Proving that Diverse Double-Compiling (DDC) Counters Trusting Trust

David A. Wheeler

November 2, 2007

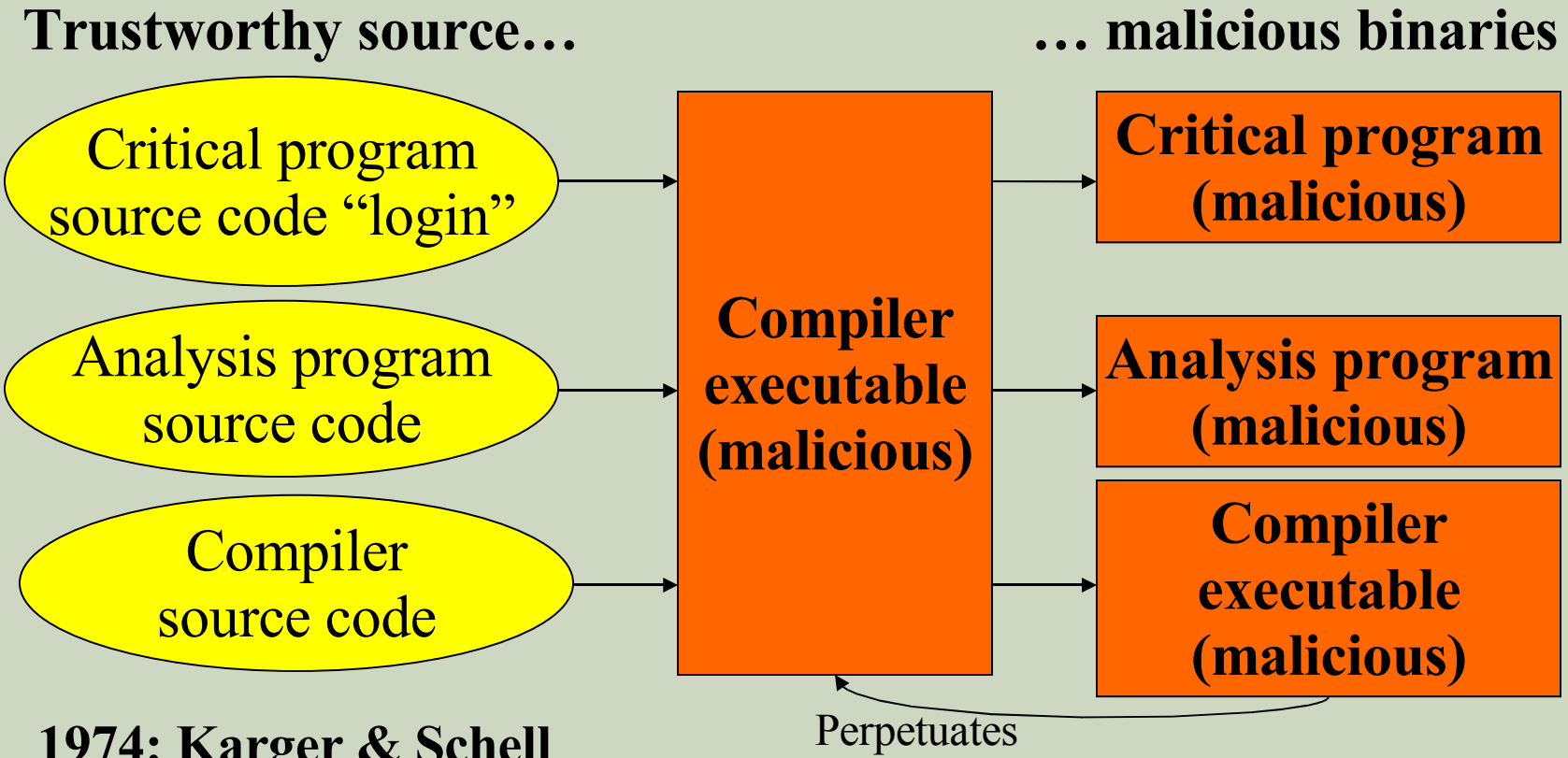
This presentation contains the views of the author and does not necessarily indicate endorsement by IDA, the U.S. government, or the U.S. DoD.

Outline

- Problem: Trusting trust attack (the problem)
- Claimed Solution: Diverse double-compiling (DDC)
- Lessons learned on formal methods tools
- Intro to Prover9/Ivy tools, inc. prover9 notation
- Review proof assumptions
  - Proof 1: "If DDC produces the same executable as the (potentially malicious) compiler-under-test, then source code of the compiler corresponds to the executable"
  - Proof 2: "Under reasonable conditions, the compiler-under-test and DDC results *will* be the same"
  - Proof 3: "Proof 2 assumption cP_corresponds_to_sP is *very* reasonable"

Your mission: Are the assumptions reasonable?

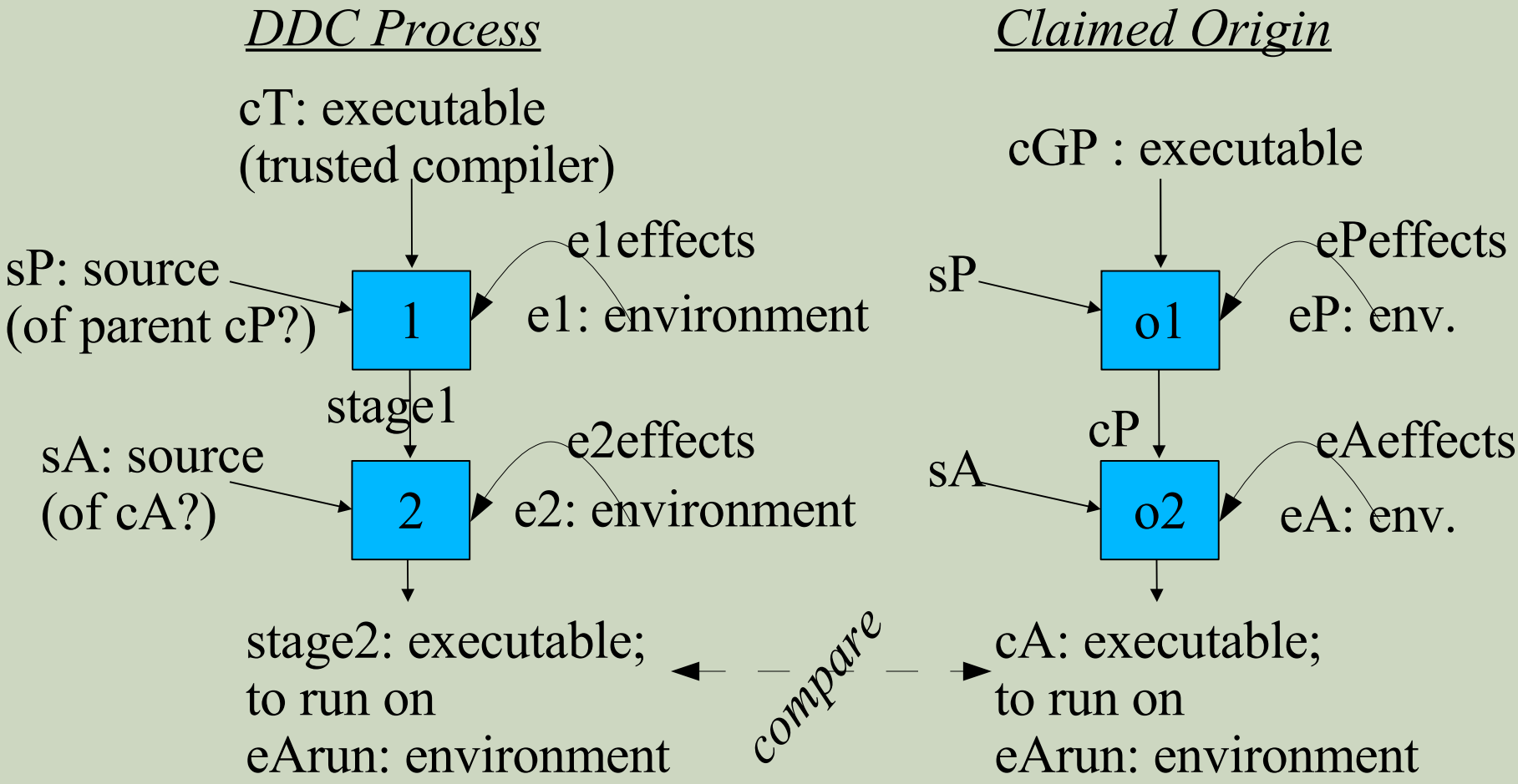
Problem: Trusting trust attack



1974: Karger & Schell
1984: Ken Thompson. Demo'd (inc. disassembler), undetected

Fundamental security problem

Solution: Diverse double-compiling



But does DDC really counter the attack?

- **Many doubt my textual arguments**
  - **"Trusting trust has been unsolved for decades, why do you think *you* can solve it?"**
- **So, go for gold standard: Formal mathematical proof**
  - **Have proof, & used tools to help create it**

Lessons learned about formal methods (FM) tools (1)

- **Originally did hand proofs, very painful**
  - **Mistakes, time-consuming (bookkeeping)**
- **Looked at a number of FM tools, e.g.:**
  - **ACL2: Good for proving *specific* programs, but weak for general circumstances & uses unreadable LISP notation**
  - **Alloy: Excellent for specs, but no absolute proof**
  - **Coq, Isabelle/HOL: Not really automated; proof *checkers***
  - **PVS: Used at first. Great language (types, etc.) - specs clean, and found missing assumption. Poor automation**
  - **E: Poor doc, related pieces unavailable**
  - **Prover9/mace4/ivy: Less capable language vs. PVS but *automatically* find proof/counter, + proved checker**
- **Good tools *really helped***

Lessons learned about formal methods (FM) tools (2)

- FM tools becoming more useful
  - Faster: Order-of-magnitude faster hardware, improved algorithms for automation, better implementation
  - Open source software (OSS) enabling much more sophisticated tools (building on other tools)
  - FM new approaches (esp. for automation)
    - Alloy: Give up guarantees for automation, ease-of-use
    - Prover9/Mace4/Ivy: Combine different approaches
- "Right tool for job" esp. important for FM
  - Currently "learn a bunch of them" necessary to select
- Great potential: Combining tools + automation
- In *some circumstances* really useful now
  - Significant learning curve

Toolsuite Selected: Prover9/Mace4/Ivy

- Prover9 accepts assumptions and goals in first-order logic (FOL)
  - If it can prove it, outputs proof (often < 1 sec!)
  - If can't, can ask Mace4 to give counterexample
  - DDC modeled using prover9 FOL
- Ivy (separate tool) can verify prover9 proof
  - DDC proofs are ivy-verified!
- But: garbage in, garbage out
- Your mission: Determine if assumptions & goals okay
  - If they are, then claims are proven

Prover9 Notation

-A	not A
A & B, A | B	A and B, A or B
A -> B	A implies B (if A then B)
all X ...	for all X, ... (optional)

Initial uppercase is variable; else constant

Example

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

Proof #1

- **Proof #1 proves goal:**
  - **source_corresponds_to_executable**
  - **I.E., if DDC recreates the compiler-under-test, the compiler source and binary 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 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 address values, 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 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, would be suitable for compiling sA.

Proof #2

- Proof #1 not useful if goal (equality) never occurs
- Proof #2 proves goal that normally it *will* occur when there's been no attack:
  - 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
- Proof split into lemmas (not shown)

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 grandfather cGP.

define_compile

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

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. This is more than needed (it only needs to port to certain environments in certain ways), but spec'ing more is complex and unnecessary. Often a developer will just use the portable subset of the language anyway.

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).

Proof #3

- **Proof #3 proves cP_corresponds_to_sP when there's a grandfather (common case):**
  - **exactly_correspond(cP, sP, lsP, eA).**
- **This was assumption of #2**
  - **Separately-proved assumption so we don't *have* to have a grandfather 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.

Conclusions

- If assumptions & goals valid, proved!
- See proof graphs for more
  - 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.

Backup

Attacker motivations

- **Huge benefits – Controlling a compiler controls everything it compiles**
  - **Controlling 2-3 compilers would control almost every computer worldwide**
- **Risks low – no viable detection technique**
- **Costs low...medium**
  - **Requires one-time write of trusted binary**
    - **Not necessarily easy, but *someone* can, one-time, & not designed to withstand determined attack**
  - **Even if costs were high, the power to control every computer would be worth it to some**

Triggers & payloads

- **Attack depends on triggers & payloads**
  - **Trigger: code detects condition for performing malicious event (in compilation)**
  - **Payload: code performs malicious event (i.e., inserts malicious code)**
- **Triggers or payloads can fail**
  - **Change in source can disable trigger/payload**
- **Attackers can easily counter**
  - **Insert multiple attacks, each narrowly scoped**
  - **Refresh periodically via existing compromises**

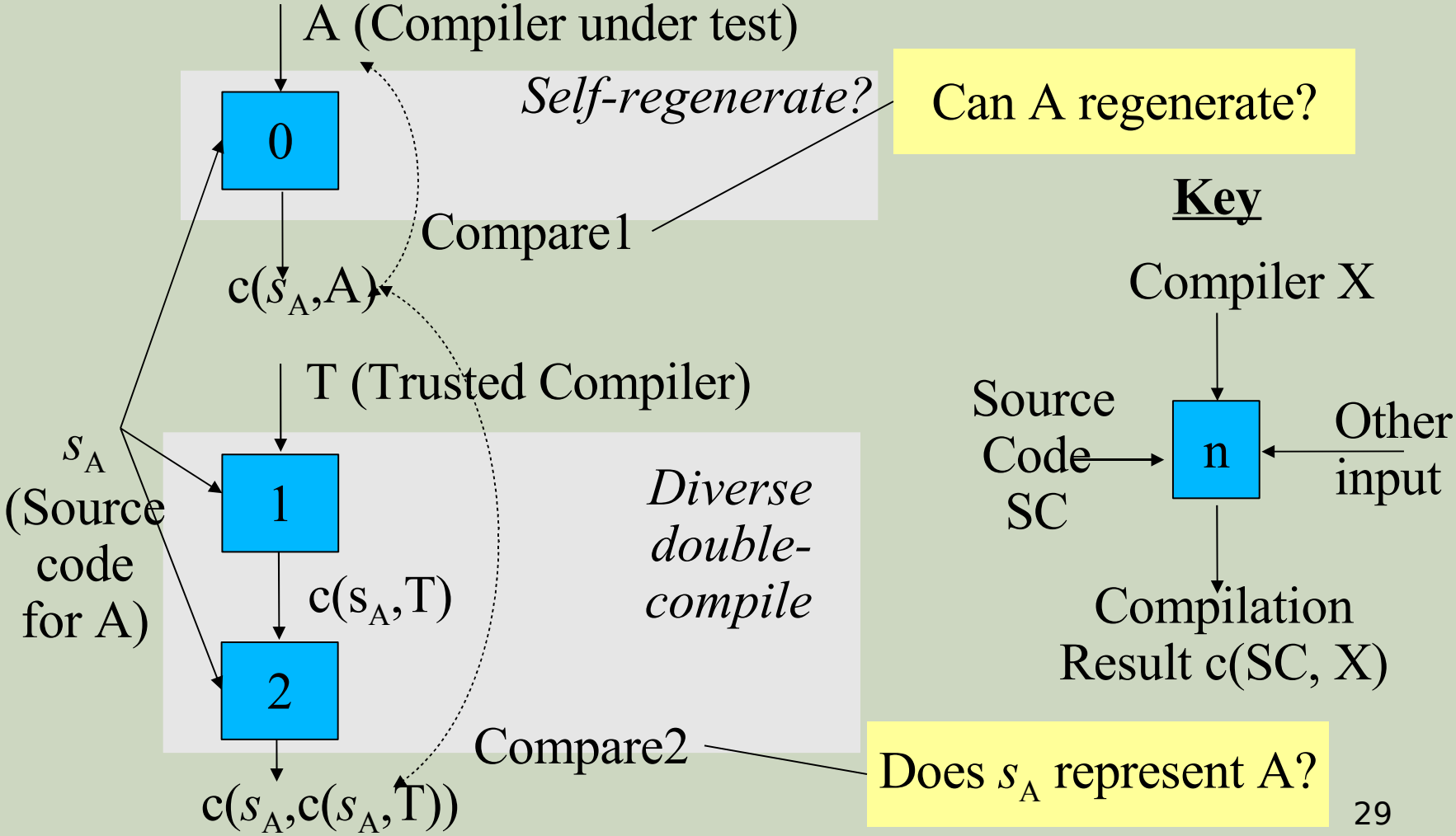
Inadequate solutions & Related work

- Manual binary review: Size, subverted tools
- Automated review / proof of binaries: Hard
- Recompile compiler yourself: Fails if orig. compiler malicious, massive diligence
- Interpreters just move attack location
- Draper/McDermott: Compile paraphrased source or with 2nd compiler, then recompile
  - Any who care must recompile their compilers
  - Can't accumulate trust – can still get subverted
  - Helps; another way to use 2nd compiler?

Solution: Diverse double-compiling

- Developed by Henry Spencer in 1998
  - Check if compiler can self-regenerate
  - Compile source code twice: once with a second "trusted" compiler, then again using result
  - If result bit-for-bit identical to original, then source and binary correspond
- Never described/examined/justified in detail
- Never tried

Diverse double-compiling in pictures



Why does it work?

Assuming:

1. **Have trusted: compiler T, DDC environment, comparer, process to get s_A & A**

   Trusted = triggers/payloads, if any, are different
2. **T has same semantics as A for what's in s_A**
3. **Flags etc. affecting output identical**
4. **Compiler s_A deterministic (control seed if random)**

Then:

1. **c(s_A,T) functionally same as A – same source code!**
2. **If A malicious, doesn't matter – never run in DDC!**
3. **Final result bit-for-bit equal iff s_A represents A – only an untainted compiler, with identical functionality, creates the final result!**

How to increase diversity

- Trusted Compiler T must not have triggers/payloads for compiler A
- Could prove T's binary – hard
- Alternative: increase diversity
  - Compiler implementation (maximally different)
  - Time (esp. old compiler as trusted compiler)
  - Environment
  - Source code mutation/paraphrasing

Practical challenges

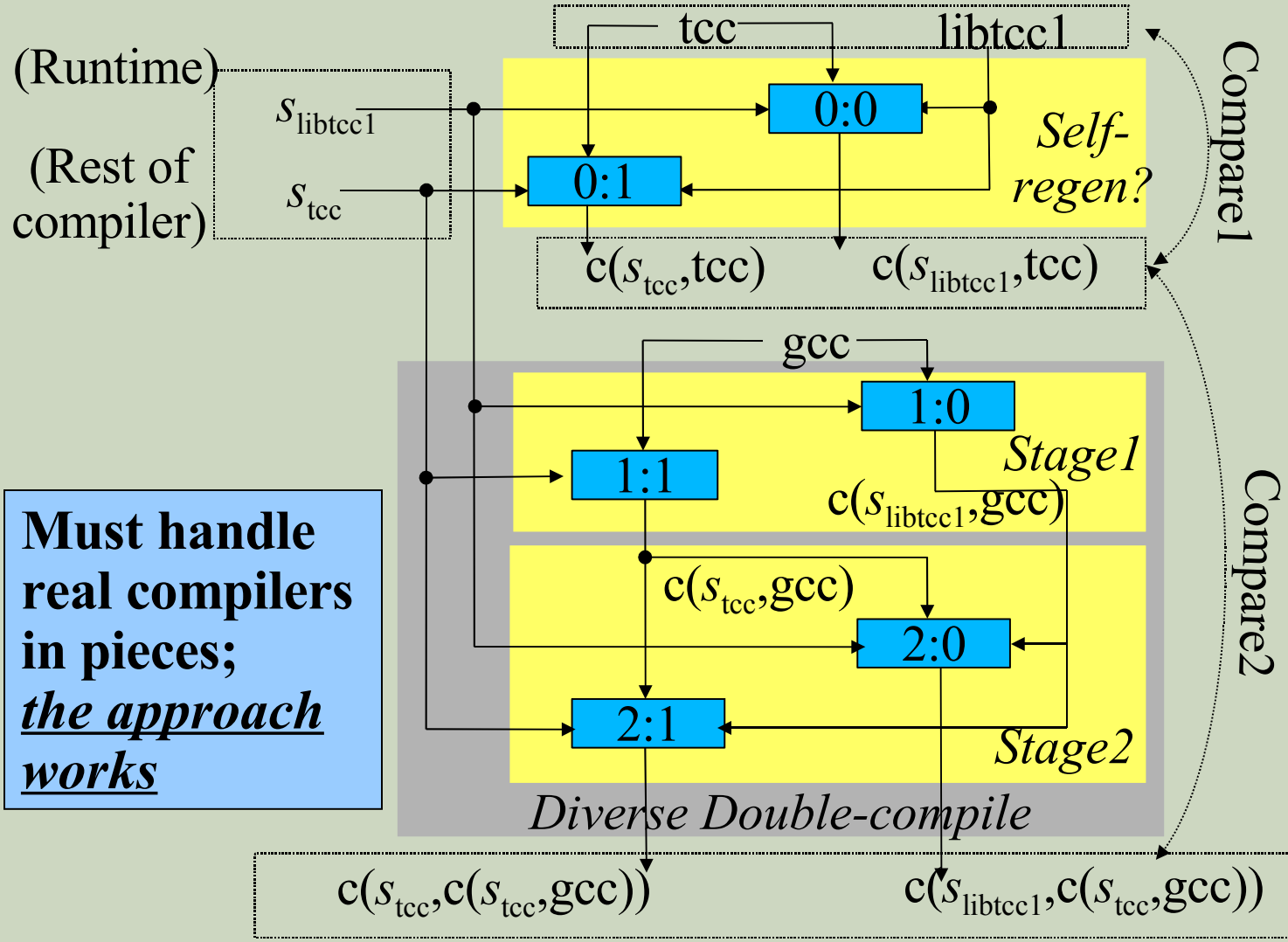
- Uncontrolled nondeterminism
- May be no alternative compiler that can handle *s*
  - Can create, or hand-preprocess
- "Pop-up" attack
  - Attacker includes self-perpetuating attack in only some versions (once succeeds, it disappears)
  - Defenders must thoroughly examine every version they accept, not just begin/end points
- Multiple compiler components
- Malicious environment? Redefine A as OS
- Inexact comparison (e.g., date/time stamp)

Demo: tcc

- Performed on small C compiler, tcc
  - Separate runtime library, handle in pieces
- tcc defect: fails to sign-extend 8-bit casts
  - x86: Constants -127..128 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*

Limitations

- Not absolute proof (unless T & environment proved)
  - But you can make as strong as you wish
  - Hard to overcome & can use more tests/diversity
- Only shows source & binary correspond
  - Could still have malicious code in source
  - But we have techniques to address that!
- A's source code must be available (easier for FLOSS)
- Source/binary correspondence primarily useful if you can see compiler source
- Not yet demonstrated on larger scale – doing that now
- Easier if language standard & no software patents
  - Visual Basic patent app for “isNot” operator

Broader implications

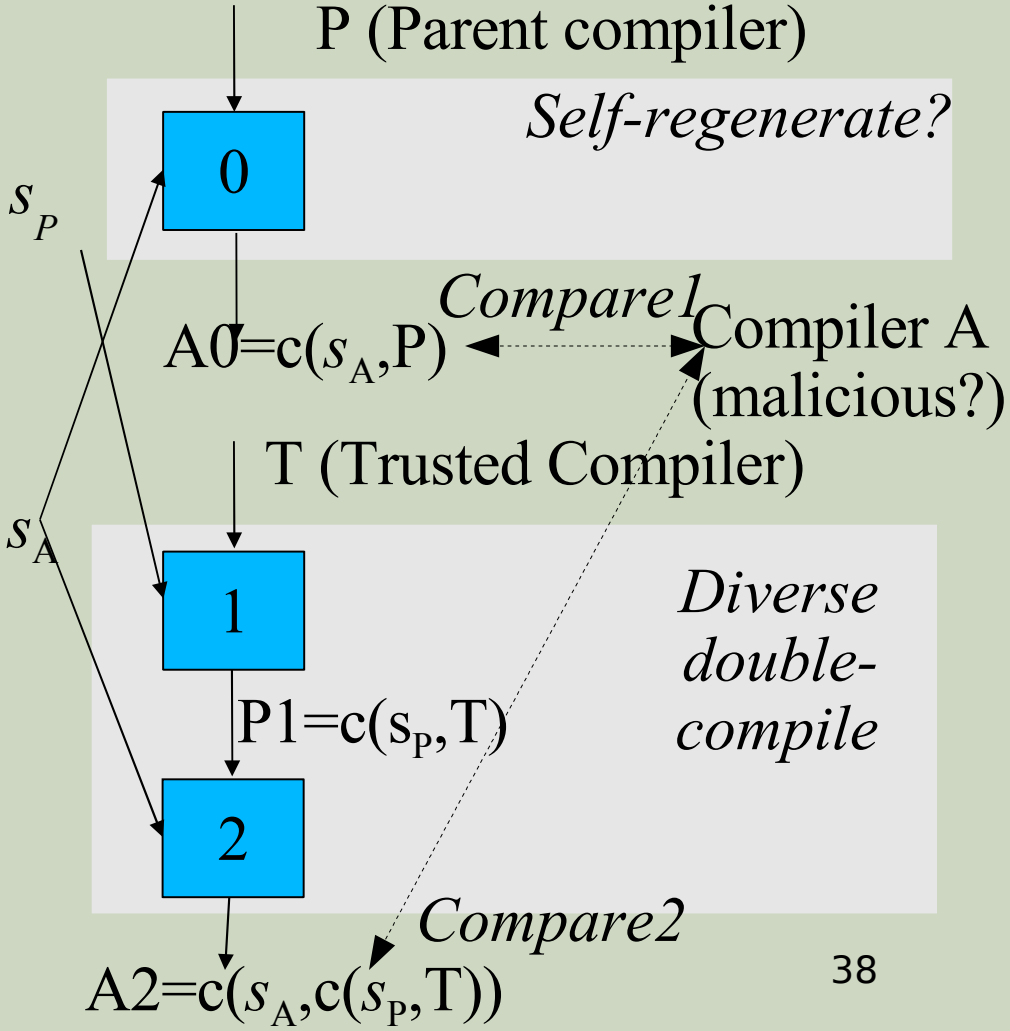
- Practical counter for trusting trust attack
- Can expand to TCB, whole OS, & prob. hardware
- Governments could require info for evals
  - Receive all source code, inc. build instructions:
    - Of compilers: so can check them this way
    - Of non-compilers: check by recompiling
  - Could establish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

In the News...

- Published *Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*, December 2005, “Countering Trusting Trust through Diverse Double-Compiling”
- Required reading: Northern Kentucky University's CSC 593: Secure Software Engineering Seminar, Spring 06
- 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 ("Backdoor"), Open Web Application Security Project (OWASP)
- Bruce Schneier's weblog and Crypto-Gram

Recent Work: Relaxing Constraint: Compiler Need not be Self-compiled

- **Instead of self-compiling, can use parent compiler P**
  - **P may be just a different version of A**
- **Source code s is now s_A union s_P**
  - **Needs examining**
  - **If similar, diff**
- **Can be used to "break" a loop**



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