd/continuousreasoning>

More information: <https://www.floc2018.org/speaker/peter-ohearn>

## 2. Do this

Complete and submit your written assignment. If you have questions then send me mail or post on Piazza.



```
errorVar: cat
{
  cat.h0();
}
{
  if (cat == null)
    return;
  cat.h0();
}

errorVar: cat
1 {
  cat.sleep();
}
{
  if (cat == null)
    return;
  cat.sleep();
}

errorVar: cat
1 {
  cat.meow(volume);
}
{
  if (cat == null)
    return;
  cat.meow(volume);
}

errorVar: dog
{
  if (dog == null)
```

Merge edits which check `cat` for null before calling `cat.h0()` (either `sleep()` or `meow()`)



POSTED ON NOV 6, 2018 TO [DEVELOPER TOOLS](#), [ML APPLICATIONS](#)

# Getfix: How Facebook tools learn to fix bugs automatically

# Outline

- 1 Opening
- 2 Proving Programs Correct**
- 3 Certifying Code
- 4 Verifying Complete Systems
- 5 Conclusion

# Specification and Verification

**Specification** Stating what properties a program ought to have, either before it is written at all, or by annotating existing code.

For example, we might do this with Hoare preconditions and postconditions; object invariants; code contracts; separation logic; or wit annotations about non-null pointers, lack of side-effects, control of exceptions, etc.

**Verification** Checking that a program does indeed have these desired properties.

This is *static checking*, which might be straightforwardly automatic, or require additional user annotations such as loop invariants or library specifications.

Some static checkers turn the problem into *verification conditions* in formal logic and pass these to an automated theorem prover — perhaps an *SMT solver* able to reason about arithmetic, arrays, bitvectors, and other relevant theories.

Program testing can be used to show the presence  
of bugs, but never to show their absence!  
— E. W. Dijkstra, 1969



# Lightweight Verification

Proving arbitrary assertions can be arbitrarily difficult.

In *lightweight verification* we simplify things by focusing on standard properties of common interest, rather than full functional correctness. For example:

**Exception freedom** no uncaught exception is raised.

**Pointer validity** no null pointer is ever dereferenced.

**Arithmetic safety** no arithmetic expression divides by zero or overflows.

**Race freedom** access to shared state does not conflict in different threads.

Standard properties are easy for the programmer to indicate, providing shorthand for possibly complex logical expressions.

Standard properties may be easier for tools to handle, using *ad hoc* static analyses or decidable fragments of logic.

If a tool cannot establish a property, then the programmer may be able to add additional annotations, or perhaps to rewrite the code in a way that makes the property evident.

# Many Different Things to Check

This list shows some of the many things that the *ESC/Java2* tool checked in source code.

Other tools provide similar checks, often specialising in a particular area and possibly relying on user interaction or additional programmer annotations.

- Null pointer dereference
- Negative array index
- Array index too large
- Invalid type casts
- Array storage type mismatch
- Divide by zero
- Negative array size
- Unreachable code
- Deadlock in concurrent code
- Race condition
- Unchecked exception
- Object invariant broken
- Loop invariant broken
- Precondition not satisfied
- Postcondition not satisfied
- Assertion not satisfied

# Outline

- 1 Opening
- 2 Proving Programs Correct
- 3 Certifying Code**
- 4 Verifying Complete Systems
- 5 Conclusion

# Once We Have Verified Code, What Then?

## Oxford English Dictionary

**Certify** *v. trans.* To make (a thing) certain; to guarantee as certain, attest in an authoritative manner; to give certain information of.

Once we have verified code, or shown it satisfies some key property, how do we communicate this to others? How do we decide whether code from other sources is trustworthy?

- Informal argument written in English
- Checklist of manually measured/assessed criteria
- Set of executable tests that are checked automatically
- A signature of an authority, analogue or digital
- Transcript of input and output to a verification system
- Digital evidence checked electronically

# Code Signing

**Digital signatures** are a very widely-used mechanism for certifying code: mobile device apps; Microsoft updates; Windows Driver Signing; Linux package archives; browser plugins; ...

This *code signing* is better than trusting unauthenticated code; a digital version of “shrink-wrapping” products with holographic stamps.

However, this is authenticating the *supplier* of the product, and doesn't directly say anything about the product itself.



# What Does Code Signing Ensure?

Code signing has two particular fallibilities.

1. The signature chain may be compromised.

## Microsoft Security Bulletin MS01-017 - Critical

In mid-March 2001, VeriSign, Inc., advised Microsoft that on January 29 and 30, 2001, it issued two VeriSign Class 3 code-signing digital certificates to an individual who fraudulently claimed to be a Microsoft employee. The common name assigned to both certificates is "Microsoft Corporation".

## Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI

Analysis of malware that bypasses system and anti-virus protection with signed code certificates.

"We identify 27 certificates issued to malicious actors impersonating legitimate companies."

CCS 2017 — <http://signedmalware.org/>

# What Does Code Signing Ensure?

Code signing has two particular fallibilities.

2. The code being signed may not in fact be trustworthy.

This might be because the signer failed to check properly, procedures weren't adequate, or that they weren't followed. This might be accidental or malicious.



# What are we Missing Here?

Aside from errors in implementation or procedure, there is a deeper issue that code signing delegates trust to somebody else rather than examining the code for ourselves.

Digital signatures authenticate the supplier of code, not the code itself.

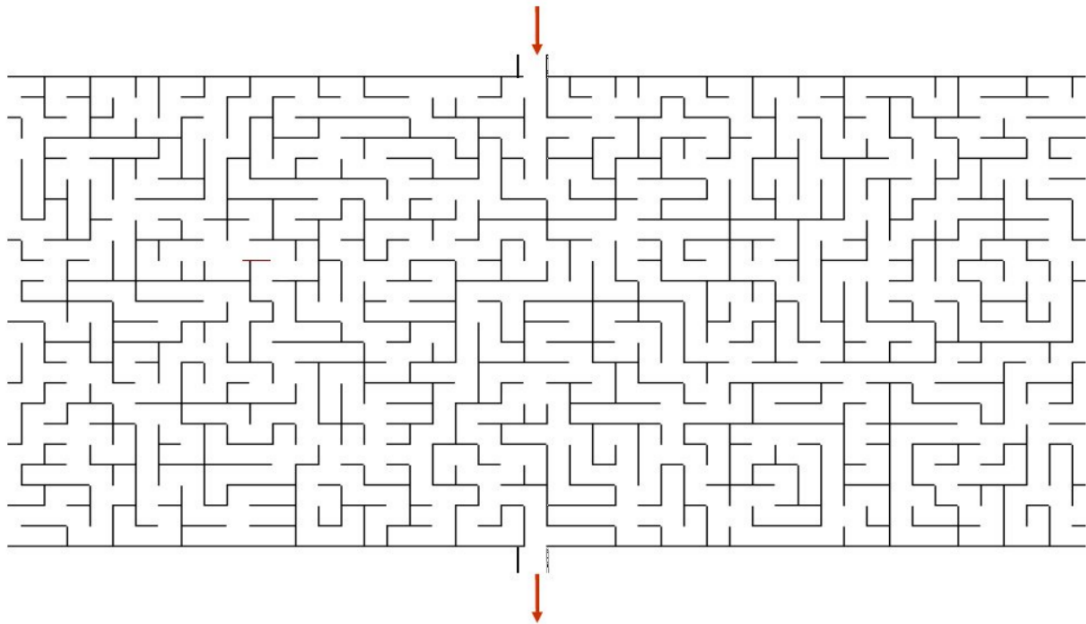
What if we could check the code ourselves, directly?

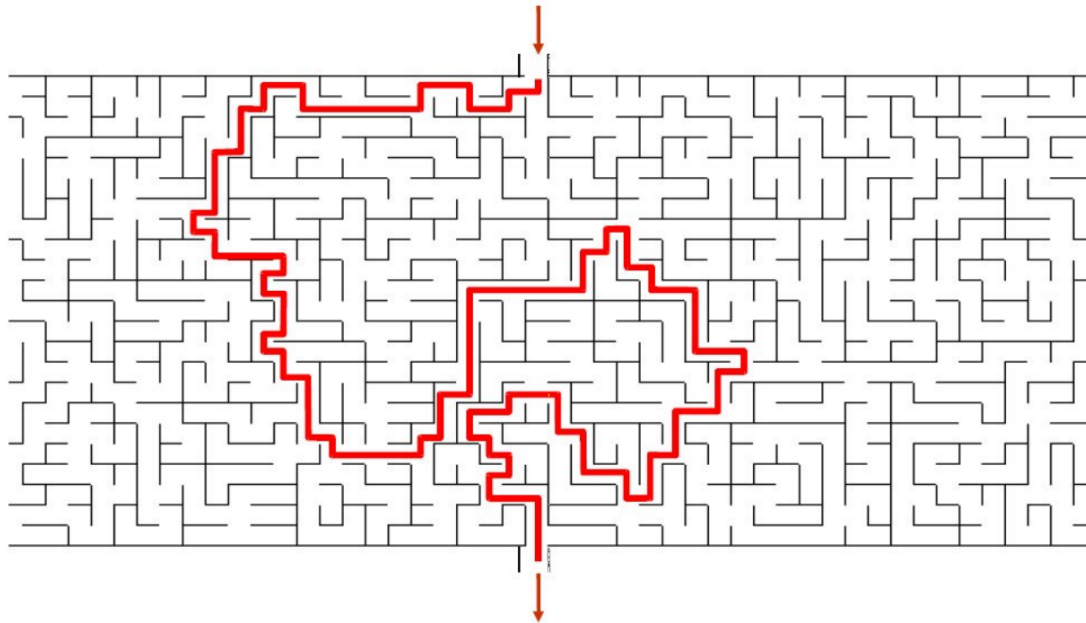
One challenge here is that correctness proofs are hard; and even harder if we don't have the source code, its annotations, or any insight from the programmer.

Often, however, **checking** a proof is much simpler than **finding** a proof.

This lies behind research into *Proof-Carrying Code* (PCC) and, more generally, the idea of *evidence-based security*.







# Proof-Carrying Code

PCC was originally proposed by George Necula and Peter Lee in 1996 with the example of network packet filters. It has since been applied in a variety of contexts to provide assurances about code safety.

The general mechanism is as follows.

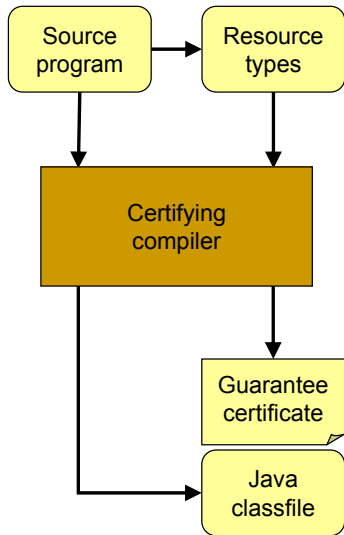
A **code producer** bundles together:

- Executable code;
- A formal statement about the code's behaviour — the *guarantee*;
- A proof that the code satisfies the guarantee.

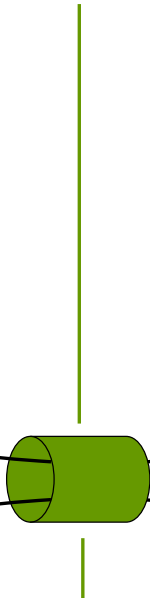
The **code consumer** receives the bundle and checks:

- That the guarantee ensures the desired security or safety properties;
- That the proof does show the code satisfies the guarantee.

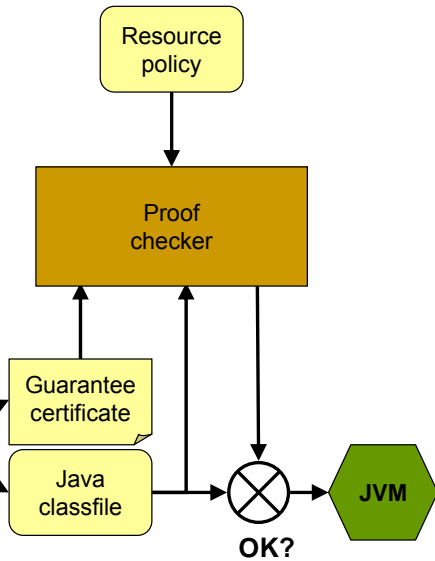
## Code producer



## Network



## Code consumer



# Digital Evidence

The PCC framework can extend beyond proofs to all kinds of *digital evidence*.

- Machine-checked proofs; fixpoints for abstract interpretation; loop invariants for bytecode logics; constraint solutions; ...

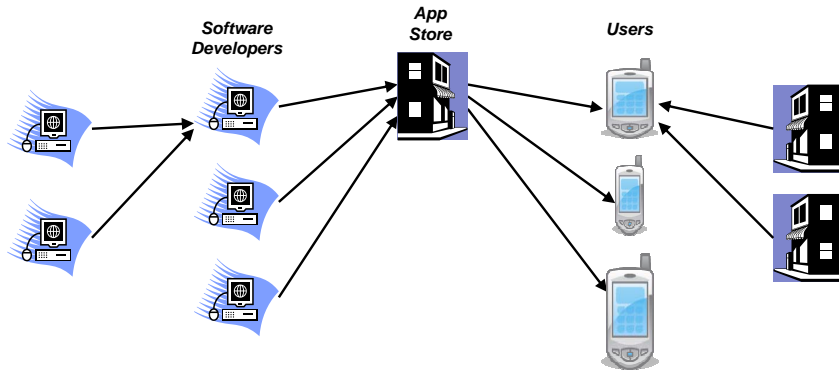
These may show different kinds of properties:

- Type safety, memory safety, stack and array bounds;
- Secure information flow, noninterference, declassification;
- Resource usage, access policies;
- Invocation protocols, method contracts; ...

Some existing examples:

- Stackmaps in the Java bytecode verifier; similarly for .NET;
- Typed Assembly Language (TAL) in the *Verve OS* from Microsoft Research.

# Wholesale Digital Evidence



## Flows for Digital Evidence

- From store to user, evidence to satisfy security/resource policy
- From store to developer, stating objective acceptance policies
- From developer to store, providing evidence to meet these

# Summary on PCC and Digital Evidence

Trust in mobile code can be strengthened by supplementing **digital signatures** with **digital evidence**.

## Digital Signatures

- Cryptographic, confirming that software has been approved
- Checked against external trust hierarchy of public keys

## Digital Evidence

- Certificate presenting data about the software itself
- Confirms key aspects of software behaviour
- Can be independently checked, without external authority

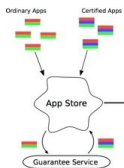
Both rely on mathematical foundations to ensure that no certificate can be forged, and that code cannot be tampered with. Both can work without revealing any original source code.



Mobile Resource Guarantees  
Edinburgh/Munich 2002–2005



Mobius: Mobility, Ubiquity and Security  
Enabling proof-carrying code for Java on mobile devices  
European integrated project 2004–2009



App Guardian: Resilient Application Stores  
Edinburgh 2013–2016  
UK national cybersecurity programme GCHQ/EPSRC/BIS

Mobility and Security

<http://www.lfcs.ed.ac.uk/m+s>

Security and Privacy

<http://secpriv.inf.ed.ac.uk/>



# Outline

- 1 Opening
- 2 Proving Programs Correct
- 3 Certifying Code
- 4 Verifying Complete Systems**
- 5 Conclusion

# There's More to Check than Code

Static checkers make it possible to verify properties of source code.

Proof-carrying code and digital evidence lets users check properties of executable binaries.

But how do we get from proofs about source code to proofs about binaries?

And there is more — what must we do to trust all the other components?

- Compiler
- Bytecode engine / JIT compiler
- Hardware execution of machine code
- Libraries
- Host operating system
- ...

# Compiler Correctness

Compiler correctness has been a concern ever since there were compilers. Different people take different views on this; two of these we can stereotype as follows.

## Programming-Language Researchers

Compilers are complicated pieces of code. We rely on them, but they are buggy.

Like other complicated pieces of code aim to verify that they do what is intended, and fix them if they don't.

## Compiler Researchers

Compilers are *very* complicated pieces of code. We rely on them and they are buggy. But that's life.

The underlying hardware also has bugs. There haven't been any verified CPU cores since the 1980s. We just hope the bugs are rare, and for controlling nuclear power stations it's probably best to only use old CPU architectures.

# Translation Validation

The idea of *translation validation* is that rather than verifying a whole compiler, we check that **individual** compilations actually performed are correct.

This can be even be refined to individual optimisation steps, or specific code transformations.

- Translation validation relaxes the requirement that the compiler be globally correct. Instead, the compiler checks that it has produced the right result each time.
- Some input programs may produce errors or trigger compiler bugs; validations cannot be produced for these programs.

Amir Pnueli, Michael Siegel and Eli Singerman. *Translation Validation*, TACAS '98.

<http://portal.acm.org/citation.cfm?id=691453>

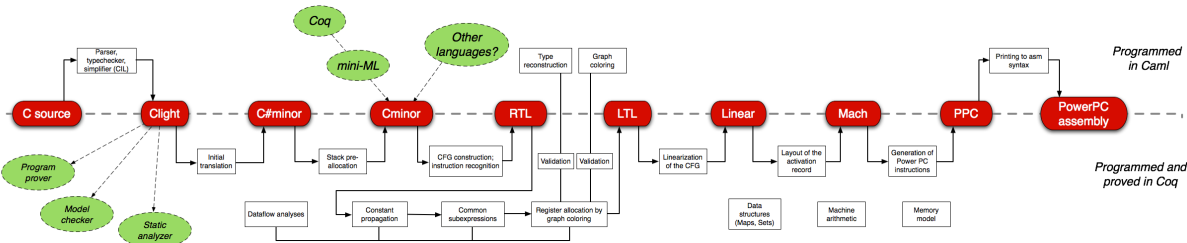
George C. Necula. *Translation Validation for an Optimizing Compiler*, SIGPLAN Notices, 35/5, 2000. <http://doi.acm.org/10.1145/358438.349314>.

# Complete Compiler Verification

The *CompCert* project, led by Xavier Leroy, maintains a C compiler for PowerPC, ARM and x86 that has been formally verified using the *Coq* theorem-prover.

This verification shows that the compiler is *semantics-preserving*. As a result any safety property proved on the source code is sure to hold also for the compiled executable binary.

<http://compcert.inria.fr>



# Further Compiler Verification

Following Leroy's lead, other projects have built further verified compilers. For example:



## CerCo: Certified Complexity

A verified C compiler that provides cycle-precise execution costs for an embedded microcontroller.

Edinburgh / Bologna / Paris

<http://cerco.cs.unibo.it>



## CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency

Enhances CompCert with concurrency primitives for thread management and synchronisation, and checks their semantics against the *Total Store Order* (TSO) weak memory model.

Cambridge

<http://www.cl.cam.ac.uk/~pes20/CompCertTSO>



## CakeML: A Verified Implementation of ML

Verified compiler from ML to x86-64; the compiler is self-hosting (can compile itself)

Cambridge / NICTA Canberra / Kent

<https://cakeml.org>

# Platform Verification

Other projects aim to verify yet more of the tower between source and execution. For example:



## SAFE: A Secure Computing Platform

<http://www.crash-safe.org>

Built on *tagged hardware* that allows per-instruction checking of safety and security policies.

DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH)



## seL4: A Secure OS Kernel

Verified implementation of the L4 operating system microkernel.

National ICT Australia (NICTA)

<http://sel4.systems>



## REMS: Rigorous Engineering for Mainstream Systems

Specification and verification for processor architectures, systems programming, and concurrent software.

Cambridge / London / Edinburgh

<https://www.cl.cam.ac.uk/~pes20/remis/>



## HACMS: High-Assurance Cyber-Military Systems

Formal verification for embedded control systems and autonomous vehicles

US DARPA

[https://is.gd/hacms\\_darpa](https://is.gd/hacms_darpa)







# Outline

- 1 Opening
- 2 Proving Programs Correct
- 3 Certifying Code
- 4 Verifying Complete Systems
- 5 Conclusion**

# Summary

## Proving Programs Correct

**Specification** states what properties are expected of a program.

**Verification** checks that the code given does have those properties

**Lightweight specification and verification** focuses on simple properties not full correctness.

## Certifying Code

**Digital Signatures** authenticate the supplier of code.

**Proof-Carrying Code** certifies the code itself and can be independently checked.

## Verifying Complete Systems

**Translation Validation** checks compilation of individual programs.

**Compiler Verification** checks the compiler itself, as in the *CompCert* C compiler.

**Platform Verification** looks at other components between source and runtime behaviour.

# Homework

## 1. Do this

Watch these two short talks where Prof. Kathleen Fisher, first HACMS project leader, explains to project and its results: [https://is.gd/hacms\\_fisher](https://is.gd/hacms_fisher) and [https://is.gd/hacms\\_keynote](https://is.gd/hacms_keynote)

To find out more about how the project went and what's next read these two articles.

- [https://is.gd/hacms\\_quadcopter](https://is.gd/hacms_quadcopter)
- [https://is.gd/hacms\\_helicopter](https://is.gd/hacms_helicopter)
- [https://is.gd/hacms\\_report](https://is.gd/hacms_report)

## 2. Read this



Ken Thompson

Reflections on Trusting Trust

Communications of the ACM, 27(8):761–763, 1984.

DOI: 10.1145/358198.358210