# Specification and Verification of a Multicopter Flight Controller

•••

March 18th, 2019

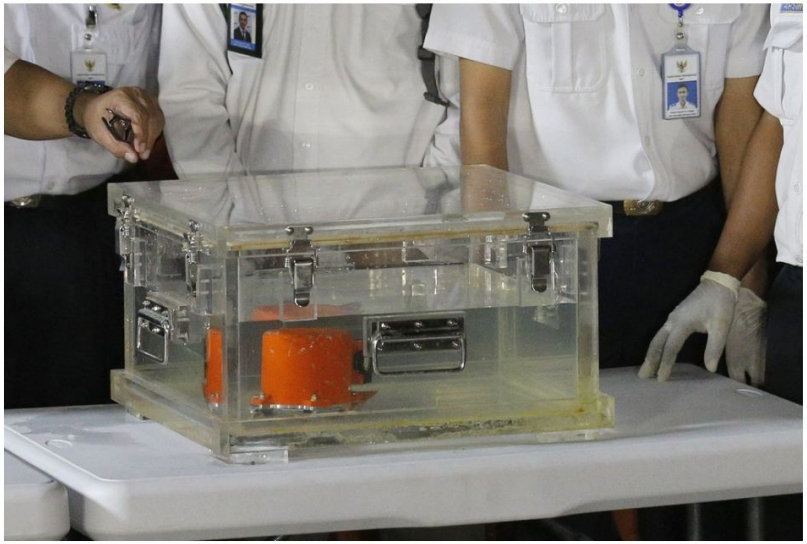
Rafael Bachmann

# Boeing 737 MAX 8: Two related Incidents, Avionics suspected

Boeing & Aerospace | Business

## Pilots struggled against Boeing's 737 MAX control system on doomed Lion Air flight

Originally published November 27, 2018 at 1:30 pm | *Updated November 28, 2018 at 4:07 pm*



*The recovered flight-data recorder, the so-called "black box," of the Lion Air jet that crashed into the sea Oct. 29 is displayed during a... (Tatan Syuflana / The Associated Press)* **More**

**Data from the fatal Oct. 29 flight that killed 189 people, and from the prior day's flight of the same jet, raises questions about three factors that seem to have contributed to the crash.**

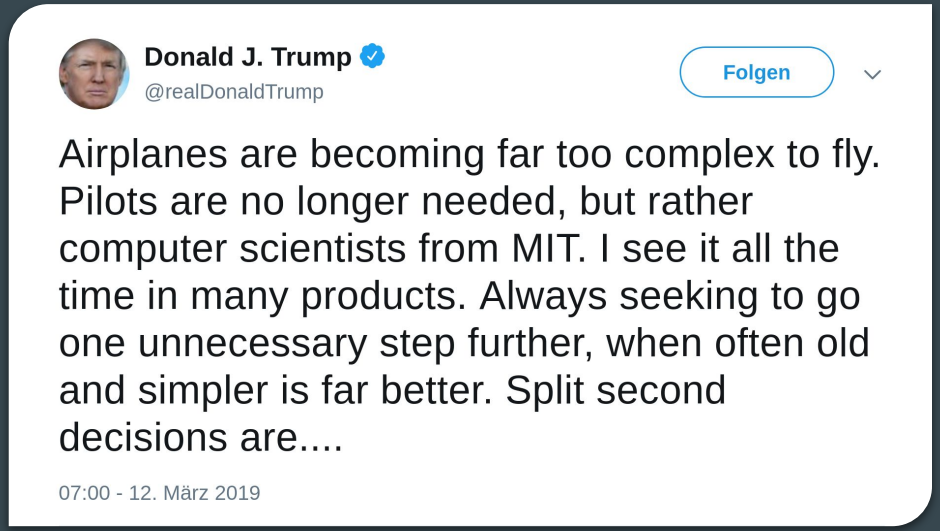
By Dominic Gates

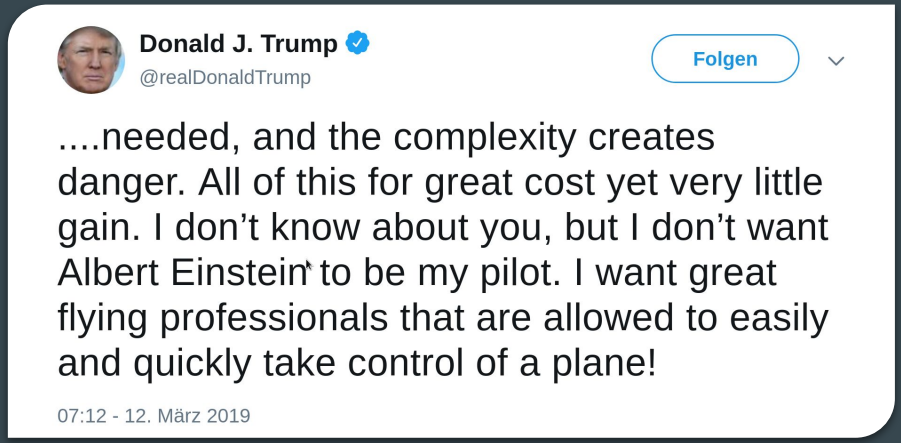
*Seattle Times aerospace reporter*

Share story

Article in Seattle Times

# Reactions

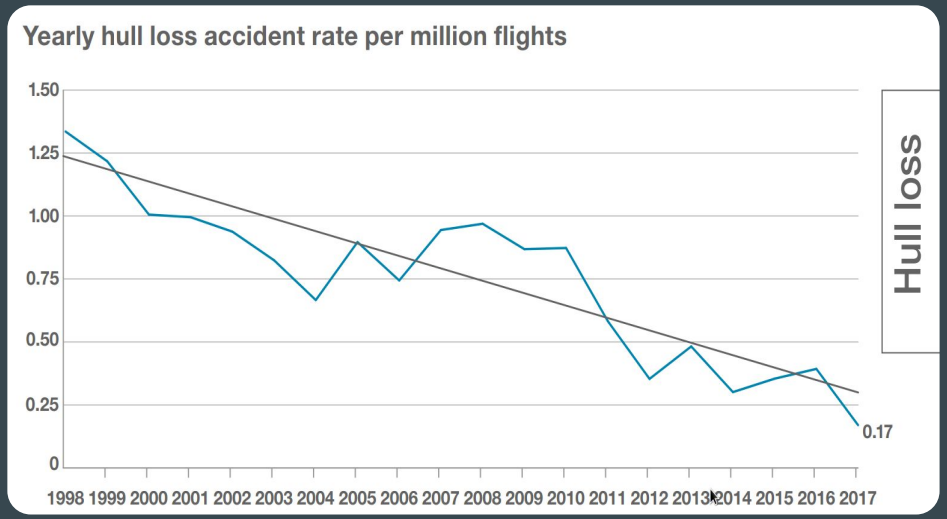


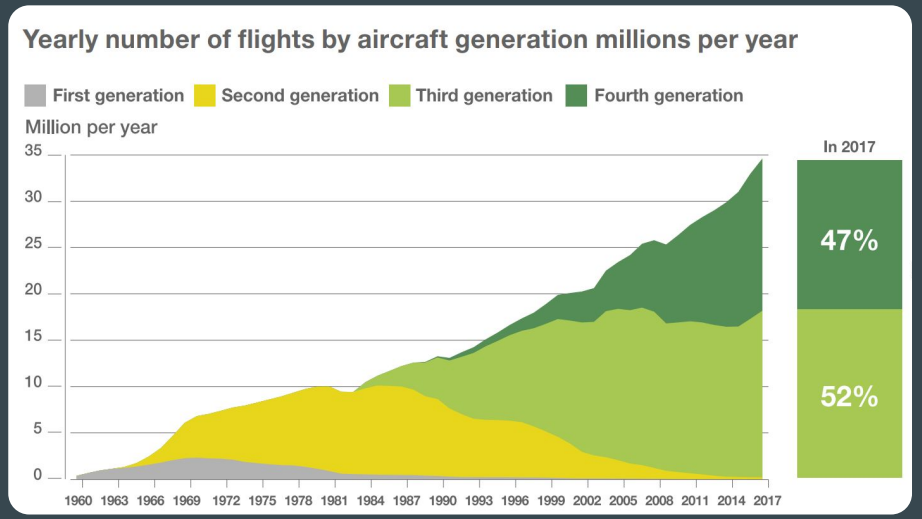




"[...] complexity creates danger. [...]"

https://twitter.com/realDonaldTrump/status/1105468569800839169
https://twitter.com/realDonaldTrump/status/1105471621672960000

# Avionics Safety Trends




Accident rate is declining rapidly
Number of flights is increasing
**Avionics complexity is increasing**

A Statistical Analysis of Commercial Aviation Accidents 1958-2017 (Airbus)

# Pervasive Embedded Systems

- Transport
- Infrastructure
- Energy
- Healthcare

# Example: Multicopter Flight Control

DJI F450 Quadcopter: A versatile & elegant Multirotor

Representative of a safety-critical embedded system:

- Realtime Requirements
- Sensors
- Control System

Dangers:

- Loss of vehicle
- In-flight collisions with other aircraft
- Operator/Bystander injuries

# Example: Multicopter Flight Control

DJI F450 Quadcopter: A versatile & elegant Multirotor

## Flight Controller tasks:

- Radio Communication
- Motor Drivers
- IMU Sensors
  - Sensor Fusion
  - Timeseries “smoothing”
- Control System
  - Cascaded PID control
- Energy Management
- Waypoint Navigation (GPS, …)
- Obstacle Avoidance
- …

# Ensuring Safety and Correctness

## Software Testing

- Successful and widely used, accepted
- Tool and language support
- Promotes good design
- In embedded context: mocking of the environment may be required

“Program testing can be used to show the presence of bugs,
but never to show their absence!”

— Edsger Dijkstra, “Notes on Structured Programming”

# Ensuring Safety and Correctness

## Formal Verification

- Prove adherence of a program to its specification
- Formally prove absence of bugs, (requires specification)
- Partial specification possible
- Relatively low acceptance in industry

“Correctness of computer programs is the fundamental concern of the theory of programming and of its application in large-scale software engineering.”

— Tony Hoare, “The Verifying Compiler:
A Grand Challenge for Computing Research”

# Goals

- Ensure safety of sample flight controller
- Evaluate applicability of state-of-the-art static analysis
  - How expensive is the process?
  - Is it feasible for embedded projects?
  - How large is the confidence gain?
  - What are areas for improvement?

# Approach:
# Verify flight controller (deductive Verification)

# Requirements & Formal Specification

## Requirements elicitation

Abstract requirements

- Natural language
- Domain Knowledge
- Final approval criteria

## Formal Specification

Specific system requirements

- Formal “language”
- Pre- and Postconditions
- Invariants, Axioms, Ghost Code, ...
- More than regular C syntax:
  - quantifiers, ranges, special types

# ACSL: ANSI/ISO C Specification Language

Formal specification of binary search algorithm:

```
/*@ predicate Sorted{L}(value_type* a, integer n) =
        \forall integer i, j; 0 <= i < j < n ⇒
            a[i] <= a[j];
*/
/*@ requires Sorted(array, n);
    requires \valid_read(array + (0 .. n - 1));
    assigns \nothing;
    ensures result: \result <==>
        \exists integer i; 0 <= i < n && a[i] == val;
*/
bool binary_search(const int* array, size_t n, int val);
```

# Deductive Verification

## Verification

- Transform program and specification to logical formulas
  - Requires formal model of the implementation language:
  - Deduction Rules for assignment, conditionals, loops, ...
- Attempt to prove these formulas using theorem provers
  - If successful: specified property always holds

# Discharging Verification Conditions

- ensures
  valid_transition:
    \old(internal_state) ≡ ARMED ⇒
    \at(internal_state, Post) ≡ ARMED ∨
    \at(internal_state, Post) ≡ DISARMING;

# Results on Flight Controller Software

- Some components completely proven

  with comprehensive specification
- Some components not verified!
- Problems with floating point arithmetic
- No problems with pointers/arrays, integers, loops, resources
- Discovered issues:
  - Possible division by zero in PID controller
  - Potential (signed!) overflow in `abs(x)` function

# Conclusions from Verification work

- Works well for sorting, search, data structure manipulation
- Standard library functions
  - “ACSL by Example” by Fraunhofer (repo)
- State machines, value ranges, loop termination works well
- Automatic annotation of possible runtime errors (undefined behavior) works really well

# Conclusions from Verification work

- Specification often similar to implementation
- Floating point arithmetics create difficulty
- Ghost code and axioms are powerful
  - But can introduce unsoundness
- Abstract properties hard to specify
  - Timing properties
  - Asymptotic runtime properties
  - Non-functional properties

# Conclusions from Verification work

- Some idioms of safety-aware industrial C code
  are unsupported by Frama-C/ACSL
  - Function-local statics, non-literal array length initializers
  - Discussion/doubt how/if they should be supported
- Dynamic memory allocation, non-constant loop bounds, scheduling/multithreading difficult
  - Not used in many embedded systems anyway

# Conclusions from Verification work

- Meaningless specifications:
  - Memory mapped peripherals
  - Communication protocols

```
// 1 second WDT timeout (e.g. reset in < 1 sec or a reset
WDOG_TOVALH = 0x006d;
WDOG_TOVALL = 0xdd00;

// watchdog timer at 7.2MHz
WDOG_PRESC  = 0x400;

// Enable WDT.
// This must happen in one single write to WDOG_CTRLH
WDOG_STCTRLH |= WDOG_STCTRLH_ALLOWUPDATE |
    WDOG_STCTRLH_WDOGEN | WDOG_STCTRLH_WAITEN |
    WDOG_STCTRLH_STOPEN | WDOG_STCTRLH_CLKSRC;
```

# Ensuring Memory Mapping Contracts

- Solution: Generate specification from formalized datasheet information
  - ARM: System-View-Description (SVD)
  - Read/Write registers
  - Finite number of valid configurations
- **Ensure implicit memory-mapping contracts are not violated!**

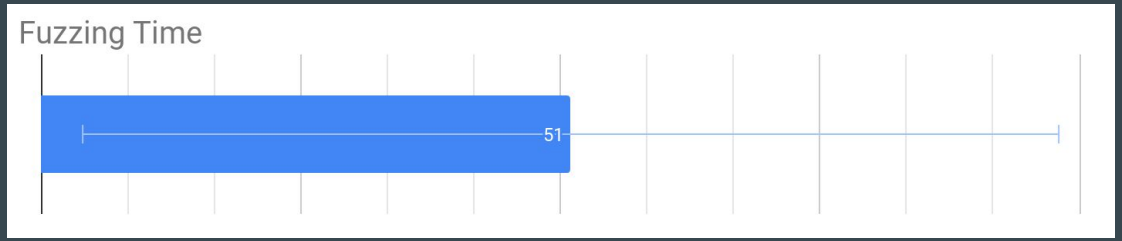
# Dynamic Techniques

- Fuzzing in embedded context:

  Fuzz more than just function signatures:
  - Memory-mapped peripherals (timers)
  - Sensor measurements
  - Communication protocols
- Random fuzzing may be unlikely to reach all states
  - Kalman filter: static data depending on estimation quality
  - PID: Integral term
- Automatically generate fuzz values from formal datasheet?

# Example: Fuzzing vs. Deductive Verification

- Discovered issue:
  Division by zero prevented only by valid but implicit assumption
  - Trigger: timer register read yields identical value
    in successive iterations
- Frama-C/WP-rte: less than 1sec proof time, unannotated program
- Fuzzer (purpose-built):
  - Time to failure: average 51sec, standard deviation 47sec



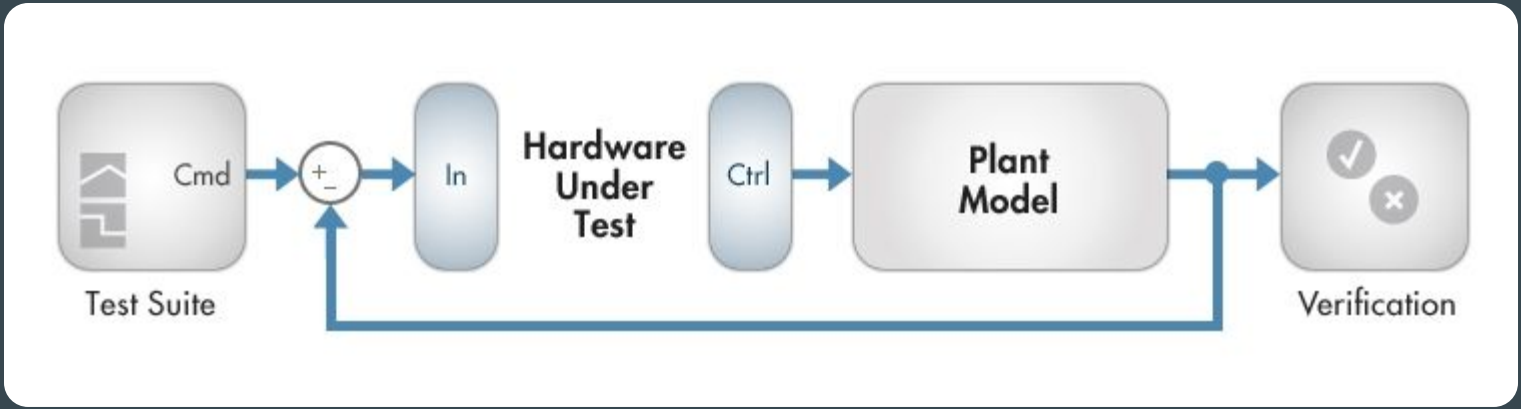
# Mocking/Hardware-In-The-Loop

- Software Testing on embedded targets requires mocking:
  - Memory mapped peripherals
  - Sensor measurements
  - Communication protocols

# Mocking/Hardware-In-The-Loop

- Next step: “Mocking” of real world
  - Ideal for control systems requiring a system model
  - Simulation allows to test likely and unlikely scenarios
  - Allows to test system before hardware is finished
  - Allows to train operators in simulation with actual interface
  - Running “Test Suite” is costly (real time simulation)

# Mocking/Hardware-In-The-Loop



# Conclusions

- Challenges in embedded static & dynamic analysis:
  - Interaction with environment
  - Implicit contracts
- Challenges in deductive verification:
  - Floating point arithmetic
  - Low level of abstraction

# Conclusions

- Static and dynamic techniques should be used together
  - Not fragmentation:

    Approach problem from different angles
  - Not “more work”:

    Faults can be found before they become failures