Verification

of

Programs and Systems

Georg Weissenbacher
https://www.forsyte.at

Formal Methods
in Systems Engineering

Bugs in the news ...

Toyota Prius

(New York Times, Feb. 12, 2014)

Toyota Motor is recalling all of the 1.9 million newest-generation Prius vehicles it has sold worldwide because of a programming error ...

Bugs in the news ...

Toyota Prius

(New York Times, Feb. 12, 2014)
Toyota Motor is recalling all of the 1.9 million newest-generation Prius vehicles it has sold worldwide because of a programming error ...

Heathrow Airport

(The Guardian, December 2014)
An unprecedented systems failure was responsible for the air traffic control chaos [...] "In this instance a transition between the two states caused a failure in the system which has not been seen before," ...

Heathrow
Making every journey better

What goes up ...

Lufthansa Airbus A321

(Spiegel, March 20, 2015)

Beinahe wäre ein Airbus A321 der Lufthansa mit 109 Passagieren auf dem Flug von Bilbao nach München abgestürzt – irregeleitete Bordcomputer hatten die Kontrolle übernommen.

What goes up ...

Lufthansa Airbus A321

(Spiegel, March 20, 2015)
Beinahe wäre ein Airbus A321 der Lufthansa mit 109 Passagieren auf dem Flug von Bilbao nach München abgestürzt – irregeleitete Bordcomputer hatten die Kontrolle übernommen.

Boeing 787 Dreamliner

(The Guardian, May 2015)
The US air safety authority has issued a warning and maintenance order over a software bug that causes a complete electric shutdown of Boeing's 787 ...

Some hardware bugs ...

Meltdown and Spectre

(New York Times, January 2018)

Called Meltdown, the first and most urgent flaw affects nearly all microprocessors by Intel. The second, Spectre, affects most other chips ...

Some hardware bugs ...

Meltdown and Spectre

(New York Times, January 2018)

Called Meltdown, the first and most urgent flaw affects nearly all microprocessors by Intel. The second, Spectre, affects most other chips ...

Rowhammer Bug

(InfoWorld, March 9, 2015)

...with certain varieties of DRAM an attacker can create privilege escalations by simply repeatedly accessing a given row of memory.

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

126 Comments

Quelle: `www.edn.com`

- Oklahoma courd ruled against Toyata in case of unintended acceleration that lead to death
- Expert witness found numerous bugs in software (including bugs that can cause unintended acceleration), founds source code of "unreasonable quality"

RISK ASSESSMENT —

“Most serious” Linux privilege-escalation bug ever is under active exploit (updated)

Lurking in the kernel for nine years, flaw gives untrusted users unfettered root access.

DAN GOODIN - 10/20/2016, 10:20 PM

DIRTY COW

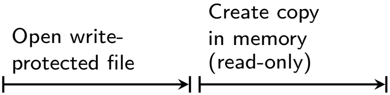
“copy on write”

(CVE-2016-5195, published October 2016)

Anatomy of “Dirty Copy-On-Write” Attack

Goal: Write to protected systems file

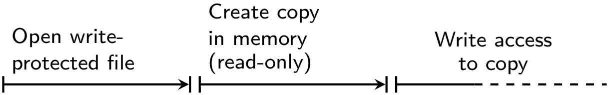
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of "Dirty Copy-On-Write" Attack

Goal: Write to protected systems file

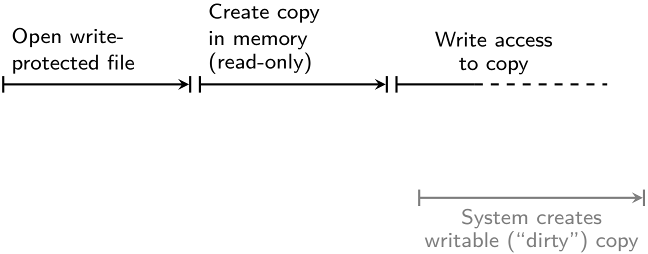
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of “Dirty Copy-On-Write” Attack

Goal: Write to protected systems file

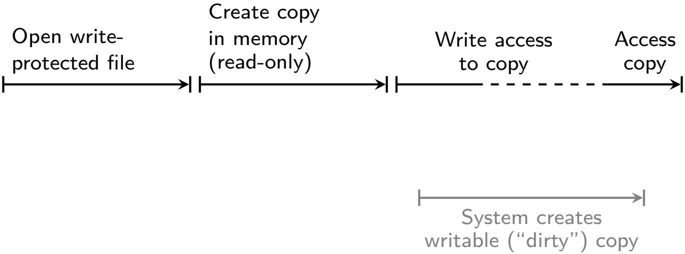
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of "Dirty Copy-On-Write" Attack

Goal: Write to protected systems file

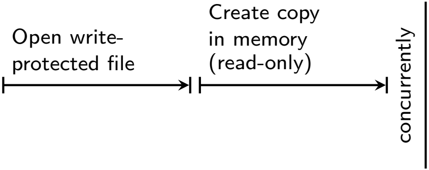
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of "Dirty Copy-On-Write" Attack

Goal: Write to protected systems file

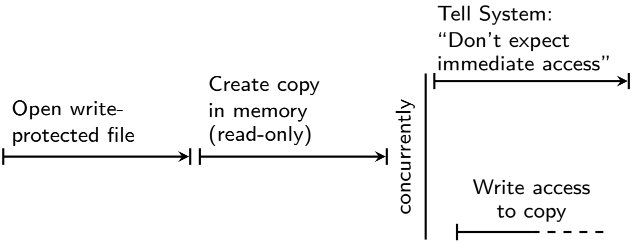
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of "Dirty Copy-On-Write" Attack

Goal: Write to protected systems file

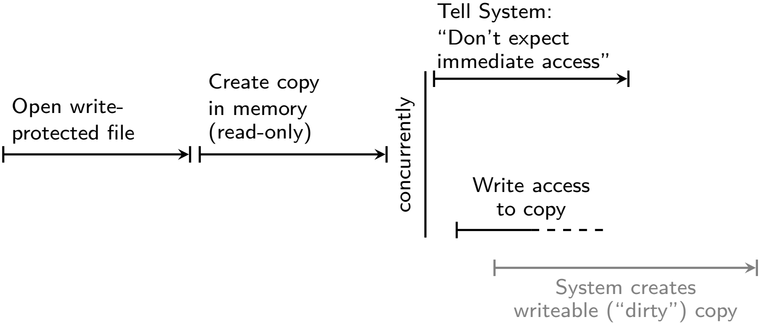
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of “Dirty Copy-On-Write” Attack

Goal: Write to protected systems file

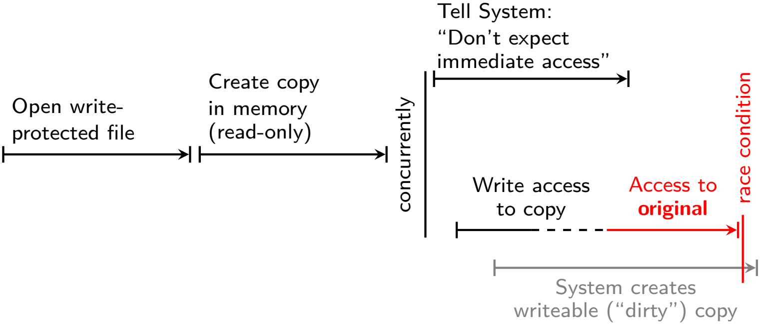
Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



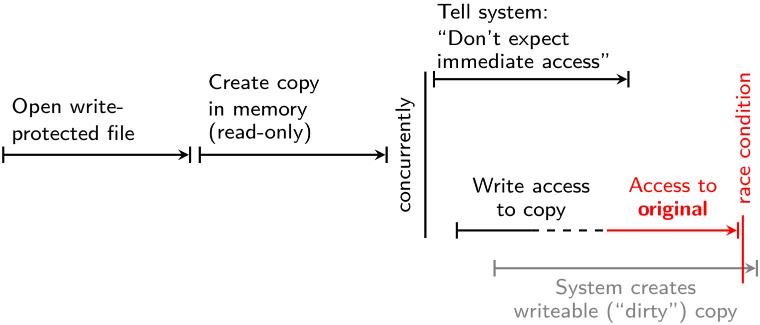
Anatomy of "Dirty Copy-On-Write" Attack

Goal: Write to protected systems file

Copy-On-Write (COW) Upon write attempt, system creates copy of protected memory area



Anatomy of "Dirty Copy-On-Write" Attack



Race Condition rarely happens

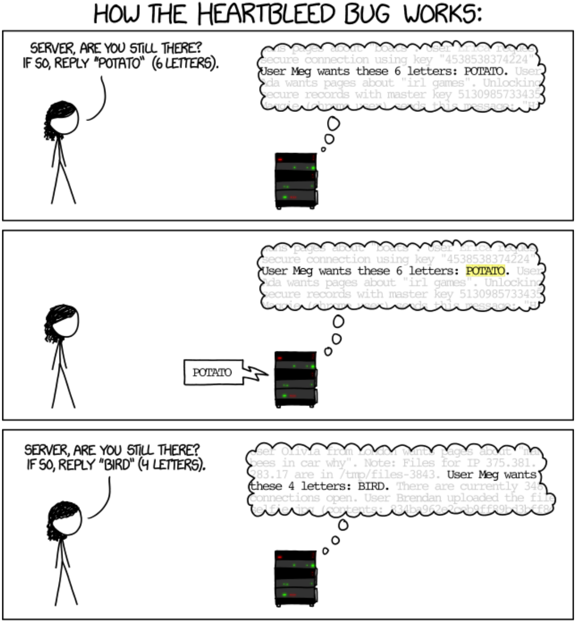
- *Testing* isn't particularly effective
- Systematic search (of schedules) is required

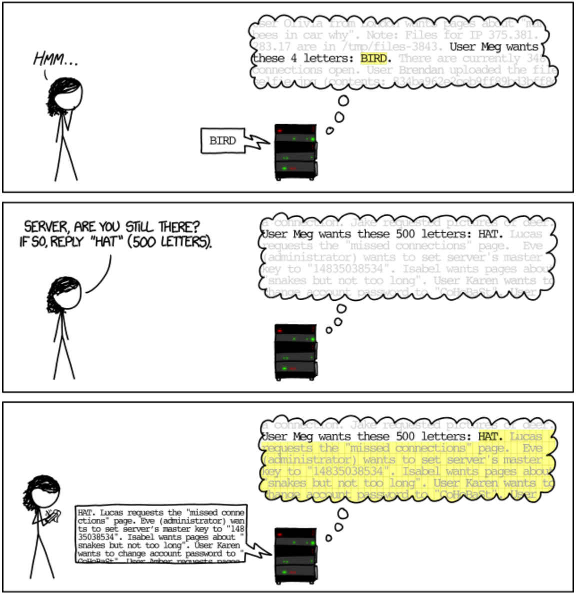
Another security bug ...

Heartbleed Bug

(CNN, April 9, 2014)

A major online security vulnerability dubbed "Heartbleed" could put your personal information at risk, including passwords, credit card information and e-mails.


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.


HMM...
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User
a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas
requests the "missed connections" page. Eve
(administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
"snakes but not too long". User Karen wants to
change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed conne
ctions" page. Eve (administrator) wan
ts to set server's master key to "148
35038534". Isabel wants pages about "
snakes but not too long". User Karen
wants to change account password to "

Can you see it?

```
typedef struct {
  char* data;
  unsigned int len;
} ssl_buffer;

typedef struct {
  ssl_buffer buffer;
} SSL;

int tls1_process_heartbeat(SSL *s)
{
  char *p=s->buffer.data,*pl;

  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16;

  hbtype = *p++;
  n2s(p, payload);
  pl = p;
```

```
if (hbtype == TLS1_HB_REQUEST)
{
  unsigned char *buffer, *bp;
  int r;

  buffer = malloc(1 + 2 +
                  payload +
                  padding);
  bp = buffer;
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);
  bp += payload;
  RAND_pseudo_bytes(bp, padding);

  r = ssl3_write_bytes
      (s,TLS1_RT_HEARTBEAT,
       buffer,
       3 + payload + padding);
  free(buffer);
  if (r < 0)
    return r;
}
```

Let's use a tool to find the bug! (Try this at home)

- C Bounded Model Checker (CBMC): `https://www.cprover.org/cbmc`
- Install command line tool
  - On Ubuntu: `sudo apt install cbmc` (version > 5.10):

Let's use a tool to find the bug! (Try this at home)

- C Bounded Model Checker (CBMC):
  `https://www.cprover.org/cbmc`
- Install command line tool
  - On Ubuntu: `sudo apt install cbmc` (version > 5.10):
- Run: `cbmc --pointer-check heartbleed.c`

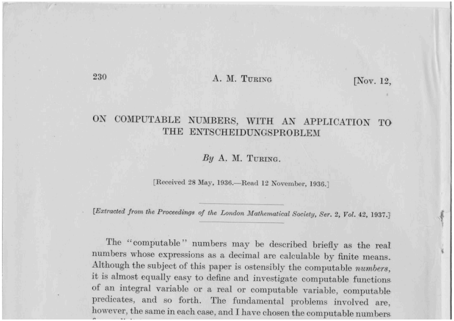
Let's use a tool to find the bug! (Try this at home)

- C Bounded Model Checker (CBMC): `https://www.cprover.org/cbmc`
- Install command line tool
  - On Ubuntu: `sudo apt install cbmc` (version > 5.10):
- Run: `cbmc --pointer-check heartbleed.c`
- Output:

```
** Results:
<builtin-library-malloc> function malloc
[malloc.assertion.1] line 25 max allocation size exceeded: SUCCESS

src/heartbleed.c function tls1_process_heartbeat
[tls1_process_heartbeat.precondition_instance.1] line 54 memcpy
src/dst overlap: SUCCESS
[tls1_process_heartbeat.precondition_instance.2] line 54 memcpy
source region readable: FAILURE
[tls1_process_heartbeat.precondition_instance.3] line 54 memcpy
destination region writeable: SUCCESS
```

Can we always find bugs automatically?



230 A. M. TURING [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

[*Extracted from the Proceedings of the London Mathematical Society, Ser. 2, Vol.* 42, 1937.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers

Alan Turing (1912–1954)

Turing's Halting Problem (1936)

Given a description of a program, decide whether the program finishes running or continues to run forever.

(undecidable)

Turing's Halting Problem

Proof ingredients:

- Program can be encoded as string
- Program operations can be simulated by Turing machine
- Diagonalization

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i,i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i, x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i, i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Assume e is program implementing g ($\perp$ amounts to infinite loop)

- So e with input i does *not* terminate if i terminates on input i

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i,i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Assume e is program implementing g ($\perp$ amounts to infinite loop)

- So e with input i does *not* terminate if i terminates on input i

We perform a case split:

- $g(e) = h(e,e) = 0$.

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i,i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Assume e is program implementing g ($\perp$ amounts to infinite loop)

- So e with input i does *not* terminate if i terminates on input i

We perform a case split:

- $g(e) = h(e,e) = 0$. But e halts on input e, thus $h(e,e) = 1$

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i, x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i, i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Assume e is program implementing g ($\perp$ amounts to infinite loop)

- So e with input i does *not* terminate if i terminates on input i

We perform a case split:

- $g(e) = h(e, e) = 0$. But e halts on input e, thus $h(e, e) = 1$
- $g(e) = \perp$ and $h(e, e) \neq 0$.

Turing's Halting Problem (Proof Sketch)

Assume h is a computable function

$$h(i,x) = \begin{cases} 1 & \text{if program } i \text{ halts on input } x \\ 0 & \text{otherwise} \end{cases}$$

$$g(i) = \begin{cases} 0 & \text{if } h(i,i) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Assume e is program implementing g ($\perp$ amounts to infinite loop)

- So e with input i does *not* terminate if i terminates on input i

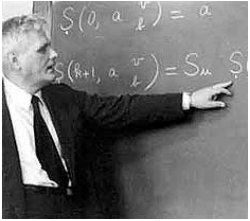
We perform a case split:

- $g(e) = h(e,e) = 0$. But e halts on input e, thus $h(e,e) = 1$
- $g(e) = \perp$ and $h(e,e) \neq 0$. But e doesn't halt, so $h(e,e) = 0$

Can we always find bugs automatically?

Kurt Gödel, 1931:
Über formal entscheidbare Sätze der Principia Mathematica und verwandter Systeme

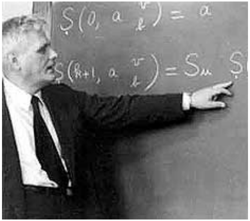
Alonzo Church, 1936:
An Unsolvable Problem of Elementary Number Theory



Can we always find bugs automatically?

Kurt Gödel, 1931:
Über formal entscheidbare Sätze der Principia Mathematica und verwandter Systeme

Alonzo Church, 1936:
An Unsolvable Problem of Elementary Number Theory



Mission impossible?

What can be done?

“Software pioneers” in WW2

Alan Turing

Herman Goldstine
J. Robert Oppenheimer
John von Neumann

ECP

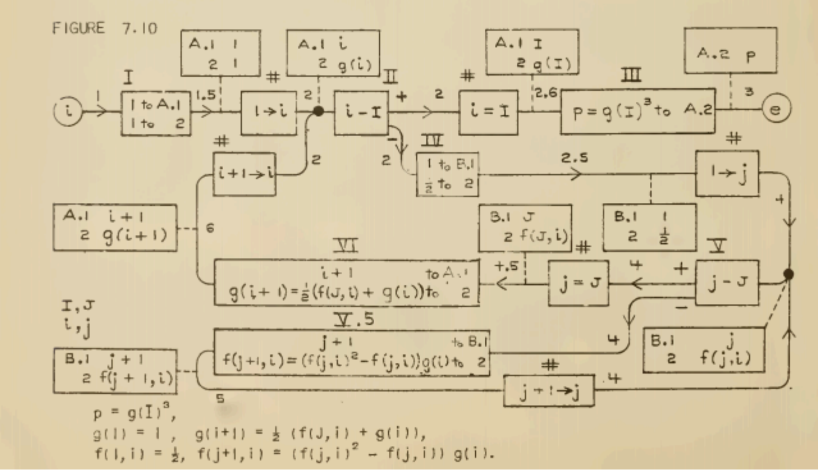
PLANNING AND CODING OF PROBLEMS FOR AN ELECTRONIC COMPUTING INSTRUMENT

BY

Herman H. Goldstine John von Neumann

Report on the Mathematical and Logical aspects of an Electronic Computing Instrument

Part II, Volume I – 3



An assertion box never requires that any specific calculations be made, it indicates only that certain relations are automatically fulfilled whenever C gets to the region which it occupies.

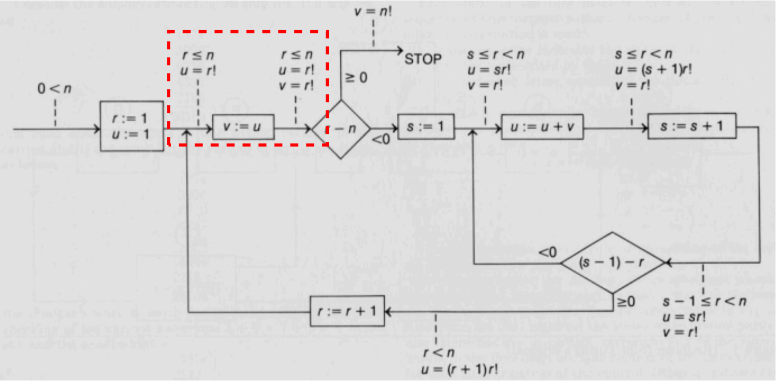
Turing didn't give up either

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.



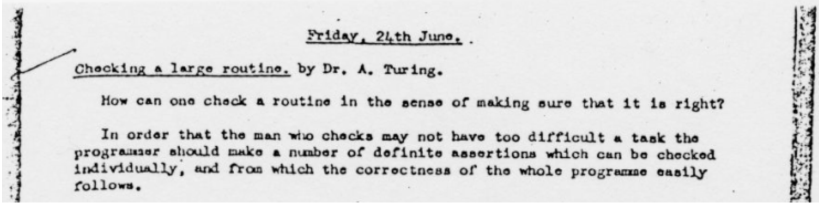
Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

the programmer is expected to state a number of observations ("assertions"), which can be checked independently of each other


Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

> the programmer is expected to state a number of observations ("assertions"), which can be checked independently of each other

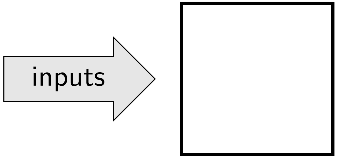
Questionnaire:

- Who of you writes programs?
- Who knows what assertions are?
- Who uses assertions?

Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

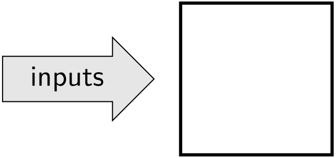
- Poke and prod the program with the right *inputs*



Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

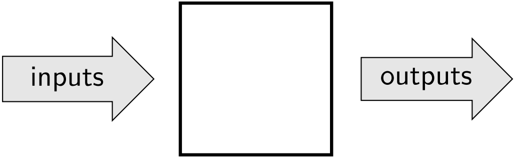
- Poke and prod the program with the right *inputs*
  - But how do we find those?



Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

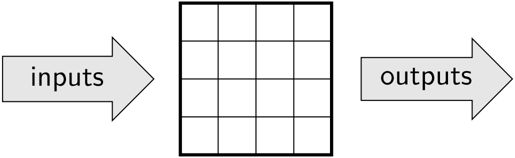
- Poke and prod the program with the right *inputs*
  - But how do we find those?
- Check whether it behaves as desired (*outputs*)



Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

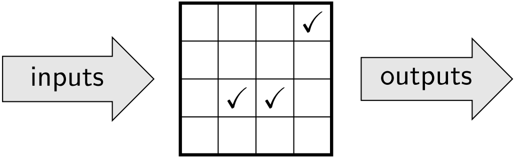
- Poke and prod the program with the right *inputs*
  - But how do we find those?
- Check whether it behaves as desired (*outputs*)
- But when are we done?



Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

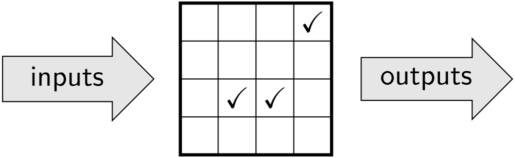
- Poke and prod the program with the right *inputs*
  - But how do we find those?
- Check whether it behaves as desired (*outputs*)
- But when are we done?



Testing: The State-of-the-Art of Verification

How do we know Assertions hold?

- Poke and prod the program with the right *inputs*
  - But how do we find those?
- Check whether it behaves as desired (*outputs*)
- But when are we done?



Time for another questionnaire:

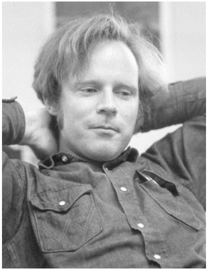
- Who of you tests *systematically*?
- Which coverage metrics do you know?

What are we even testing?

Does the program <u>behave</u> as <u>specified</u>?

- Specification
  - Required ingredients: Formalism, Assertion Language
- Program
  - Required ingredients: Language semantics

Assertions and Program Semantics



Robert W. Floyd

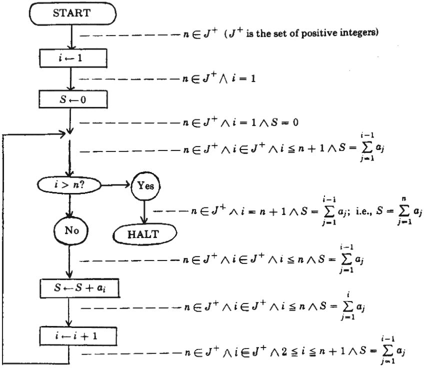
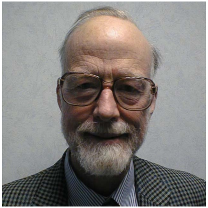


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ ($n \geqq 0$)

Then, by induction on the number of commands executed, one sees that if a program is entered by a connec(on whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at the time. By this means, we may prove certain properties of programs, ...

Floyd-Hoare Logic: Axioms for Programs



Sir C.A.R. Hoare

Hoare Triples:

$\{$Pre-Condition$\}$ Program $\{$Post-Condition$\}$

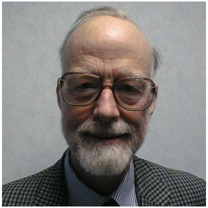
- Assignments:

$$\frac{}{\{Q[\mathtt{x}/e]\}\ \mathtt{x\ :=}\ e\ \{Q\}}$$

- Composition:

$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Floyd-Hoare Logic: Axioms for Programs



Sir C.A.R. Hoare

Hoare Triples:

$\{$Pre-Condition$\}$ Program $\{$Post-Condition$\}$

- Assignments:

$$\frac{}{\{Q[\mathtt{x}/e]\}\ \mathtt{x\ :=\ }e\ \{Q\}}$$

- Composition:

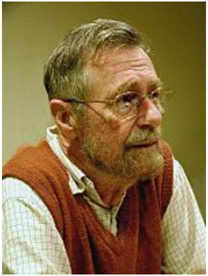
$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

Allows us to prove programs correct!

Dijkstra's Predicate Calculus

What *effect* does an instruction have on an assertion?

$$\{\mathtt{x} < 10\} \quad \mathtt{x\ :=\ x\ +\ 1} \quad \{?\}$$



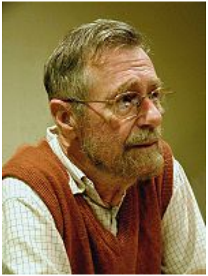
Edsger W. Dijkstra

- Strongest Postcondition:

$$\mathrm{sp}(\mathtt{x} := e, P) =$$
$$\exists \mathtt{x}_0 \,.\, \mathtt{x} = e[\mathtt{x}/\mathtt{x}_0] \wedge P[\mathtt{x}/\mathtt{x}_0]$$

where $\mathtt{x}_0$ is the "old" value of $\mathtt{x}$

Dijkstra's Predicate Calculus



Edsger W. Dijkstra

What *effect* does an instruction have on an assertion?

$$\{\mathtt{x} < 10\} \quad \mathtt{x\ :=\ x\ +\ 1} \quad \{?\}$$

- Strongest Postcondition:

$$\mathrm{sp}(\mathtt{x} := e, P) =$$
$$\exists \mathtt{x}_0 \,.\, \mathtt{x} = e[\mathtt{x}/\mathtt{x}_0] \wedge P[\mathtt{x}/\mathtt{x}_0]$$

  where $\mathtt{x}_0$ is the "old" value of $\mathtt{x}$

- Example:

$$\mathrm{sp}(\mathtt{x} := \mathtt{x} + 1, (\mathtt{x} < 10)) =$$
$$\exists(\mathtt{x}_0 \,.\, \mathtt{x} = \mathtt{x}_0 + 1) \wedge (\mathtt{x}_0 < 10)$$

Formalisms for Assertions and Specifications

$x \geq 10$

$p \neq \texttt{null}$

Questionnaire:

- In which language are assertions written?
- When do assertions have to hold?
- If all assertions hold, is the program correct?

Limitations of Assertions

If REQ $= 1$, then eventually ACK $= 1$

Questionnaire:

- Can this be expressed as an assertion in C or Java?
- Can we use testing to find such a violation?
- How can this assertion be violated?

Temporal Logic

Amir Pnueli

Linear Temporal Logic

- Temporal operators
  - always
  - eventually
- Describes how executions evolve

Temporal Logic

Amir Pnueli

Linear Temporal Logic

- Temporal operators
  - always
  - eventually
- Describes how executions evolve

`AG (REQ` $\Rightarrow$ `F ACK)`

Model Checking

Edmund Clarke
Allen Emerson
Joseph Sifakis

Basic idea:

- Assertions in temporal logic
- Programs with finite state space
- *models* instead of programs
- all reachable states are inspected!
- also works for concurrent models

T

$$s \xrightarrow{\mathbf{T}} s'$$

T

$$s \xrightarrow{\quad} s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

T

$$s \overset{\curvearrowright}{} s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2\rangle$$

(T: operational semantics of program or circuit)

T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2\rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:

I "starting states"

$\neg P$ "bad states"

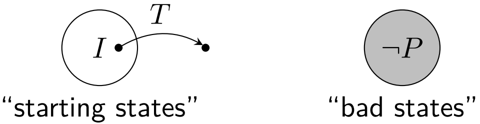
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



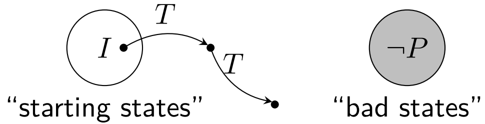
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2\rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



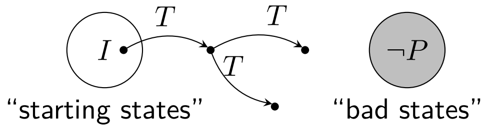
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1 \rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2 \rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



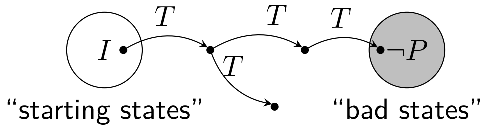
T

$$s \longrightarrow s'$$

$$\langle pc \mapsto 2, x \mapsto 1\rangle \xrightarrow{T} \langle pc \mapsto 3, x \mapsto 2\rangle$$

(T: operational semantics of program or circuit)

The **Model Checking** problem:



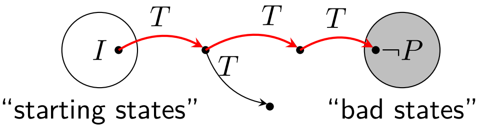
T

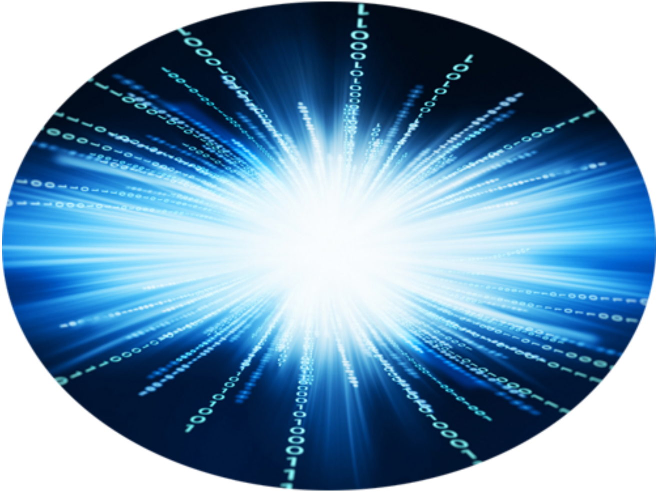
$$\begin{tikzcd} s \arrow[r, bend left] & s' \end{tikzcd}$$

$$\begin{tikzcd} \langle pc \mapsto 2, x \mapsto 1 \rangle \arrow[r, bend left, "T"] & \langle pc \mapsto 3, x \mapsto 2 \rangle \end{tikzcd}$$

(T: operational semantics of program or circuit)

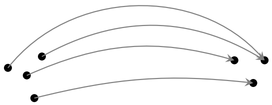
The **Model Checking** problem:



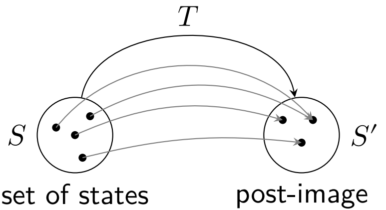


State Space Explosion

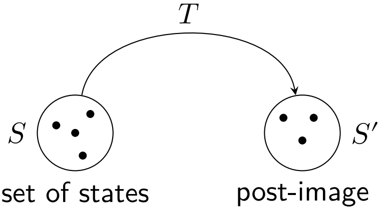
Why explore states one by one?



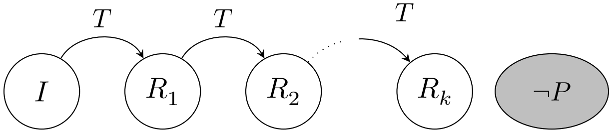
Why explore states one by one?



Why explore states one by one?



$$S' \quad = \quad T(S) \stackrel{\text{def}}{=} \{s' \mid T(s, s') \wedge s \in S\}$$



T
T
T
I
R_1
R_2
R_k
$\neg P$

How do we efficiently represent sets of states?

Ken McMillan

Basic idea:

- use logic to represent states
- implementation: SMV model checker

Logical formulas to represent states

$$\underbrace{V}_{\substack{\text{program variables,} \\ \text{registers, latches,} \\ \text{signals, ...}}}$$

Logical formulas to represent states

$$F(\underbrace{V}_{\text{program variables, registers, latches, signals, ...}})$$

Logical formulas to represent states

$(x > 0)$ represents $\{s \mid s(x) > 0\}$

And what about transitions?

Binary Relations!

$$T(\,V,\,\underbrace{V'}_{\text{target states}}\,)$$

And what about transitions?

Binary Relations!

$(x' = x + 1)$ represents $\{\langle s, s' \rangle \mid s'(x) = s(x) + 1\}$

And what about transitions?

Binary Relations!

$$\underbrace{(x' = x + 1)}_{\texttt{x++}} \quad \text{represents} \quad \{\langle s, s' \rangle \mid s'(x) = s(x) + 1\}$$

R

R

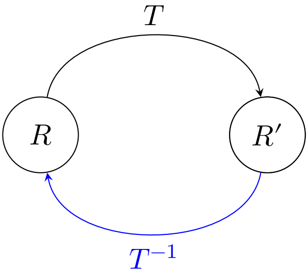
$$\begin{tikzcd} R \arrow[rr, bend left, "T"] & & R' \end{tikzcd}$$

$$R'(V') \quad \stackrel{\mathsf{def}}{=} \quad \exists V \,.\quad R(V) \quad \wedge \quad T(V, V')$$

$$\begin{tikzcd}
R \arrow[rr, bend left, "T"] & & R' \arrow[ll, bend left, "T^{-1}"]
\end{tikzcd}$$

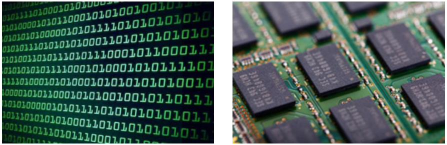
$$R'(V') \stackrel{\text{def}}{=} \exists V \,.\; R(V) \;\wedge\; T(V, V')$$

$$R(V) \stackrel{\text{def}}{=} \exists V' \,.\; T(V, V') \;\wedge\; R'(V')$$



$$
\begin{aligned}
R'(V') &\overset{\mathsf{def}}{=} \exists V\,.\quad R(V) \;\wedge\; T(V,V') \\
R(V) &\overset{\mathsf{def}}{=} \exists V'\,.\quad T(V,V') \;\wedge\; R'(V')
\end{aligned}
$$

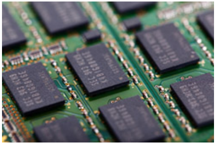
(Note the similarity to strongest postcondition)



T

(transition relation)

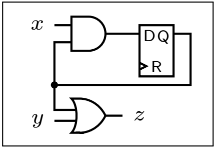
```
if (x>0) {
  x = x - 1;
} else {
  x = x + 1;
}
```



T

(transition relation)

```
if (x>0) {
  x = x - 1;
} else {
  x = x + 1;
}
```



T

(transition relation)

```
if (x>0)
  x = x - 1;
else
  x = x + 1;
assert (x≥0);
```

$$T(\langle pc, x\rangle, \langle pc', x'\rangle)$$

$$\wedge \left(\qquad \right)$$

```
1:  if (x>0)
2:    x = x - 1;
3:  else
4:    x = x + 1;
5:  assert (x≥0);
```

$$\underbrace{\hspace{6em}}_{T(\langle pc, x\rangle, \langle pc', x'\rangle) \stackrel{\text{def}}{=}}$$

$$\bigwedge \left((pc = 1) \ \wedge \ (x > 0) \quad \Rightarrow \quad (pc' = 2) \ \wedge \ (x' = x) \right)$$

```
1:   if (x>0)
2:     x = x - 1;
3:   else
4:     x = x + 1;
5:   assert (x≥0);
```

$$T(\langle pc, x\rangle, \langle pc', x'\rangle) \stackrel{\text{def}}{=}$$

$$\bigwedge \left(\begin{array}{ccccccc} (pc = 1) & \wedge & (x > 0) & \Rightarrow & (pc' = 2) & \wedge & (x' = x) \\ (pc = 1) & \wedge & \neg(x > 0) & \Rightarrow & (pc' = 4) & \wedge & (x' = x) \\ & & & & & & \end{array} \right)$$

```
1:  if (x>0)
2:    x = x - 1;
3:  else
4:    x = x + 1;
5:  assert (x≥0);
```

$$T(\langle pc, x\rangle, \langle pc', x'\rangle) \stackrel{\text{def}}{=}$$

$$\bigwedge \left(\begin{array}{lllllll} (pc = 1) & \wedge & (x > 0) & \Rightarrow & (pc' = 2) & \wedge & (x' = x) \\ (pc = 1) & \wedge & \neg(x > 0) & \Rightarrow & (pc' = 4) & \wedge & (x' = x) \\ (pc = 2) & & & \Rightarrow & (pc' = 5) & \wedge & (x' = x - 1) \\ & & & & & & \end{array} \right)$$

```
1:  if (x>0)
2:    x = x - 1;
3:  else
4:    x = x + 1;
5:  assert (x≥0);
```

$$T(\langle pc, x\rangle, \langle pc', x'\rangle) \stackrel{\text{def}}{=}$$

$$\bigwedge \left(\begin{array}{lcccccc} (pc = 1) & \wedge & (x > 0) & \Rightarrow & (pc' = 2) & \wedge & (x' = x) \\ (pc = 1) & \wedge & \neg(x > 0) & \Rightarrow & (pc' = 4) & \wedge & (x' = x) \\ (pc = 2) & & & \Rightarrow & (pc' = 5) & \wedge & (x' = x - 1) \\ (pc = 4) & & & \Rightarrow & (pc' = 5) & \wedge & (x' = x + 1) \end{array} \right)$$

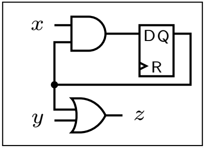
```
1:   if (x>0)
2:     x = x - 1;
3:   else
4:     x = x + 1;
5:   assert (x≥ 0);
```

$$T(\langle pc, x\rangle, \langle pc', x'\rangle) \stackrel{\text{def}}{=}$$

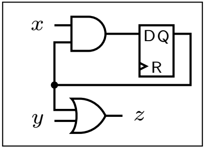
$$\bigwedge \left(\begin{array}{llllll} (pc = 1) & \wedge & (x > 0) & \Rightarrow & (pc' = 2) & \wedge \quad (x' = x) \\ (pc = 1) & \wedge & \neg(x > 0) & \Rightarrow & (pc' = 4) & \wedge \quad (x' = x) \\ (pc = 2) & & & \Rightarrow & (pc' = 5) & \wedge \quad (x' = x - 1) \\ (pc = 4) & & & \Rightarrow & (pc' = 5) & \wedge \quad (x' = x + 1) \end{array} \right)$$

$$P(V) \stackrel{\text{def}}{=} (pc = 5) \Rightarrow (x \geq 0)$$

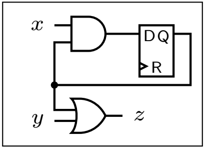
$$I(V) \stackrel{\text{def}}{=} (pc = 1)$$



$T(V, V')$



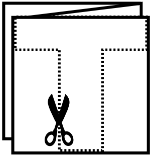
$$T(V, V') \stackrel{\text{def}}{=} (Q' \Leftrightarrow (x \wedge Q)) \wedge (z \Leftrightarrow (y \vee Q))$$

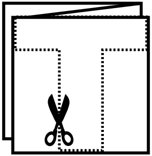


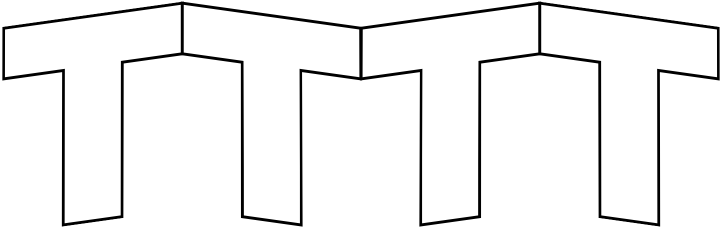
$$\underbrace{\qquad\qquad}_{T(V,V')} \stackrel{\text{def}}{=}$$

$$(Q' \Leftrightarrow (x \wedge Q)) \wedge (z \Leftrightarrow (y \vee Q))$$

$$\begin{aligned} P(V) &\stackrel{\text{def}}{=} z \\ I(V) &\stackrel{\text{def}}{=} \mathsf{Q} \end{aligned}$$







The C Bounded Model Checker (CBMC)

Daniel Kröning

- Checks C programs for assertion violations
- Checks only k loop iterations
- Converts T into propositional logic

`https://www.cprover.org/cbmc`
(and that's where our journey started)

Course Outline

- Part 1: Assertions and Testing
  - Programming and Reasoning with Assertions
  - Testing and Coverage Metrics
  - Automated Test-Case Generation

  March

- Part 2: Logic and Reasoning
  - Propositional Logic (and SAT Solvers)
  - First-Order Logic (and SMT Solvers)
  - Hoare Logic
  - Temporal Logic

  April

- Part 3: Automated Verification
  - SMV (Symbolic Model Checking)
  - SPIN (Partial Order Reduction)
  - Bounded Model Checking of C Programs

  May

Lecture, Exercises and Exam

- Lectures: Wednesday and Friday, 9:30am-11am
  - Recordings on LectureTube (see TUWEL)
- VU = lecture + exercises
  - Application of verification and testing tools
  - Pencil & paper homeworks
  - Exercises form 50% of the grade
- 3 exercises (TUWEL)
  - Assertions/Testing/Coverage:
    Released March 22, due April 24
  - Hoare Logic and BMC:
    Released April 24, due May 24
  - Temporal Logic & Automated Reasoning:
    Released May 08, due May 29
- Written Exams (in-person):
  - June 12, 9:00am to 11:00am
  - end of September/beginning of October