Fully Countering Trusting Trust through Diverse Double-Compiling (DDC)

David A. Wheeler

October 26, 2009

This presentation contains the views of the author and does not necessarily indicate endorsement by IDA, the U.S. government, or the U.S. DoD.

Outline

1. Introduction
2. Background
3. Description of threat
4. Informal description of DDC
5. Formal proof (skip assumption details for now)
6. Methods to increase diversity
7. Demonstrations of DDC (Tinycc, Lisp, GCC)
8. Practical challenges
9. Conclusions and ramifications

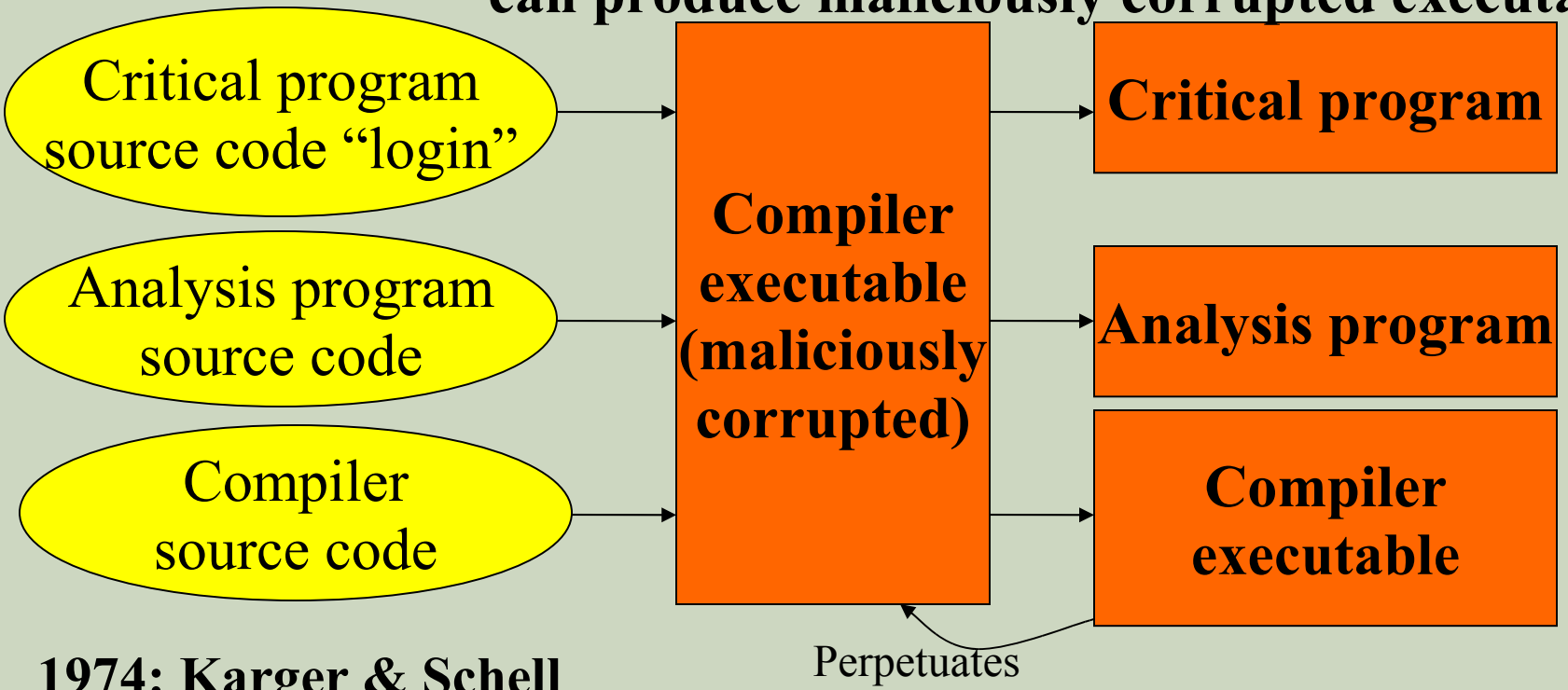
1. Introduction: Definitions

- **Executable: Data that can be directly executed by a computing environment. Aka "binary"**
- **Source code aka source: Data that is a representation of a program that can be translated into an executable; typically human-readable**
- **Compiler: An executable that translates source code into an executable**
- **Corrupted executable: An executable that does not correspond to its putative source code. An executable e corresponds to source code s iff execution of e always behaves as specified by s when the execution environment of e behaves correctly**
- **Maliciously corrupted executable: Intentionally-created corrupted executable [was: malicious executable]**

Trusting trust attack

Trustworthy source…

can produce maliciously corrupted executables



1974: Karger & Schell

1984: Ken Thompson. Demo'd (inc. disassembler), undetected

Fundamental security problem

Definition of “trusting trust” attack

Trusting trust attack: An attack in which the attacker attempts to disseminate a compiler executable that produces corrupted executables, at least one of those produced corrupted executables is a corrupted compiler, and the attacker attempts to make this situation self-perpetuating

- **May insert other Trojan horse(s) – software that appears to the user to perform a desirable function but facilitates unauthorized access into the user’s computer system**

Problem Importance

- "Trusting trust" has been treated as if it were a fundamental computer security "axiom"
  - Attack that "can't" be countered
  - Decades of no adequate solution
  - "Computer security is hopeless"
- Attackers have incentive to use it at some point if uncounterable
  - Huge benefits: possibly control nearly all computers worldwide?
  - Risks low: undetectable (til now!)
  - Costs often low...medium (vary by circumstance)
    - Even if costs were high, to some it'd be worth it
- Irrational to trust computers unless resolved

Dissertation Thesis

The trusting trust attack can be detected and effectively countered using the "Diverse Double-Compiling" (DDC) technique, as demonstrated by:

1. **a formal proof that DDC can determine if source code and generated executable code correspond**
2. **a demonstration of DDC with three compilers (a small C compiler, a small maliciously corrupted Lisp compiler, and a large industrial-strength C compiler, GCC)**
3. **a description of approaches for applying DDC in various real-world scenarios, including a description of how to scale DDC up to entire operating systems**

2. Background

- Karger & Schell 1974
- Ken Thompson 1984
- "Simple" solutions ineffective
  - Manual comparison impractical (size, change)
  - Interpreted languages—merely moves attack
- Draper 1984: "Paraphrase" compiler could filter (but have no way to confirm)
- McDermott 1984: Paraphrase compiler could be reduced-function or with irrelevant functions
- Proof of compiler correctness

Compiler bootstrap test

- "Compiler bootstrap" test is a common test for detecting compiler errors
  - Formally described in Goerigk 1999
- If c(s,e) is the result of compiling source s using compiler executable e, m is a correct "bootstrap" compiler, m0=c(s,m), m1=c(s,m0), m2=c(s,m1), all compilations terminate, m0 and s are both correct and deterministic, and the underlying hardware works correctly, then m1=m2 (passes the test, increasing confidence that s is correct)
- If m1≠m2, an assumption is wrong (fails the test)
  - Often the wrong assumption is "s is correct", making this a helpful test
- A corrupted compiler can pass this test

3. Description of threat

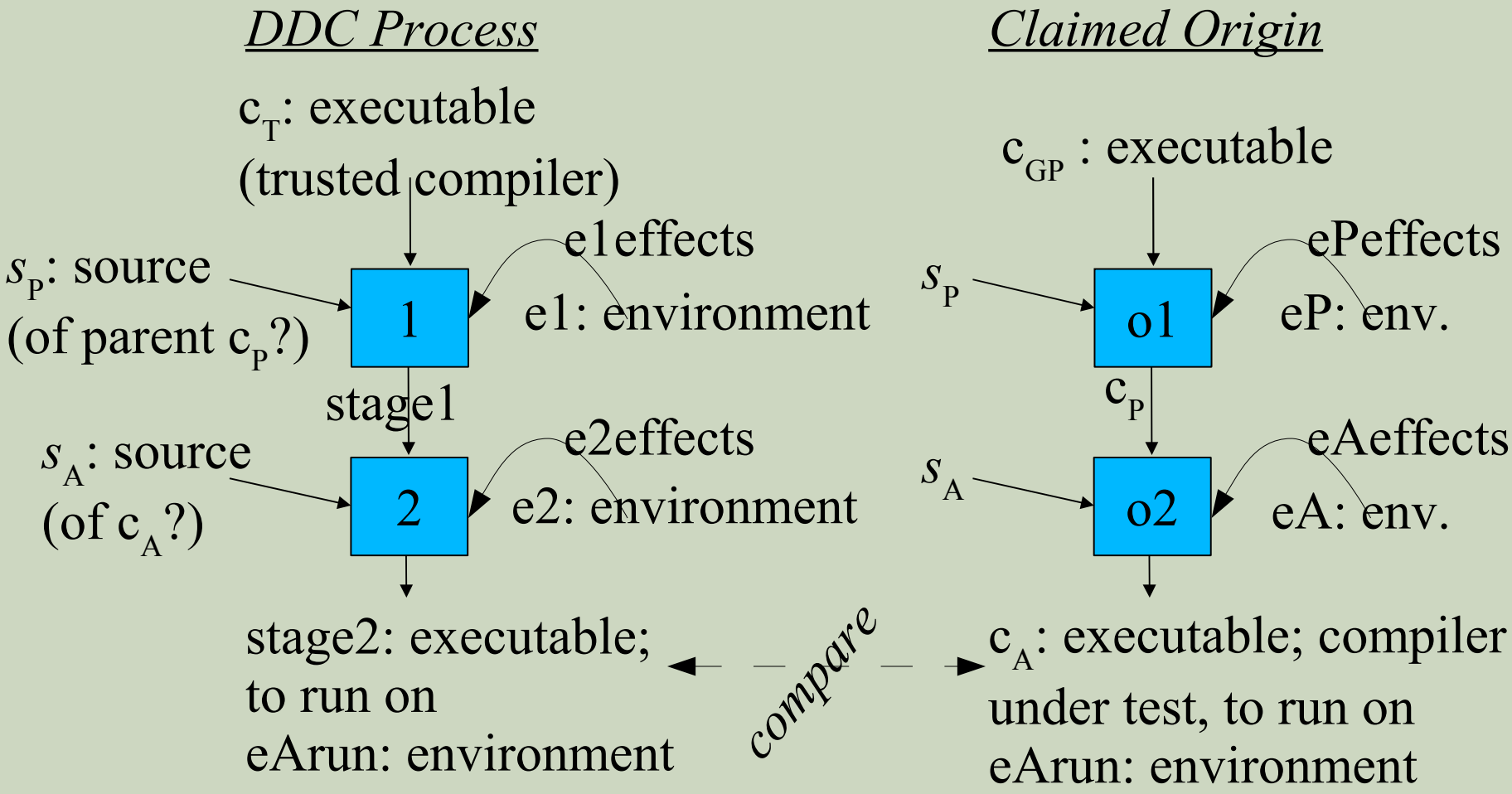
- **Attacker motivation**
  - **Successful "trusting trust" attack enables control of all systems compiled by that compiler**
  - **Until this work, essentially undetectable**
- **Attack depends on:**
  - **Trigger: Condition determined by an attacker in which a malicious event is to occur (e.g., when malicious code is to be inserted into a compiled program)**
  - **Payload: Code that performs the malicious event (e.g., the inserted malicious code and the code that causes its insertion)**
  - **Non-discovery: Victims don't detect attack**

4. Informal description of diverse double-compiling (DDC)

- **Idea created by Henry Spencer in 1998**
  - **Uses a different (diverse) trusted* compiler**
  - **Two compilation steps: Compile source of "parent" compiler, then use results to compile source of compiler-under-test**
  - **If DDC result bit-for-bit identical to compiler-under-test c_A, then source and executable correspond**
  - **Source code may include malicious/erroneous code, but now we can review source instead**
- **Before this work:**
  - **Never examined/justified in detail**
  - **Never tried**

* We will define "trusted" soon.

Diverse double-compiling (DDC)



Assumptions (informal)

- DDC performed by trusted programs/processes
  - Includes trusted compiler c_T, trusted environments, trusted comparer, trusted acquirers for c_A, s_P, s_A
  - Trusted = *justified confidence* that it does not have triggers and payloads that would affect the results of DDC. Could be malicious, as long as DDC is unaffected
- Correct languages (Java compiler for Java source)
- Compiler defined by s_P is deterministic (same inputs always produce same outputs)
- Do *not* assume that different compilers produce equal executables (they usually do not)

Special case: Self-parenting compiler

- If $s_P = s_A$, termed "self-parenting"
- My 2005 ACSAC paper explained DDC for this case
- Dissertation generalizes 2005 paper; DDC no longer requires it, it's simply a special case

Why not always use the trusted compiler?

- May not be suitable for general use
  - May be slow, produce slow code, generate code for a different CPU, be costly, have undesirable license restrictions, may lack key functions, etc.
  - In particular, a simple easily-verified compiler (with limited functionality & optimizations) could be used
- Using a *different* compiler greatly increases the confidence that source and executable correspond
  - Attacker must now subvert multiple executables and executable-generation processes to avoid detection
  - DDC can be performed multiple times, using different compilers and/or different environments, increasing difficulty of undetected attack

5. Formal proof

- ACSAC paper had only informal justification; dissertation provides a formal proof for DDC
- Classical First-Order Logic (FOL)
- Tools: Prover9 & Ivy
- Three proofs (once thought there'd be just one)
  - Proof 1: "If DDC produces the same executable as the compiler-under-test, then source code of the compiler corresponds to the executable"
  - Proof 2: "Under reasonable (benign) conditions, the compiler-under-test and DDC results *will* be the same" (thus, if they're different, something is wrong)
  - Proof 3: "Proof 2 assumption cP_corresponds_to_sP is *very* reasonable" (when there's a grandparent compiler)

Classical First-Order Logic (FOL)

-A	not A, ¬A. --A equivalent to A
A & B	A and B, A ∧ B
A | B	A or B, A ∨ B
A -> B	A implies B, A → B, if A then B, (-A)|B
all X ...	for all X … , ∀ X … ; notation is optional

Initial uppercase is variable, else constant

Examples

```
human(X) -> mortal(X). % "All humans are mortal."
human(socrates).       % "Socrates is human."
  % This is enough to prove:
mortal(socrates).      % "Socrates is mortal."
```

Tools: Prover9 & Ivy

- Prover9 accepts assumptions and goals in first-order logic (FOL)
  - If it can prove goal, outputs proof (by contradiction)
  - If can't, Mace4 tool can give counterexample
    - This may not make the issue clearer...!
  - DDC modeled using prover9 representation of FOL
- Ivy (separate tool) can verify prover9 proof
  - DDC proofs are ivy-verified!
  - Ivy is itself proved using ACL2
- Proofs also hand-verified (myself & co-workers)
- Thus, many reasons to believe that if the assumptions are true, the conclusions follow

Proof #1

- Proof #1 proves goal:
  - source_corresponds_to_executable
  - I.E., if DDC recreates the compiler-under-test, then the compiler source and executable correspond
- It requires 5 assumptions:
  - definition_stage1
  - definition_stage2
  - cT_compiles_sP
  - define_exactly_correspond
  - sP_compiles_sA
- This is the heart of DDC

Proof #1 Goal: source_corresponds_to_executable

(stage2 = cA) ->

exactly_correspond(cA, sA, lsA, eArun).

where predicate exactly_correspond(Executable, Source, Lang, RunOn) is true iff Executable exactly implements source code Source when interpreted as language Lang and run on environment RunOn.

definition_stage1 and definition_stage2

stage1 = compile(sP, cT, e1effects, e1, e2).

stage2 = compile(sA, stage1, e2effects, e2, eArun).

where compile(Source, Compiler, EnvEffects, RunOn, Target) represents compiling Source with the Compiler, running it in environment RunOn but targeting the result for environment Target. When Compiler runs, it uses Source and EnvEffects as input; EnvEffects models the inputs (data and timing) from the environment, which may vary between executions while still conforming to the language definition (e.g., random number generators, heap allocation addresses, thread exec order, etc.).

cT_compiles_sP

all EnvEffects accurately_translates(cT, lsP, sP, EnvEffects, e1, e2).

Trusted compiler cT is a compiler for language lsP, and it will accurately translate sP if run in environment e1, regardless of EnvEffects. cT targets (generates code for) environment e2.

Thus, you can't use a Java compiler to compile C (directly)!

You can use the random number generator, but the results have to be an accurate translation (even if it's not the same each time).

define_exactly_correspond

accurately_translates(Compiler, Lang, Source, EnvEffects, ExecEnv, TargetEnv) ->

exactly_correspond(

compile(Source, Compiler, EnvEffects, ExecEnv, TargetEnv),

Source, Lang, TargetEnv).

If some Source (in language Lang) is compiled by a compiler that accurately translates it, then the resulting executable exactly corresponds to the original Source.

exactly_correspond(Executable, Source, Lang, RunOn) iff Executable exactly implements source code Source in language Lang when run on RunOn.

sP_compiles_sA

accurately_translates(GoodCompilerLangP, lsP, sP, EnvEffectsMakeP, ExecEnv, TargetEnv) ->

accurately_translates(

compile(sP, GoodCompilerLangP, EnvEffectsMakeP, ExecEnv, TargetEnv),

lsA, sA, EnvEffectsP, TargetEnv, eArun).

Source sP (written in language lsP) must define a compiler that, if accurately compiled (by some GoodCompilerLangP), would be suitable for compiling sA.

Proof #2

- Proof #1 not useful if goal (equality) never occurs
- Proof #2 proves that equality *will* occur in a benign environment given reasonable assumptions:
  - Goal always_equal: cA = stage2.
- Requires 10 assumptions:
  - 4 same: definition_stage1, definition_stage2, cT_compiles_sP, define_exactly_correspond
    - Unused from previous: sP_compiles_sA
  - definition_cA
  - cP_corresponds_to_sP (see proof 3!)
  - define_compile
  - sP_deterministic
  - sP_portable
  - define_determinism

definition_cA and cP_corresponds_to_sP

cA = compile(sA, cP, eAeffects, eA, eArun).

exactly_correspond(cP, sP, IsP, eA).

The first follows from the figure.

The second is an assumption based on the figure. It is phrased this way so that no grandparent cGP is strictly required (perhaps the compilation was done by hand). Proof #3 will show how we can prove this is true if there *is* a grandparent cGP.

define_compile

compile(Source, Compiler, EnvEffects, RunOn, Target) =
extract(converttext(run(Compiler, retarget(Source, Target), EnvEffects, RunOn), RunOn, Target)).

For proof #2, we need more information about compilation, and we must deal with potentially-different text encodings.

To compile, retarget source for target, then run Compiler, input Source, on RunOn. Convert text format from "RunOn" to "Target", and extract only executables.

sP_deterministic and sP_portable

deterministic(sP, lsP).

portable(sP, lsP).

Source sP is deterministic. I.E., it avoids all non-deterministic capabilities of language lsP, or uses them only in ways that will not affect the output of the program

Similarly, sP is portable; it only uses the portable constructs of lsP.

define_determinism

(deterministic(Source, Language) &

portable(Source, Language) &

exactly_correspond(Executable1, Source, Language, Environment1) &

exactly_correspond(Executable2, Source, Language, Environment2))

-> (converttext(run(Executable1, Input, EnvEffects1, Environment1), Environment1, Target) = converttext(run(Executable2, Input, EnvEffects2, Environment2), Environment2, Target)).

If source code deterministic & portable, and two executables both exactly correspond to it, then those executables - when given the same input - produce the same output when run on their respective environments (convert text to Target format).

Example of define_determinism

- **Given this portable C source:**

```
#include <stdio.h>
main() {
   (void) printf("%d\n", 2+2);
}
```

- **And two different properly-working deterministic C compilers processing this portable code:**
  - **Resulting executables will usually differ**
  - ***Running* those executables produces the same results (modulo character encoding)**

Proof #3

- Proof #3 proves cP_corresponds_to_sP when there's a grandparent (common case) & benign circumstances:
  - exactly_correspond(cP, sP, lsP, eA).
- This was an assumption of proof #2
  - Separately-proved assumption so we don't *have* to have a grandparent cGP
- Requires 3 assumptions:
  - 1 reused: define_exactly_correspond
  - definition_cP
  - cGP_compiles_sP

definition_cP and cGP_compiles_sP

cP = compile(sP, cGP, ePeffects, eP, eA).

all EnvEffects accurately_translates(cGP, lsP, sP, EnvEffects, eP, eA).

These are just like definition_cA (follows from figure) and cT_compiles_sP.

6. Methods to increase diversity

- To gain justified confidence in the trusted compiler c_T & the DDC environments we could perform a complete formal proof of them... but this is difficult
- Another, often simpler method is diversity:
  - Diversity in compiler implementation
  - Diversity in time (e.g., c_T developed long before)
  - Diversity in environment
  - Diversity in source code input (mutated source)
    - Semantics-preserving mutations
    - Non-semantics-preserving mutations

7. Demonstrations of DDC

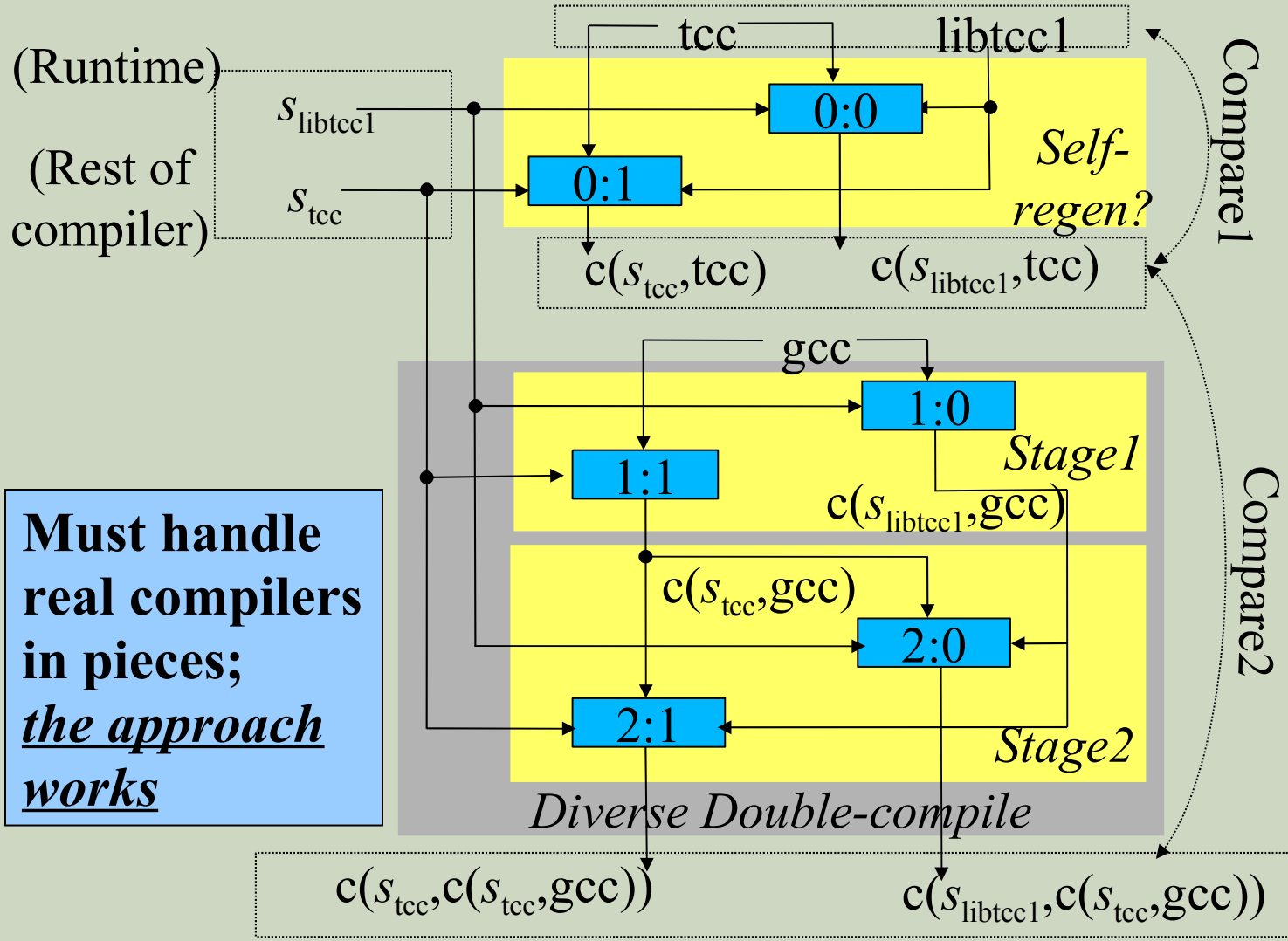
- tcc (TinyCC): C compiler (ACSAC paper)
- Goerigk Lisp compilers (including a malicious compiler)
- GCC (scales up)

tcc

- Performed on small C compiler, tcc (ACSAC)
  - Separate runtime library, handle in pieces
- tcc defect: fails to sign-extend 8-bit casts
  - x86: Constants -128..127 can be 1 byte (vs. 4)
  - tcc detects this with a cast (prefers short form)
  - tcc bug – cast produces wrong result, so tcc compiled-by-self always uses long form
- tcc junk bytes: long double constant
  - Long double uses 10 bytes, stored in 12 bytes
  - Other two "junk" bytes have random data
- Fixed tcc, technique successfully verified fixed tcc
- Used verified fixed tcc to verify original tcc

It works!

Diverse double-compilation of tcc



Must handle real compilers in pieces; *the approach works*

Goerigk Lisp compilers

- Pair of Lisp compilers, “correct” & “incorrect”
  - “Incorrect” implemented the trusting trust attack
  - Ported to Common Lisp
- DDC applied
  - “Correct” compiler compared correctly, as expected
  - Executable based on “incorrect” source code did *not* match the DDC results when DDC used the “correct” source code, as expected
    - “Diff” between results revealed that the “incorrect” executable was producing different results, in particular for a “login” program
    - Tip-off that executable is probably malicious

GCC

- GNU Compiler Collection (GCC) is widely-used compiler in industry – shows DDC scales up
  - Many languages; for demo, chose C compiler
- Used Intel C++ compiler (icc) as trusted compiler
  - Completely different compiler
- Fedora didn't record info to reproduce executable
- Created C compiler executable to capture all necessary data & use that as compiler under test
  - Chose GCC version 3.0.4 as compiler under test
  - "gcc" is a front-end that runs the real compiler programs; C compiler is actually cc1
  - Code outside of GCC (including linker, assembler, archiver, etc.) considered outside compiler

GCC (continued)

- **Challenges:**
  - **“Master result” pathname embedded in executable (so made sure it was the same)**
  - **Tool semantic change (“tail +16c”)**
  - **GCC did *not* fully rebuild when using its build process (libiberty library not rebuilt)**
- **Once corrected, DDC produced bit-for-bit equal results as expected**

8. Practical challenges

- **Limitations**
  - **Depends on confidence trusted compiler & DDC environment does not include triggers/payloads**
  - **DDC only applies to the specific executable under test; use cryptographic hashes to identify it**
  - **Source code may have malicious code; DDC shows that there is "nothing hidden in the executable"**
  - **If DDC result is different from compiler under test, at least one of proof #2's assumptions has been violated... but it may not be obvious which one(s)**
- **Non-determinism**
- **Difficulty in finding alternative trusted compilers**
- **Countering "pop-up" attacks**
  - **Re-run DDC on every executable release**

8. Practical challenges (continued)

- Multiple sub-components: See tcc
- Inexact comparison
- Interpreters/recompilation dependency loops
- Untrusted environment/broadening DDC application
  - Operating system+compiler as “compiler under test”
- Trusted build agents
- Application problems with current distributions
  - Inadequate information for DDC
  - Prelink, ccache
- Finding maliciously misleading code

Hardware

- BIOS, microcode: DDC still applies trivially
- DDC unnecessary to counter direct subversion of hardware components (not trusting trust attack)
- If hardware is subverted so it intentionally subverts the implementation process of other later hardware, that *is* a trusting trust attack
  - Harder – different level of abstraction
- DDC can apply in such cases, but to apply to hardware (e.g., integrated circuits):
  - Technical challenges: "Equality" operator. Perhaps scanning electron microscope or a tool that performed optical phase array shifting
  - Legal challenges: Difficult to legally obtain needed detailed information (much proprietary)

9. Conclusions and ramifications

- DDC shows that source and executable correspond. Executable may have errors or be malicious, but these can be found by examining source (easier than examining executable)
- DDC primarily useful to those who have access to the source code
- Potential future work: Recompiling whole OS, relax being "exactly equal"
- Possible policy implication: For compilers of critical software, require information to do DDC
- Trusting trust attack can be detected and effectively countered by DDC

Backup

Have the right things been proved?

- This can't be shown formally; must always be a "top" specification
- Many reasons to believe they are the right things:
  - Many have reviewed proofs & agree they're sensible
  - Based on informal justification in peer-reviewed ACSAC paper, which no one has refuted
  - Formalization process forced clarification that there were multiple claims to prove; suggests insight
  - All demonstration results explainable by proofs [added]
  - Proofs clearly fit together
    - #3: If benign + a grandparent, then cP_corresponds_to_sP
    - #2: If benign inc. cP_corresponds_to_sP, then stage2 = cA
    - #1: If stage2 = cA, then cA and sA correspond

What does DDC not address?

- Passing DDC process shows source and executable, *not* that executable non-malicious
  - Still useful – now you can focus on source review
- Less useful if source review can't find malicious code
  - Concern is "maliciously misleading" source code
  - Believe maliciously misleading can be countered, but fully addressing this is out of scope
  - DDC still helpful in countering unintentional error
- Determining if corruption is intentional (malicious)
- DDC results will differ if proof #2 assumptions do not hold... but finding out *why* can be difficult
  - Scale and complexity. Still, *can* do it, even with GCC

Contributions to the field

- "Trusting trust" is a serious, well-known attack
  - Considered unsolveable since 1974. "Paraphrase" & "recompile" approach *might* fix, *or* might introduce the problem
  - Could not accumulate non-corruption evidence
  - Henry Spencer had promising idea, but no evidence
- Dissertation establishes a useful countermeasure:
  - Describes process & assumptions in detail
  - Formally proves
  - Demonstrates it can be done (inc. large, malicious)
  - Explains how to expand & apply in varying cases
- *Major result* in computer security – crushes what was considered a fundamental vulnerability

What have I learned?

- **Formal methods & tools**
  - **Much greater understanding of specification languages and notations**
  - **Consider the "weakest" notation (more automatable)**
  - **Prefer a prover with verifier**
- **Debugging big compilers is painful**

Different compilers produce different binary outputs!

- Of course they do!
  - Can be x86 vs. PPC!
  - Don't directly compare; RUN them
- Binaries should produce the same output
  - If same source code compiled by two different compilers, & both given same input to run
  - E.G., for C, printf("%c", getc()) given input "Q" prints Q, even if different compilers!
  - If assumptions (other than "trusted") fail, matches fail (safe)

Key to large proof graphs

- Rectangles are assumptions
- Octagon is goal
- Rest are steps; arrows flow into uses
- Proof-by-contradiction
  - Negates goal, then shows that it leads to “always false” $F.

ACSAC paper: Big positive splash!

- Published *Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*, Dec 2005, "Countering Trusting Trust through Diverse Double-Compiling"
- Required reading in at least two Spring 2006 courses
  - Northern Kentucky University's CSC 593
  - GMU's IT 962
- Referenced in Bugtraq, comp.risks (Neumann's Risks digest), Lambda the ultimate, SC-L (the Secure Coding mailing list), LinuxSecurity.com, Chi Publishing's Information Security Bulletin, Wikipedia, OWASP
- Bruce Schneier's weblog and Crypto-Gram

Old slides

Problems unsolved with Fedora Core (and probably true in general)

- Lots of "Edison successes"
- Vendor does not record exact recompilation info
  - Compilers & libraries written by different people, unsync
  - "Good practice" says change one component at a time, and once it works don't change it (inc. its binary)
    - Different library binaries built at different times by different versions of compiler, inc. in-house versions
    - Exact order of recompilations *not* recorded, so no way to reproduce exactly what's distributed
    - Even if did, multi-week runs not helpful
  - Massive transitive dependencies
    - FC4 gcc build depends on XFree86, *not* on CD
    - Result: compiler depends on fontconfig (!)
- Without exact info, cannot regenerate compiler
  - Vendors won't capture info or change process until demo'd
- Solution: Simulate distribution to capture info

Can DDC be used with hardware?

- Probably; not as easy for pure hardware
- Requires 2nd implementation T
  - Alternative hardware compiler, simulated chip
- Requires "equality" test
  - Scanning electron microscope (surface) & scanning transmission electron microscope (STEM), use focused ion beam. *Maybe* use superposition to detect different phase changes (diffraction), though this may be *too* sensitive
  - Issue: Real chips *have* defects – false + issues
- Requires knowing exact correct result
  - Often cell libraries provided to engineer are *not* the same as what is used in the chip
  - Quantum effect error corrections for very high densities considered proprietary by correctors
- Only shows the chip-under-test is good

Can DDC be used with hardware?

- Probably; not as easy for pure hardware
- Requires 2nd implementation T
  - Alternative hardware compiler, simulated chip
- Requires "equality" test
  - Scanning electron microscope, focused ion beam
- Requires knowing exact correct result
  - Often cell libraries provided to engineer are *not* the same as what is used in the chip
  - Quantum effect error corrections for very high densities considered proprietary by correctors
- Only shows the chip-under-test is good

Threat: Trusting trust attack

- First publicly noted by Karger & Schell, 1974
- Publicized by Ken Thompson, 1984
  - Back door in “login” source code would be obvious
  - Could insert back door in compiler source; login's source is clean, compiler source code is not
  - Modify compiler to also detect itself, and insert those attacks into compilers' binary code
  - Source code for login and compiler pristine, yet attack perpetuates even when compiler modified
  - Can subvert analysis tools too (e.g., disassembler)
  - Thompson performed experiment - never detected

Fundamental security problem

Broader implications

- Practical counter for trusting trust attack
- Can expand to TCB, whole OS, & prob. hardware
- Governments could require info for evals
  - Receive all source code, inc. build instructions:
    - Of compilers: so can check them this way
    - Of non-compilers: check by recompiling
  - Could establish groups to check major compiler releases for subversion
- Insist languages have public unpatented specifications (anyone can implement, any license)
- Source code examination now justifiable