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

David A. Wheeler

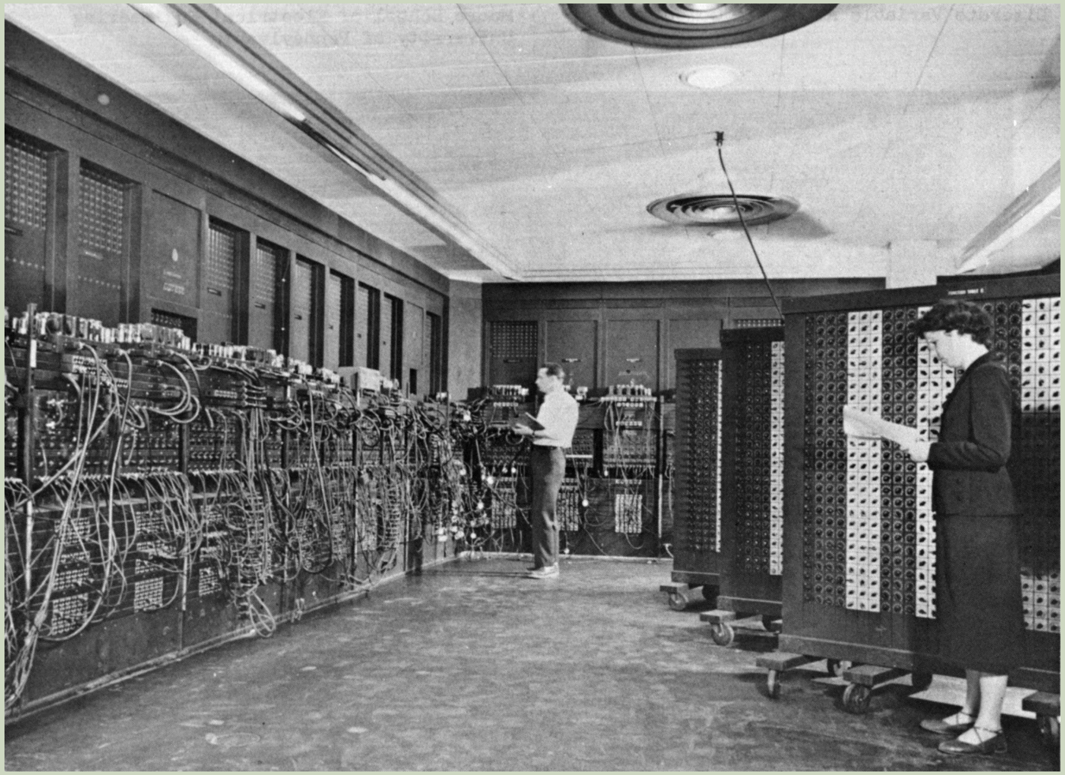
May 7, 2013

http://www.dwheeler.com/trusting-trust

Outline

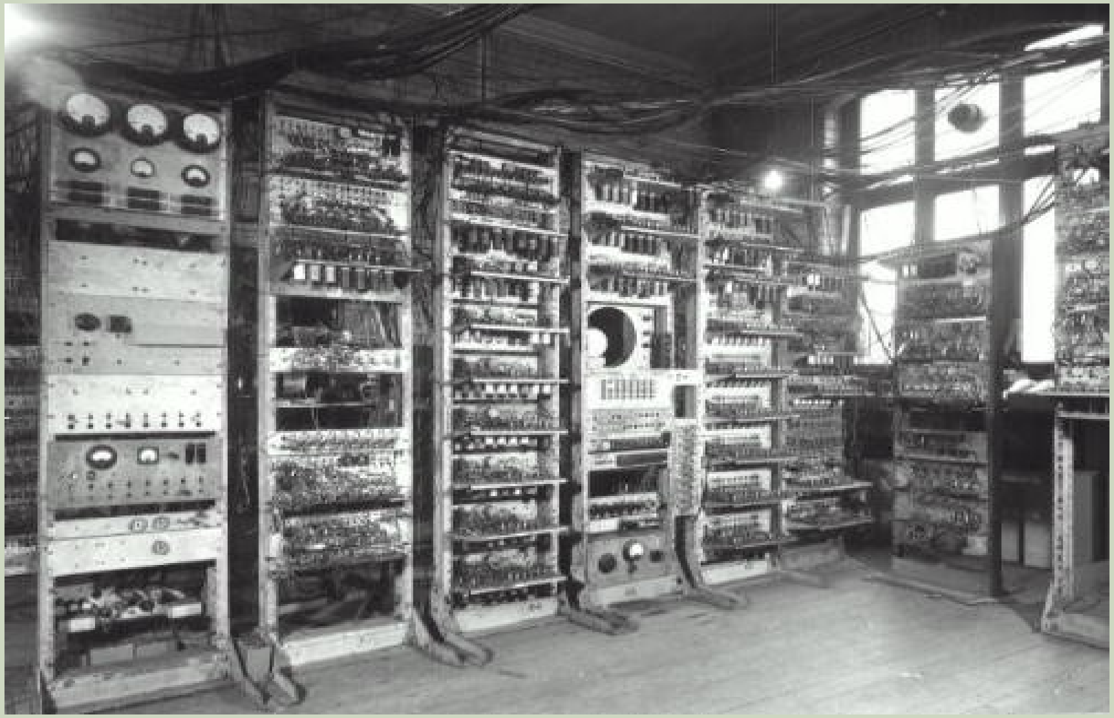
Presentation generally follows dissertation outline (+ additional background)

ENIAC (Operational July 1946)



- First general-purpose electronic computer
- Programmed by patch cables and switches
  - Could take <u>days</u> to reprogram

Manchester Small-Scale Experimental Machine [“Baby”] (June 1948)

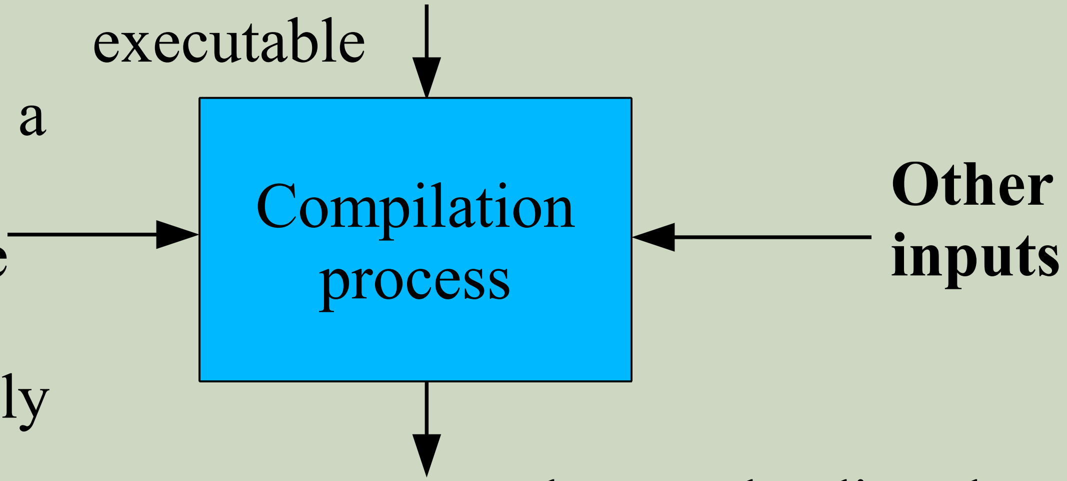


- First stored-program computer (memory held the program as well as the data it was working on)
  - Could (re)load programs quickly
  - Computer can generate computer programs

Executables, Source, Compilers

Compiler: An executable that translates source code into an executable

Source code aka **source**: Data that is a representation of a program that can be translated into an executable. Typically human-readable



Executable: Data that can be directly executed by a computing environment (real or virtual). Aka "binary"

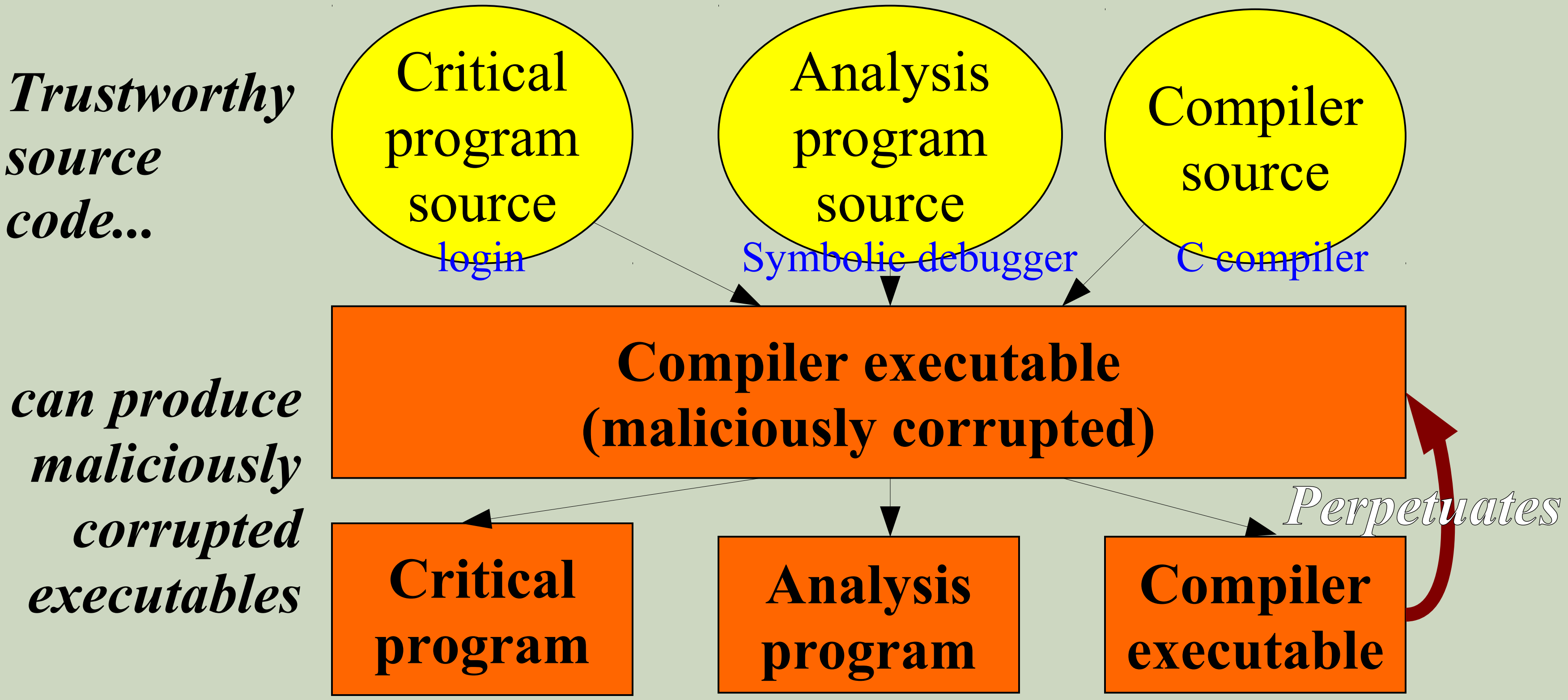
Compilers ease development & modification of software... but with risks that were only revealed later

Executables can be corrupted

- 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

We can find maliciously corrupted executables by compiling again & seeing if the results match, right?
No, not always.

Trusting Trust attack



1974: Karger & Schell first described (obliquely)

1984: Ken Thompson. Demo'd.

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”

Other attacks exist, but tend to have detection techniques/countermeasures

Attack	Detection/Countermeasure
…	…
Find & exploit unintentional weaknesses in existing program	Search for weaknesses in program, modify design to reduce impact, etc.
Insert weakness/attack in program source code	Review source code
Modify/replace executable without trusting trust attack	Regenerate executable and compare
Trusting trust attack	*No effective measure before now*

Problem Importance

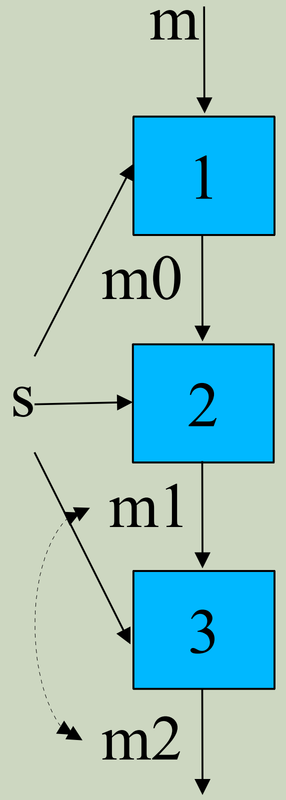
- “Trusting trust” has been treated as if it were a fundamental computer security “axiom”
  - Attack that “can’t” be countered
  - Decades of no adequate solution
  - “Computer security is hopeless”
- Attackers have incentive to use it at some point if uncounterable
  - Huge benefits: possibly control 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

Some related background

- "Simple" solutions ineffective
  - Manual analysis impractical (size, change)
  - Interpreted languages—merely moves attack
- Draper 1984: "Paraphrase" compiler could filter
  - No way to confirm if countered
- McDermott 1984: Paraphrase compiler could be reduced-function, could add 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 test)
- If m1≠m2, an assumption is wrong (fails test)
  - Often the wrong assumption is "s is correct", making this a helpful test
- A corrupted compiler can pass this test
  - Passing test doesn't prove correctness



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 four compilers (a small C compiler, a small Lisp compiler, a small maliciously corrupted Lisp compiler, and a large industrial-strength C compiler, GCC), and
3. a description of approaches for applying DDC in various real-world scenarios

3. Description of threat

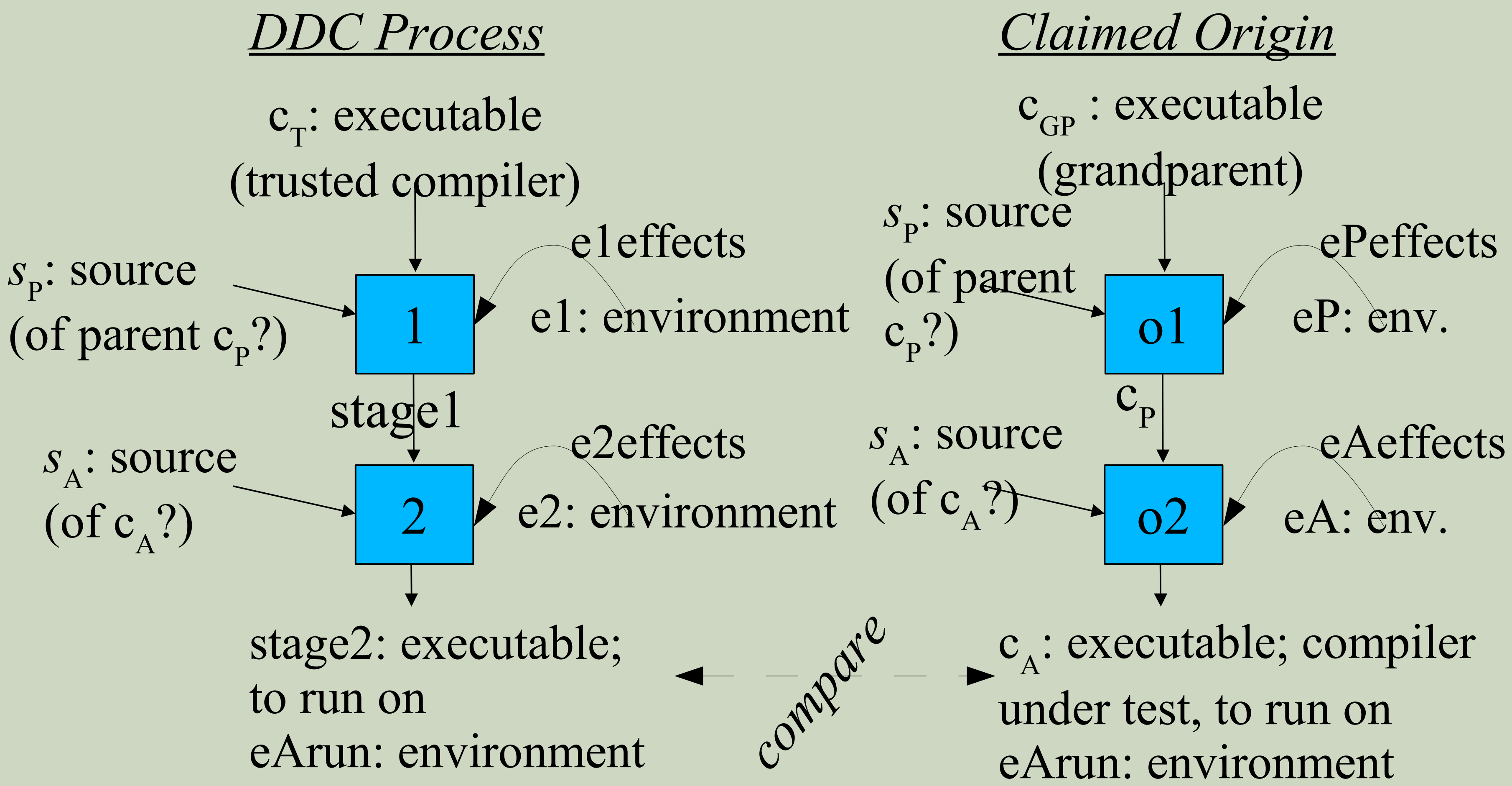
- Attacker motivation
  - Successful “trusting trust” attack enables control of all systems compiled by that compiler
  - Until this work, essentially undetectable
- Attack depends on:
  - Trigger: Condition determined by an attacker in which a malicious event is to occur (e.g., when malicious code is 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
    - 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
    - Testing for bit-for-bit equality is easy
  - 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)



DDC Process
c_T: executable (trusted compiler)
s_P: source (of parent c_P?)
1
e1effects
e1: environment
stage1
s_A: source (of c_A?)
2
e2effects
e2: environment
stage2: executable; to run on eArun: environment
compare
Claimed Origin
c_{GP} : executable (grandparent)
s_P: source (of parent c_P?)
o1
ePeffects
eP: env.
c_P
s_A: source (of c_A?)
o2
eAeffects
eA: env.
c_A: executable; compiler under test, to run on eArun: environment

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)
  - Real compilers typically deterministic
    - Non-deterministic compilers hard to test & can't use compiler bootstrap test

DDC does *not* assume that different compilers produce identical executables

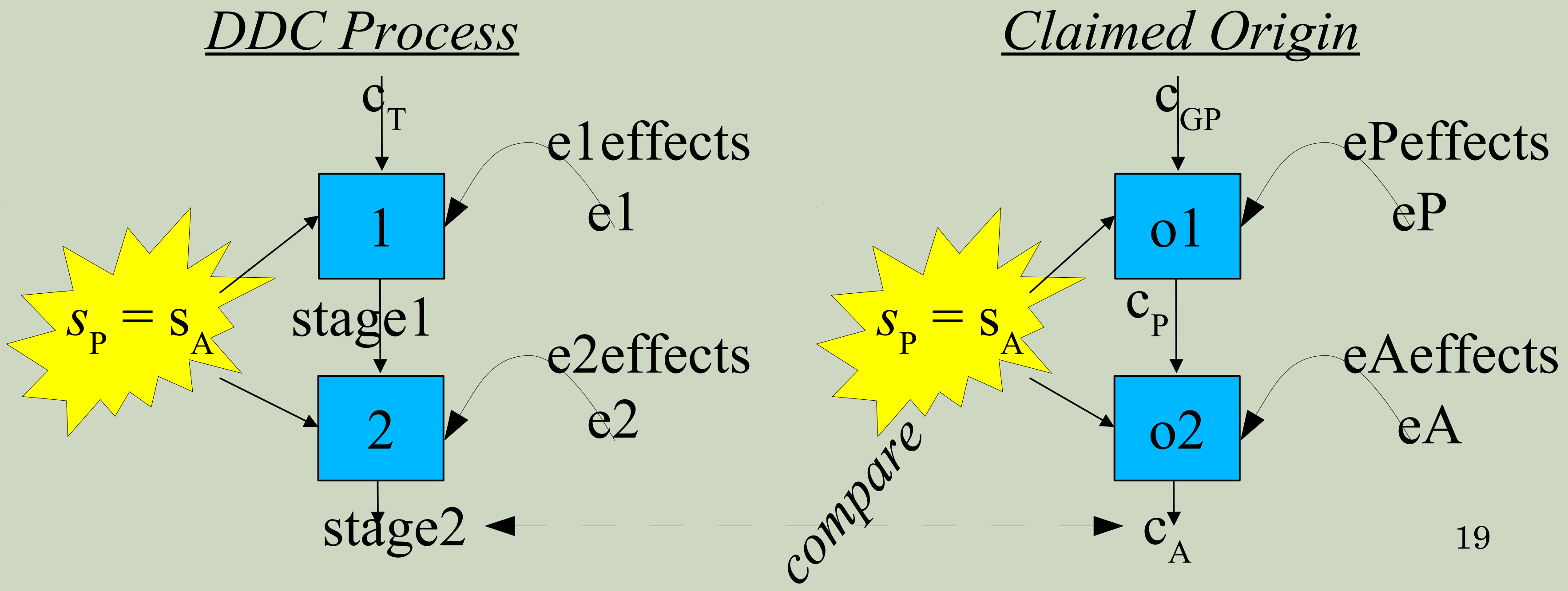
- Different compilers typically produce different executables
- But given this C source:

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

- And two different properly-working C compilers:
  - Resulting executables will usually differ
  - *Running* those executables should produce “4” (modulo text encoding, & presuming certain other assumptions)

Special case: Self-parenting compiler

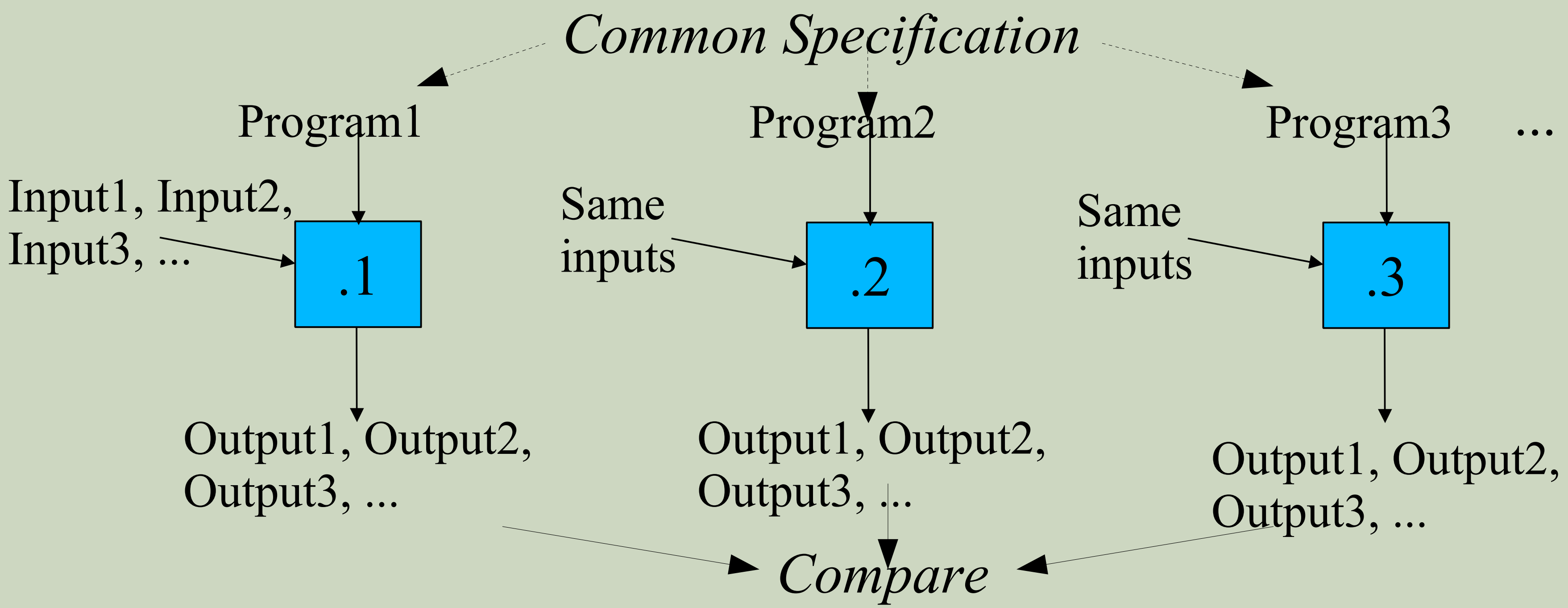
- If source $s_P = s_A$, termed "self-parenting"
- My 2005 ACSAC paper explained DDC for this case
- Dissertation generalizes 2005 paper
  - DDC no longer requires it, it's simply a special case



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 confidence that source & 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**

N-version programming is fundamentally different from DDC


Common Specification
Program1
Program2
Program3
…
Input1, Input2, Input3, …
Same inputs
Same inputs
.1
.2
.3
Output1, Output2, Output3, …
Output1, Output2, Output3, …
Output1, Output2, Output3, …
Compare

- **N-version: Multiple programs implement same specification, receive same inputs—outputs equal? Independence of errors doesn't hold!**
- **DDC: Different purpose—detect when <u>*not*</u> common specification**
  - **Trusting trust attack *not* a tiny accidental difference (tricky!)**
  - **Trusted compiler selected as unlikely to include same attack**
- **DDC: Single input (pair of source code)/output, <u>not</u> all possible I/O**

Can we create a *strong* justification that DDC counters trusting trust?

- 2005 ACSAC paper has informal justification
  - Can we do better? Not the first with this concern...

"The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so... when there are disputes... we can simply say: Let us calculate... to see who is right."
– Gottfried Leibniz, *The Art of Discovery* (1685)

Notations & deductive systems now exist that are sufficient for our purpose

Gold Standard

5. Formal proof

- Dissertation provides a formal proof for DDC
  - More rigor than the informal justification in the 2005 ACSAC paper (though based on it)
  - Uses classical First-Order Logic (FOL) with equality
    - Very widely accepted/used
    - *Not* new logic system – *models* circumstance
- Tools: Prover9 & Ivy
  - Including why you should believe the proofs
- Three proofs
- Correct goals & assumptions?

Tools: Prover9 & Ivy

- Prover9 accepts assumptions and goals in first-order logic (FOL)
  - If it can prove goal, outputs proof (by contradiction)
  - 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
- Prover9, Ivy, & ACL2 are open source software
  - Open to review by all
- Proofs also hand-verified (myself & co-workers)
- Excellent evidence that when the assumptions are true, the conclusions ***must*** follow

Three proofs

- Proof 1: “If DDC produces the same executable as the compiler-under-test c_A, then source code s_A corresponds to the executable c_A” (5 assumptions, 19 steps)

```
(stage2 = cA) -> exactly_correspond(cA, sA, lsA, eArun).
```

- Proof 2: “Under benign conditions and cP_corresponds _to_sP, the DDC result stage2 and the compiler-under-test c_A will be the same” (9 assumptions, 30 steps)

```
stage2 = cA.
```

- Proof 3: “When there’s a benign environment & a grandparent compiler, proof 2 assumption cP_corresponds_to_sP is true” (3 assumptions, 10 steps)

```
exactly_correspond(cP, sP, lsP, eA).
```

- Discovered need for 3 proofs as proofs were developed

Correct goals & assumptions?

- This can't be shown formally
- Reasons to believe correct goals & assumptions:
  - Assumptions proven consistent (mace4 can create model that satisfies them)
  - Based on informal justification in peer-reviewed ACSAC paper, which no one has refuted
  - Author, co-workers, committee have reviewed
  - All demonstration results explainable by proofs
  - Formalization process forced clarification that there were multiple claims to prove; suggests insight from proof
  - Proofs clearly fit together
    - #3: If benign + a grandparent, then cP_corresponds_to_sP
    - #2: If benign + cP_corresponds_to_sP, then stage2 = cA
    - #1: If stage2 = cA, then cA and sA correspond

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

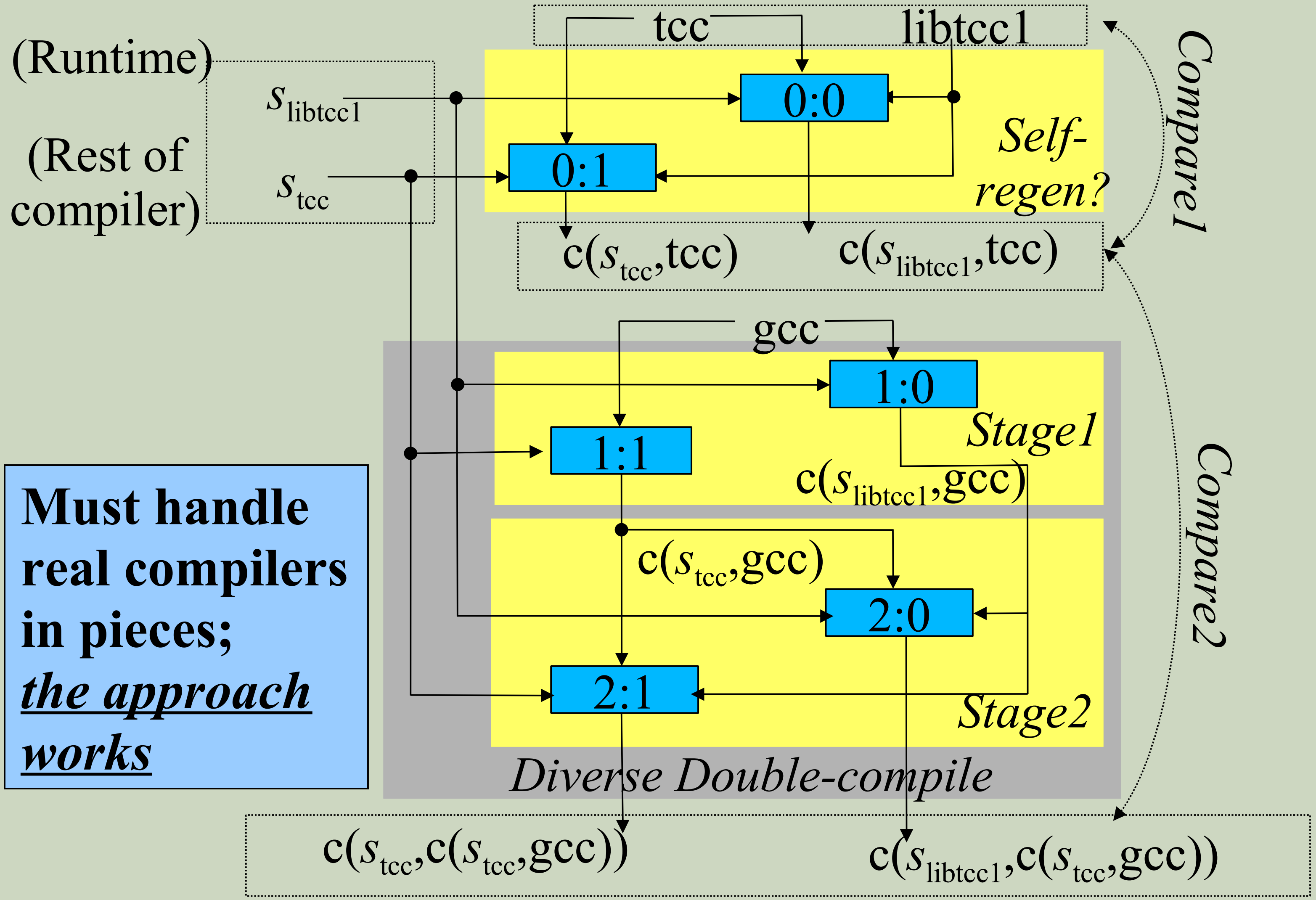
- tcc (TinyCC): C compiler (ACSAC paper)
- 2 Goerigk Lisp compilers
  - One uncorrupted
  - One maliciously corrupted
- GCC (scales up)

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*

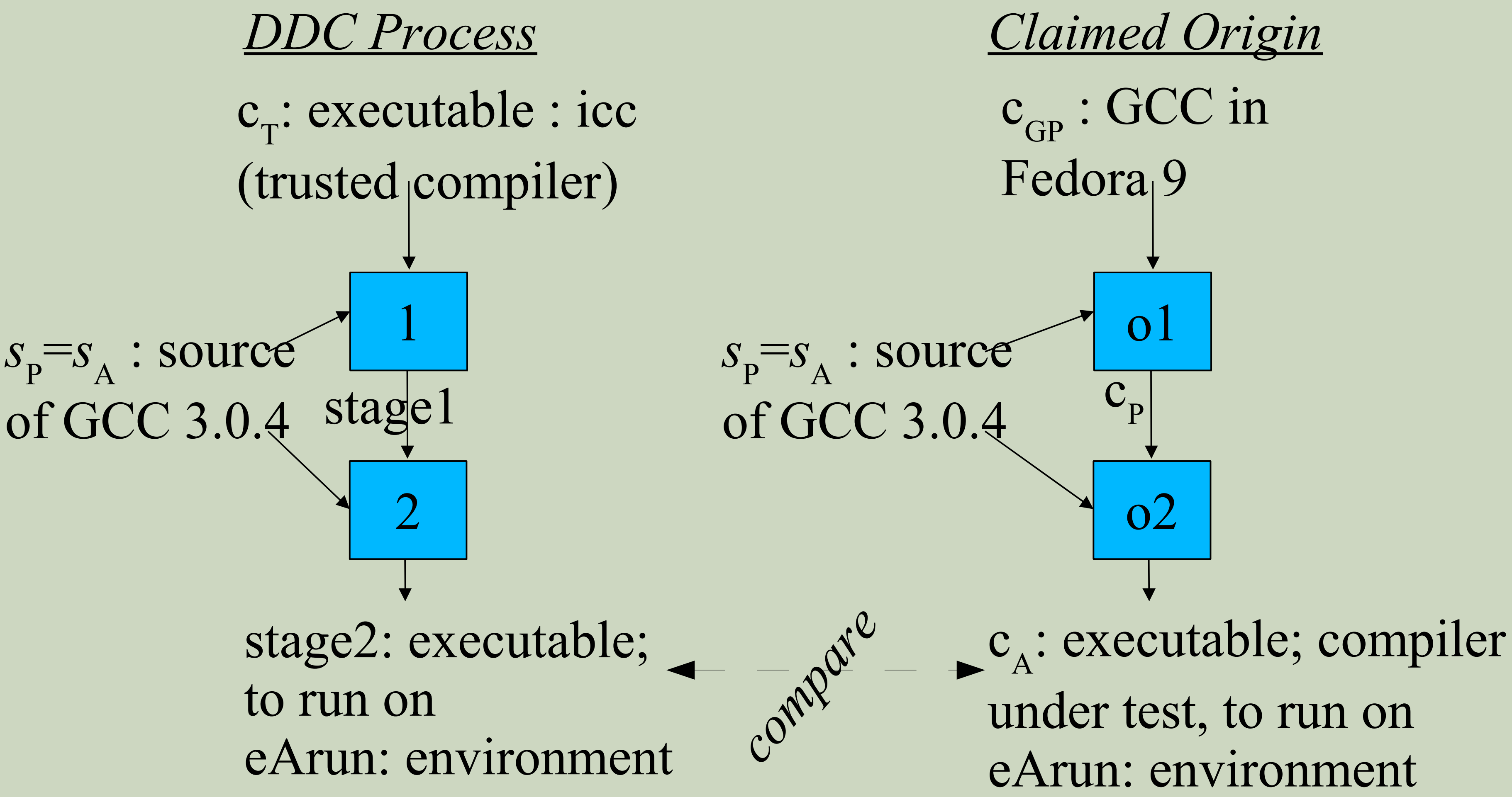
Goerigk Lisp compilers

- 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

DDC applied to GCC (simplified)


DDC Process
c_T: executable : icc (trusted compiler)
s_P=s_A : source of GCC 3.0.4
1
stage1
2
stage2: executable; to run on eArun: environment
compare
Claimed Origin
c_{GP} : GCC in Fedora 9
s_P=s_A : source of GCC 3.0.4
o1
c_P
o2
c_A: executable; compiler under test, to run on eArun: environment

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)
    - This took time to trace back & determine cause
- Once corrected, DDC produced bit-for-bit equal results as expected

8. Practical challenges

- Limitations
  - Depends on confidence DDC process elements (trusted compiler, environment, etc.) 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 only 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

How can an attacker counter DDC?

Must falsify a DDC assumption, for example:

- Swap DDC result with c_A during DDC process (!)
  - Defender can protect DDC environment
- Make compiler-under-test ≠ compiler used
  - If environment may provide inaccurate compiler under test, defender can extract without using environment
  - If environment may run different compiler, defender can redefine “compiler” to include environment & apply DDC
- Subvert trusted compiler/trusted environment(s)
  - Challenge: Don’t usually know what they’ll be
  - Defender can use DDC multiple times
    - Attacker must subvert them *all,* while defender only needs to protect at least one—unusual for defender

9. Conclusions and ramifications

- DDC can show that source and executable correspond
  - Executable may have errors or be malicious, but these can be found by examining source (easier)
- DDC primarily useful to those who have access to the source code (advantage for open source software)
- Policy implications – for compilers of critical software:
  - Require information to do DDC?
  - Require use of (unpatented) language standards?
- Potential future work: Recompiling whole OS

The trusting trust attack can be detected and effectively countered by DDC

Summary of formal proofs

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

man(X) -> mortal(X).	% "All men are mortal."
man(socrates).	% "Socrates is a man."

% 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, 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 9 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_portable_and_deterministic
  - define_portable_and_deterministic

definition_cA and cP_corresponds_to_sP

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

exactly_correspond(cP, sP, lsP, 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_portable_and_deterministic

portable_and_deterministic(sP, lsP, retarget(sA, eArun)).

sP is deterministic: Same input produces same output. Avoids all non-deterministic capabilities of language lsP, or uses them only in ways that will not affect the output of the program.

sP is portable; it only uses the portable constructs of lsP (the ones that are true on different environments)

The above only need to be true when compiling the retargeted sA

define_portable_and_deterministic

```
( portable_and_deterministic(Source, Language, Input) &
  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). This only needs to be true for the specific input Input.

Example of define_portable_and_deterministic

- Given this portable C source:

```
#include <stdio.h>
main() {
   int a;
   scanf("%d", &a);
   a = a + 1;
   printf("%d\n", a);
}
```

- And different, correct, deterministic C compilers:
  - Resulting executables will usually differ
  - *Running* executables produces the same results (modulo text encoding) *if* the input does not cause portability issues (e.g., portable range of int)

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.

Backup

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

Timestamps

- Some formats (e.g., “.a”) embed timestamps
  - Makes compiling non-deterministic (recompiling produces different results)
- Simple solution: use formats that don’t
  - ELF “.o” format does *not* include timestamps
- If *had* to handle embedded timestamps:
  - Could use “=” operator modified to ignore them
    - But must then show that these timestamps do not affect execution (more difficult)
  - Could rig functions for getting current time so that they produced repeatable results

What have I learned?

- Formal methods & tools
  - Much greater understanding of specification languages and notations
  - Consider the “weakest” useful notation, e.g., first-order vs. higher-order (more automatable)
  - Prefer a prover with verifier
  - Start with *simple* world model, prove, then build up
- “Foundations of mathematics” issues of 1900s
  - “Trusting trust” attack builds on self-reference, as does Liar’s paradox, Russell’s paradox, etc.
  - The logic systems that were built in the process of addressing them (FOL, etc.) also helped here!
- Debugging big compilers is painful

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”
- Immediately became 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

Ken Thompson's comments

- *> good to hear from you. i have, of coarse, read your DDC [ACSAC] paper. from a theoretical point of view, i think it is very nice...*
- *> but from a practical point, i think the cure is at least as bad as the disease.†*
- Makes 3 points about DDC per ACSAC paper
  - He had not read dissertation at the time, since it had just come out
  - Reasonable comments, but I believe there are reasonable responses too

†Capitalization & punctuation as per Thompson

Ken Thompson’s comments (continued)

> 1. there are very few people in a position to widely install and distribute a 'trust' bug. i think the threat is close to zero. certainly the threat is much smaller than the distribution of binary-only code that is so prevalent on the net today. how do you know that one of the norton updates, microsoft patches, flash or adobe downloads doesnt contain malicious code? routinely you get megabytes of code off the net and then type your system password to install it. if i were a bad guy, i would take the path of least resistance and buy off someone at adobe.

- **Few people that are *supposed* to be in such a position, but subversion of development environments can yield many more**
- **“Binary-only” code historically widespread, but OSS gaining**
- **Yes, there *are* other attacks that are *currently* easier, but there are known detection techniques & countermeasures for the others**
  - **If suppliers & users decide countering them is important, such techniques can be applied...**
  - **But no point if there’s an undetectable/uncounterable attack**

Ken Thompson’s comments (continued)

> 2. there are a lot more targets than just the C compiler and there are a lot more disease vectors than the C compiler. there is SH, C++, CPP, LD, LIBC, AS, AR, LINUX, JAVA, PYTHON, etc, etc, etc. each of these would have to be generated from scratch with a separate trusted vector. each one would be more difficult than your TCC example, while all of them would be close to impossible. i dont know how to even find all the potential targets.

- **Dissertation uses GCC C compiler, which is more complex**
- **True that it’s not trivial, but now that DDC exists:**
  - **Developers can include DDC as one of their tests**
  - **Developers can write software to simplify DDC (e.g., standards)**
- **Can grow to apply DDC to entire Linux distribution**
  - **That would be adequate for many**
- **If no executable stored (e.g., many interpreted languages like SH), there’s no issue; there is no hidden executable for the attack**
  - **In some cases (e.g., Python), erasing caches eliminates attack**

Ken Thompson's comments (continued)

> 3. operationally, it will be hard to keep the trusted vectors all up to date. a lot of the targets are moving targets that are distributed from one source. getting and maintaining a trusted version will be hard. keeping everything working will be super-human.

- If "trusted vectors"="trusted compilers", not serious problem
  - Only need to compile one/few programs; little to keep up
  - Performance is irrelevant
  - Developers of software-under-test may limit the code constructs (e.g., standards, easily-implemented) to ease DDC application
- If "trusted vectors"="tested compilers", not serious problem
  - DDC is *detection* technique, so it can be used after-the-fact
  - Attackers won't know when DDC will be used & will know that they can be detected, greatly reducing attack incentive

> again, i really enjoyed your paper. thanks for thinking of me.

- I thank Thompson for (1) A clear explanation & demo of the attack and (2) creating Unix—both are *key* contributions to the field

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

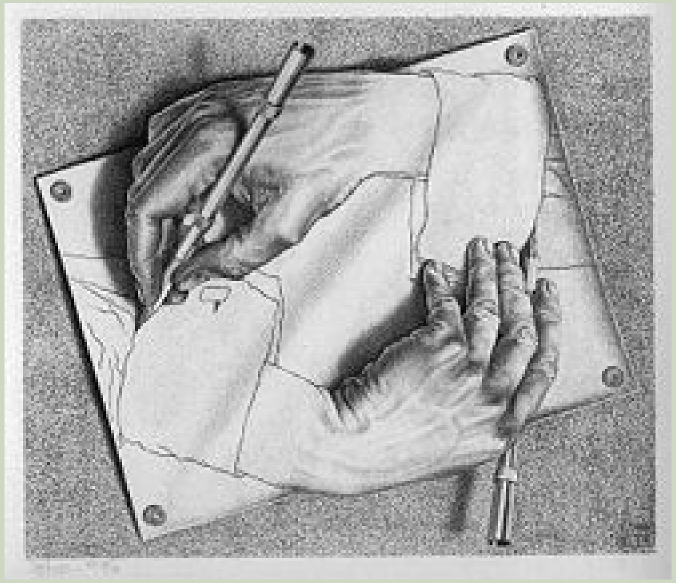
Broader implications

- 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

Early use of proof tools

- First proofs by hand
  - *Painful* to do *without* tracking environments
  - Easily argued against: “Error somewhere”
- PVS: Nice tool!
  - Supports higher-order logic (HOL)
    - Compilers are functions... that produce other compilers/functions
  - Supports typing (counters some errors)
  - Required much manual “steering” to get proofs
    - Obtained some proofs, but still painful
  - No separate verification tool
  - Over time, found did not *need* HOL or types
    - Simpler logic → more-automated tools

Self-reference & feedback are powerful, but can lead to problems & paradoxes

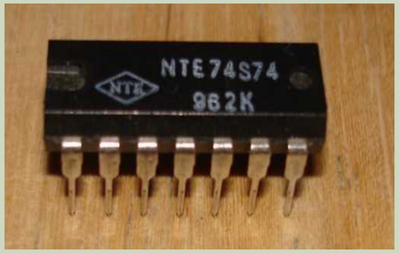


Russell's paradox

"This sentence is false"

Hardware

- BIOS/microcode is software
  - 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 software/hardware, that *is* a trusting trust attack
  - Appears that we *can* apply DDC, but there are some issues...



Applying DDC to hardware to counter trusting trust attack

- Requires 2nd implementation as trusted compiler
  - Alternative hardware compiler, simulated chip & simulator
- Requires "equality" test
  - Tool: Perhaps scanning electron microscope, scanning transmission electron microscope (STEM), use focused ion beam, or a tool that performs optical phase array shifting
  - *Maybe* use superposition—detect phase changes (diffraction)
  - Issue: Real chips *have* defects – false positive issues
- Requires knowing correct result (legal problems)
  - Often cell libraries provided to engineer are *not* the same as what is used in the chip... "real" libraries proprietary
  - "IP cores" hide information about chip subcomponents
  - Quantum effect error corrections for very high densities considered proprietary by correctors
- Only shows the chip-under-test is good

For more information

- For more information, see:
  - http://www.dwheeler.com/trusting-trust

Credits: All photographic images are from Wikipedia, except for the picture of "Baby". That picture is from the Manchester SSEM photo gallery: http://www.cs.man.ac.uk/CCS/ssem/ssemgall.htm

The images are used on the basis of U.S. "fair use":

1. Purpose/character: The presentation purpose is nonprofit educational use
2. Nature of the work: Images of Baby and ENIAC used as facts
3. Amount in containing work: The images are a small proportion of the work
4. Effect on market: No expected effect on their market value