Modular Development of Certified System Software

Zhong Shao
Yale University

July 19, 2010

USTC 2010 Summer School on Dependable Software
The code safety problem

Please install and execute this.

Credits: slides 2-16 taken from Peter Lee’s PCC Lipari lecture
Code safety

Is this safe to execute?
Approach 1
Trust the code producer

Trust is based on personal authority, not program properties

Scaling problems?
Approach 2
Java

Trusted Host

Verifier

Interp/JIT

CPU

Code

Limited in expressive power

Expensive and/or big
Approach 3
Formal verification

Theorem Prover

CPU

Flexible and powerful.
But really really hard and must be correct.

Trusted Host

Code
A key idea: explicit proofs
A key idea: explicit proofs

No longer need to trust this component.
Proof-carrying code
[Necula & Lee, OSDI'96]

Formal proof or “explanation” of safety

Typically native or VM code
Proof-carrying code

Reasonable in size (0-10%).

Simple, small (<52KB), and fast.

No longer need to trust this component.
Overview of the PCC approach

Please install and execute this.

OK, but let me quickly look over the instructions first.

Code producer

Host
Overview of the PCC approach

**Code producer**

**Host**
Overview of the PCC approach

This store instruction is dangerous!

Code producer

Host
Overview of the PCC approach

Can you prove that it is always safe?

Code producer

Host
Overview of the PCC approach

Yes! Here’s the proof I got from my certifying Java compiler!

Can you prove that it is always safe?

Code producer

Host
Overview of the PCC approach

Your proof checks out. I believe you because I believe in logic.
The role of programming languages

Civilized programming languages can provide “safety for free”.

- Well-formed/well-typed $\Rightarrow$ safe.

Idea: Arrange for the compiler to “explain” why the target code it generates preserves the safety properties of the source program.
The main roadmap

How to “reason at all”?

- Mathematical logic
- How to represent props and proofs
- Hacking inside the Coq proof assistant

How to reason about programs?

- How to define a programming language
- How to write the specification of a program and verify its correctness
Outline of this tutorial

• Introduction and problem definition
• The OCAP framework
• Certifying low-level sequential code
• Certifying garbage collectors and their mutators
• Certifying low-level concurrent code
• Conclusions and future work
Motivation

• Software is getting more complex but not more reliable
  – are we losing control?
  – don’t know how to make it better …
  – we pay huge costs on debugging, testing, & maintenance

• [Lamport03]: should computer software be considered as *a mathematical system* or *a biological system*?
Motivation (cont’d)

• **Current focus in industry:** cutting these costs via semi-automatic program checking & testing
  – It works on legacy code
  – Excellent progress has been made!

• **But it does not solve all the problems:**
  – can’t find all the bugs; no real guarantee
  – don’t support complex programs w. subtle bugs
  – don’t know how to fix the bug even if we find it,
    … or fixing it could lead to new bugs

• **Software will get more sophisticated (e.g., multicore)**
Now think about security!

• Existing computer system is not secure.

• Huge investment but no real good progress!

• One single bug can kill your system!
  – See Sam King’s virtual machine (subvirt) attack [King06]
  – Can only be avoided if we control the lowest-level core software

• Even with good source code, the binary code is still not trustworthy!
  – see Ken Thompson’s 1984 Turing award lecture
How to get things under control?

We need a “certified” computing platform!

We need firm control of the lowest-level software!

legacy code (not certified) becomes second-class citizen!
Fine, but can it be done?

**CW says:** *we will never see “bug-free” software!*

- Many valid reasons:
  - Specifications (*constantly evolving thus nobody knows*)
  - Economics (*won’t make money*)
    - Time to market
    - Cost --- just not worth it!
  - Complexity (*check out the existing OSes*)
The real questions

• **Specifications**
  – The term “bug-free” is always relative to the specs
  – The “specs” might contain bugs but …
  – Fairly stable for systems libraries, OS, runtime sys.

• **Economics**
  – Is there a market for truly reliable software?
  – How much are you willing to pay?
  – Cost of maintenance? training?
  – Interesting data from *Praxis Critical Systems*

• **Complexity**
  – Is it partly because we didn’t take a principled approach?
Problem definition

- **Hardware**
  - CPU, memory, hard disk, devices, ...

- **Software**
  - bootloader, device driver, OS, applications, ...

- Need a mathematical *proof* showing that *as long as the hardware works, the software always work according to its specification*
What needs to be done?

• Formalize the **hardware**
  – Only the hardware-software interface if we trust the hardware

• **Software** is just a list of binary machine instructions and constants

• Given the machine definition, the behavior of software is rigorously defined

• How to write the proofs & specs?

specification $S$

binary code $C$

formal proof $P$
Is this just program verification?

program verification

• Rich history --- major credit to Floyd/Hoare circ 1968’s
  – *Programs are mathematical objects!*

• Focus on source code

• Full or partial correctness

• Under attack in 1980’s
  – “It will never work!”
    [DeMilloLiptonPerlis77]

• More recent initiatives: grand challenges on verified software
“Why prog. verification will fail!” [DLP77]

- Mathematical proofs are checked by human via a social process --- software will require the same
- Even refereed mathematical proofs may contain bugs
- Proofs about computer programs will be large and tedious --- they will have loopholes
- What is the point of writing such proofs when they cannot be rigorously checked?
- Very controversial (see the book [MacKenzie01])
Is this just program verification?

**program verification**
- Rich history --- major credit to Floyd/Hoare circ 1968’s
  - *Programs are mathematical objects!*
- Focus on source code
- Full or partial correctness
- Under attack in 1980’s
  - “It will never work!” [DeMilloLiptonPerlis77]
- More recent initiatives: grand challenges on verified software

**certified software**
- Originated from PCC [Necula97]
  - Made more ambitious by us
- Focus on low-level code
  - But proofs can be done at high-level and then propagated down via a certifying compiler
- From type safety to any safety & liveness properties
- Many new hammers
  - machine-checkable proofs
  - type-based techniques
  - much better understanding of PL & compilers & proof assistants
What has changed since 1977?

• Proof checking now done automatically by machine
  – Curry-Howard-deBruijn correspondence [first published paper in 1980]
  – Can be made extremely reliable
  – Proof assistants with explicit proof objects
  – Only become mature & popular in late 1990’s

• Proof construction = writing ML-like functional programs
  – Benefit from usual software engineering methodologies
  – No need to run (just typecheck)!
  – Large & tedious proofs can be partially automated

• Advances such as type system, separation logic(!), rely-guarantee --- more synergy with PL is coming
Components of a certified framework

- certified software (proof + machine code)
- machine model
- safety & security policy
- mechanized meta-logic
- proof checker

Must be Trusted!
What is a good mechanized meta-logic?

You’d better be very paranoid!

• The logic must be “rock-solid”, i.e., consistent!

• The logic must be expressive
  to express everything a programmer might want to say

• Support explicit, machine-checkable proof objects

• The logic must be simple
  so that the proof checker can be hand-verified

• Can serve as logical framework and meta-logical framework
  to allow one to prove using specialized logics

• Compatible with “automated proof construction”
Proof assistants: a summary

• None of the existing ones are completely satisfactory:
  – Coq
  – Twelf/LF
  – HOL/Isabelle
  – ACL2
  – PVS

• We use Coq

• **Open problem**: how to combine proof assistant with general-purpose programming?
  – Mixing “type-safe” tactics with explicit proof objects
Writing proofs: is it practical?

• Modeling the bare machine is OK
  – Only care about the programmer’s interface
  – Simple transition relation between states
  – More challenging for modeling the complex devices & human interaction

• Proofs can be partially automated:
  – Certifying/certified compilers for ML/Java/C#/Cminor
  – High-level tools & theorem provers (e.g., Spec#, JML, Z3)

• Proof engineering = software engineering
  – informal proofs often exist (good hackers know what they are doing)
  – no real deep math in low-level system code(?)
Many perspectives

• Incredible synergies btw multiple disciplines
  – The hardware & VM perspective
  – The OS perspective
  – The PL perspective
  – The Compiler perspective
  – The programmer’s perspective

• Key questions
  – Can we do it at all?
  – Can we do it in a clean & smart way
  – Does it really scale? (no need of heroic efforts)
The hardware & VM perspective

• Formalizing machine architecture
  – machine instruction semantics
  – finite precision integers & floats
  – TLBs, paging & various IO devices

• Embedded devices
• Multicore and multiprocessors
• Relaxed memory models
• VMM & hypervisors
• Distributed share-memory
• Cloud computing
The OS perspective

• Moving up the OS hierarchy
  – bootloader
  – scheduler, virtual memory, interrupt handling
  – low-level concurrent code (locks & lock-free code)
  – file systems (disks, persistence, caches)
  – IPC & device drivers
  – information flow & security policies

• Runtime system issues
  – garbage collectors & VM

• seL4 (secure embedded L4 by UNSW) --- SOSP09 best paper
• Verve (MSR Redmond) --- PLDI10 paper
The compiler’s perspective

• After certifying a C program, how to make sure that the code generated from its compiler is still good?

• Moving up the compilation hierarchy
  – assembly code with C memory model
  – instruction scheduling & register allocation
  – compiler optimizations & relaxed memory models
  – intermediate code & compiler front-end

• CompCert by Leroy, INRIA (POPL06)
• PEG+ by Lerner, UCSD (PLDI10)
• Certified vs certifying compiler
The PL perspective

• Certifying different language features
  – procedure calls, stacks, exception handling, threads
  – dynamic linking & loading & self-modifying code
  – pessimistic vs optimistic concurrency
  – static vs dynamic type systems
  – object-oriented features & persistence
  – functional programs

• Certified linking of heterogenous components

• Trace-based semantics (safety & liveness)

• Information flow & security properties
The programmer’s perspective

• “Correctness-by-construction” vs. post-hoc verification

• New PLs & tools for writing certified programs
  – Old PL + pre-&post- conditions (JML, Spec#)
  – Proof assistants (Coq, Isabelle, Ynot, …)
  – Automated provers (Z3, Boogie,…)
  – Certifying PL (e.g., VeriML)

• Proof engineering = software engineering
  – informal proofs often exist
  – no deep math in low-level system code

• Proof by validation (i.e., insert dynamic checks!)
How to scale?

Modularity is the key!

- specification $S_1$ with binary code $C_1$ and formal proof $P_1$
- specification $S_2$ with binary code $C_2$ and formal proof $P_2$
- specification $S_3$ with binary code $C_3$ and formal proof $P_3$
- specification $S_4$ with binary code $C_4$ and formal proof $P_4$
- specification $S_5$ with binary code $C_5$ and formal proof $P_5$
- specification $S_6$ with binary code $C_6$ and formal proof $P_6$

Linking

specification $S$ with binary code $C$ and formal proof $P$
How to scale (cont’d)

• One logic for all code
  – Consider all possible interactions.
  – Very difficult!

• Reality: domain-specific logics
  – Only limited combinations of features are used.
  – It’s simpler to use a specialized logic for each combination.
  – Interoperability between logics

For each DSL, use as much automation as possible!
How to scale (cont’d)

Modularity on the specs/proofs as well!

- Specs can be larger than programs (thus buggy)
- Each module should only specify things it touches — “local reasoning”
- Data structures used internally should not be exported outside — “information hiding”
How to scale (cont’d)

Some bugs in specs are OK, as long as we can still prove what we want

specification $S$ 

binary code $C$  formal proof $P$

implies

dependability claim $SP$

binary code $C$  formal proof $P'$
How to scale (cont’d)?

*Software is often organized as a “stack” of abstractions!*
*Most can be certified at a much higher abstraction layer!*

```
certified L5 SW
certified L4 SW
certified L3 SW
certified L2 SW
  certified L1 software
```

Must accurately specify & certify all these interfaces!
Case study: a Mini-OS

How to certify the code?
Certifying the Mini-OS

1300 lines of code

- bootloader
- scheduler
- timer int. handler
- thread lib: spawn, exit, yield, …
- sync. lib: locks and monitors
- keyboard driver
- keyboard int. handler
- …

Many challenges:

- Code loading
- Low-level code: C/Assembly
- Concurrency
- Interrupts
- Device drivers / IO
- Certifying the whole system
  - Many different features
  - Different abstraction levels
Important questions to answer

1. Can we do it?

2. Can we do it in a clean way?
   (modularity, local reasoning, reusable, general, …)

3. Does it scale?
   (usability, support of automation, tool support)
Many new research problems

*How to certify common low-level lang. & OS abstraction?*

- mutable data structures
- general code pointers
- procedure call/return
- general stack-based control abstraction
- OS boot loader & self-modifying code
- garbage collection
- interrupt/signal handling
- device drivers and IO managers
- thread libraries and synchronization
- multiprocessor and memory model
- OS kernel/user abstraction

...........
The Keda-Yale HCS Center

- 中科大耶鲁高可信软件联合研究中心
  - 成立于2009年初
  - 网址：http://kyhcs.ustcsz.edu.cn
  - 中心备忘录由两校校长于2008年10月签署
  - 研究生交换计划备忘录签于2009年11月
    三位科大博士生（王伟，李勇，付明）现在耶鲁做科研

- 科研合作始于2005年
  - 目标是要做出世界一流的科研成果（we are doing it!）
  - 已合作发表3篇PLDI论文
    (科大合作学生: 郭宇，林春晓，李隆，项森)
The KYHCS Center (cont’d)

- 中心研究规模和方向正在日益扩大
  - 陈意云，邵中，张昱，华蓓，董群峰，。。。
  - 冯新宇博士(Xinyu Feng)将于今年夏天加盟中心
- many different research topics & projects
  
  - Compilers, programming languages, formal methods
  - Operating systems, virtual machines and hypervisors
  - Concurrency, multi-core, embedded systems, security
  - Distributed systems, cloud computing, networking
  - Proof assistants, automated theorem-proving, logic

- Yale CS faculty with related interests:
  - Paul Hudak, Bryan Ford, Richard Yang, Michael Fischer, Joan Feigenbaum, ……

欢迎加盟！
A sample of current projects

- Certified OS kernels [郭宇, 蒋新宇, 王僖, Vaynberg]
- Certifying compiler [李兆鹏, 庄重]
- Automated theorem provers [李兆鹏, 庄重, Stampoulis]
- Software transactional memory [李勇, 付明]
- Opportunistic concurrency (lock-free) [付明, 李勇]
- Trace-based models [王伟, Vaynberg]
- VeriML – next-generation PL [Stampoulis]
- Relaxed memory models [Ferreira]
- Next-generation parallel languages
- Separation logic & information flow [Costanzo]

and many others (pointer logics, secure-comp, …)
Conclusions (cont’d)

- First successful attempt at verifying low-level code such as preemptive threads w. interrupts, garbage collectors w. mutators, self-modifying code.

- Domain-specific logics + OCAP can go a long way

- Many open questions
  - what features should go into OCAP?
  - the value of higher-order closures & higher-order references?
  - local reasoning vs. frame rules?
  - simpler semantic models of states
  - weak reference vs. CSL ownership transfer?

- Other ongoing & future work
  - Certified mini-OS still on-going, more realistic ones in the future
  - Concurrency w. relaxed memory models & STM
  - How to make it scalable? (better proof assistant, more automation and reuse, use of high-level code + certified/certifying compiler)

Acknowledgement: Xinyu Feng, Yuan Dong, Yu Guo; Hongxu Cai, Rodrigo Ferreira, Andrew McCreight, Zhaozhong Ni, Alex Vaynberg, Chunxiao Lin, Long Li,