Thinking about attacks

Minimizing trusted base

Andrew W. Appel,
January 14, 2013
I. **THE PROBLEM**

II. **HOW TO SOLVE IT**
Bugs, vulnerabilities, trojans

Two lessons from the 1980s

1. Trojan horses can be inserted into your synthesis toolchain where you wouldn’t have imagined

Ken Thompson, “Reflections on trusting trust”, 1984

2. “Bugs” can be exploited as “security vulnerabilities” in ways you wouldn’t have imagined


[Turing award lecture]

trojans hiding in the compiler

[felony conviction]

exploit buffer overrun bugs via code injection, among other things
The limits of testing

1. Trojan horses can be inserted into your synthesis toolchain where you wouldn’t have imagined

2. “Bugs” can be exploited as “security vulnerabilities” in ways you wouldn’t have imagined

Program (and hardware) testing, coverage testing, etc. will be less effective against clever hackers than against inadvertent error

1. Trojan horses will carefully avoid responding except to very specific stimuli

2. Many bugs may be found, it won’t be clear which ones are exploitable vulnerabilities
The limits of fault tolerance

1. Fault tolerance (a successful and useful field) typically assumes uncorrelated and random faults

2. Malicious adversary will cause correlated, nonrandom faults/attacks

Program (and hardware) testing, coverage testing, etc. will be *less effective* against clever hackers than against inadvertent error

1. Trojan horses will carefully avoid responding except to very specific stimuli

2. Many bugs may be found, it won’t be clear which ones are exploitable vulnerabilities
Toolchain / layers of abstraction

Program verification / static analysis / software model checking

*Source-language program*

Compiler front end

*Compiler intermediate representation*

Compiler back end

*MACHINE-language program / Instruction-set Architecture*

ISA to RTL

*Register-transfer language*

Hardware synthesis

*Netlist*
Adversary will attack at any level of abstraction that he can.

- Source-language program
- Compiler front end
- Compiler intermediate representation
- Compiler back end
- Machine-language program / Instruction-set Architecture
- ISA to RTL
- Register-transfer language
- Hardware synthesis
- Netlist
Hardware Toolchain in more detail

- Instruction-set Architecture
- ISA to RTL
- Register-transfer language
- Hardware synthesis
- Netlist
- Chip functional spec
- Design
- RTL
- Synthesis
- Netlist
- Placement
- Layout
- OPC
- Mask
- Fab
- Silicon as fabbed
Compiler in more detail

Source-language program

Compiler front end

Compiler intermediate representation

Compiler back end

Machine-language program

C source

Abstract Syntax

C light

C minor

RTL

LTL

Linear

Mach

Assembly Lang.

Machine Language
Verification / analysis in more detail

Program verification / static analysis / software model checking

Source-language program

Relational spec.

Source-language program

Functional spec.
Many styles of software verification/analysis

Program verification / static analysis / software model checking

*Source-language program*
Software + hardware toolchain

Source program

SOFTWARE TOOLCHAIN

ISA

HARDWARE TOOLCHAIN

Fabbed chip
Risks relating to system compositions

**Inadequate requirements and architectures.** “If a program has not been specified, it cannot be incorrect; it can only be surprising.” (W.D. Young, W.E. Boebert, and R.Y. Kain, 1985)

**Poor software [& hardware] engineering.** The absence of sensible abstraction, modular encapsulation...; use of riskful programming languages, undisciplined programming practices, and unwary use of analysis tools.

**Multiparty incompatibilities.** ... incompatibilities among interface assumptions, the existence of proprietary internal and external interfaces ...

**Scalability issues.** Composability can create scalability problems...

**Policy composability.** Policies for security, integrity, privacy, and safety components (especially when incomplete) often cannot compose without contradictions and unrecognized emergent properties.

**Protocol composability.** Network and cryptographic protocols are a source of risks... often ignore denial-of-service attacks.

**Assurance composability.** ... if the properties are not composable, the analysis results probably aren't either.

**Certification composability.** Deriving system certifications from the components is also fraught with hidden problems.
I. THE PROBLEM

II. HOW TO SOLVE IT
(Lessons from 21st-century software verification research)
The bad old days

Program verification / static analysis / software model checking

**Source-language program**

```
gcc
```

A million lines of code, not verified, not even specified, bugs here and there.

**Machine-language program / Instruction-set Architecture**

Gap between chip-level functional spec and RTL design

**Register-transfer language**

Layout tools:
a million lines of code, not verified, not even fully specified...

**Fabbed chip**
A million lines of code, not verified, not even specified, bugs here and there.
Not quite a solution
(even though it might be nice to do)

Source-language program

Nicer compiler

gcc

Half a million
A million lines of code, not verified,
not even specified,
bugs here and there.

Machine-language program
Specification + Verification

Source-language program

**Source-language program**

**compiler**

**Machine-language program**

- Specification of Source language
- Specification of machine lang. (ISA)
- Proof that compiler preserves semantics
- Proof checker
This is only interesting if the TRUSTED BASE (red part) is much smaller than the IMPLEMENTATION (blue part).

The blue part you don't have to trust; the red part guarantees that there are no bugs in here.
Developments in software toolchain verification, 1993-2013

- How to **specify**: Operational Semantics
- How to **prove**: simulations
- How to **program**: Verifiable compilers
- How to **machine check**: Logics, proof assistants

- **Compiler**
  - Specification of source language
  - Specification of machine lang. (ISA)
  - Proof that compiler preserves semantics
  - Proof checker

- **Machine-language program**
My own research, recently

Verified
Software
Toolchain

Andrew W. Appel, Lennart Beringer, Robert Dockins, Josiah Dodds, Aquinas Hobor, Gordon Stewart

Research supported by:

Princeton University

AFOSR

NSF

DARPA
Verified Software Toolchain

CompCert verified optimizing C compiler

Xavier Leroy, Sandrine Blazy, Jean-Baptiste Tristan, Zaynah Dargaye, Sylvan Rideau ...

Relaxed-Memory Concurrency
Peter Sewell, Francesco Zappa Nardelli, Jade Alglave, Susmit Sarkar, Scott Owens... Richard Bornat, Aquinas Hobor

VERIFIED HARDWARE TOOLCHAIN?
Correctness proof of toolchain component is difficult; for *legacy components* it’s practically impossible . . . What to do?

(especially as some of you in the audience may have and sell legacy components! )
Alternative: Translation validation

Sometimes possible to fit into legacy components

- Source program
- compiler
  - Proof witness about this program
  - Specification of Source language
  - Object program
- Witness checker
  - Proof that the witness checker is sound
  - Specification of machine lang. (ISA)
- Proof checker

Sometimes possible to fit into legacy components
I. THE PROBLEM
II. HOW TO SOLVE IT
   (Lessons from 21st-century software verification research)

III. THE VERDICT
   (For the software toolchain, remarkable success in the last 15 years)
Next two talks in this panel

David Pichardie
Research Scientist, Harvard Univ. + INRIA

Will describe the CompCert verified optimizing C compiler, built/proved by Xavier Leroy et al.

Peter Sewell
Professor, Univ. of Cambridge

Will describe adventures in specifying instruction-set architectures, (esp. but not limited to weakly consistent memory models)
Can this methodology be applied to the hardware synthesis toolchain?

How to **specify**: interface languages between components

How to **prove**: simulations

How to **program**: Verifiable components

How to **machine check**: Logics, proof assistants