Static Code Analysis for Safer IoT Development

Basile STARYNKEVITCH - basile.starynkevitch@cea.fr

Commissariat à l'Énergie Atomique et aux énergies alternatives
CEA - LIST, LSL (Palaiseau, France)

October, 11th, 2018, Rome

commit **572de8b2a2437f60**

This CHARIOT project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 780075.

Overview

1. Introduction
2. Importance of *source* code
3. Simpler CHARIOT approach to static analysis

Introduction

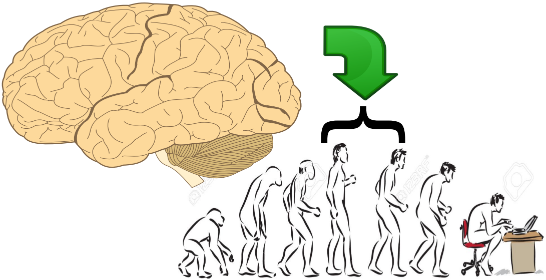
all **opinions are *only* those of the author**, Basile STARYNKEVITCH

`https://xkcd.com/1425/`

- some tasks or goals look simple but are not that simple....
- some apparently *very similar* tasks or goals are very difficult, or impossible... (maybe intractable)
- some very close tasks are even provably impossible (undecidable)

NB: *it should be ten, not five, years!*

we are unfit for {IoT, programming, project management, ...}



sub-image sources:

- "brain image": Wikimedia *Human Brain*
- "brace": rotated { from some computer font
- "green arrow": OpenClipart `arrowgonext`
- "evolution": `1234f.com` image 59774943 by Monica Roa

Our (still prehistoric) **brain is unfit** for: IoT design, programming, computer science & math, management of large projects... (but suitable -"optimized" by evolution- for prehistoric hunting and gathering).

Miller's law (1956): our working memory is limited to only 7 ± 2 **chunks**

my dynamic memory allocator : the best one ☺

```
#include <stdlib.h>
#include <errno.h>
void* malloc(size_t siz) {
    if (siz > 0)
        errno = ENOMEM;
    return NULL;
}
```

A joke for geeks :
it follows the letter, not the spirit, of the C11 (cf n1570 §7.22.3) and POSIX standard[s], since it *always* fails by giving the *null* pointer.

NB. I probably could make a good allocator, but it would take me years of work. The existing ones are enough for me (and probably you), but they are complex and make trade-offs. Specifying some properties of `malloc` is easy, implementing a good enough one is hard. Expliciting most "good" properties is nearly impossible.

Tackling *essential* complexity with *abstractions*

Cf ***The mythical man-month*** *: essays on Software Engineering* by F. Brooks (1975, 1995); includes *No silver bullet*.

Brooks's law: *Adding manpower to a late software project makes it later*.

If 1 woman can make a baby in 9 months,
9 women won't make a baby in 1 month.

- even perfect program verification can only establish that a program meets its specification.
- the essence of building a program is in fact the **debugging of the specification**

Heisenbugs. ⇒ The number of bugs tends to a non-zero asymptote.

Leaky abstractions

Even with our abstractions, the complex reality of our systems or systems of systems (e.g. IoT ones) don't fit well in our brains.

The law of leaky abstractions (Joel Spolsky, 2002) :

> ***All** non-trivial **abstractions**, to some degree, **are leaky**;*
>
> *or, (for G.Schwarz) "incomplete", or (for A.Zwinkay) "unsuitable"*
>
> ⇒ *Abstractions fail. (our `malloc` example leaks a lot!)*

- so ISO9001 QA is unsuitable for software development; cf "Joel Test" (so the software industry: Google, Facebook, MicroSoft, . . . don't follow ISO9001).
- ⇒ **state-of-the-art static source code analysis tools are *not* very effective in detecting security vulnerabilities** (alone) (Goseva-Popstojanova & Perhinschi 2015)
- But **these tools can be *helpful*** (even when they are *simple*)
- Complex static analysis tools need a detailed and formalized specification (and check difficult properties). Such specifications are hard to write.

Trade-off in static source code analysis

There is a trade-off when doing static source code [1] analysis:

- **strong formal-methods** based static analysis (cf VESSEDIA) à la *Frama-C* :
    1. do a “sound” analysis: **may** *sometimes* **give strong guarantees** on the *proven* properties (usually simple ones, or very specific ones) of the IoT software.
    2. sometimes, the analysis does not concludes anything, or times out
    3. **requires a strong formalization of the specifications**
    4. *so is costly to use* (additional skills required)
- **weak heuristical** based static analysis (the CHARIOT approach) - like Coverity or Clang-analyzer
    1. **unsound** analysis - no promises at all (could say “ok” for a buggy program, or “not-ok” for a good program) and **no guarantee**
    2. less (or incomplete, or missing) formalization of specifications
    3. perhaps **simpler** to use (but weaker)
    4. won’t always work (but might require less skills from the developer)

[1] All static analysis in this talk is on source code!

strong static analysis

The VESSEDIA way:

- **significant effort on formal specifications** (and deeply understanding the real-world problem). The "source code" is not only C, but also a (mathematical) formalization of the specification (e.g. in ACSL).
- **strong guarantees** on the *proven* properties of the software. Unit testing can become useless. Whole-system test is still essential.
- sound tool (when ok is given, the program "is" bug-free w.r.t. specification)
- **costly approach** (e.g. $> \times 30$ "traditional software development").
- required *additional skills* (to formalize the specification)
- add constraints to programmers (e.g. coding style forbidding `malloc`)
- the analyzing tool (Frama-C code prover) might fail. Then we change the code or the specifications, until the code is proven correct.
- suitable (and often *suggested* by regulations, such as DO-178C) for **life-critical IoT systems**

weak static analysis

The simpler CHARIOT approach:

- only *very simple* or *trivial* properties are detected. Unit testing is still absolutely needed.
- “cheaper” approach: less formalization on specifications
- can be built above existing ***cross*-compiler technology** (GCC)
- so less disruptive to use (e.g. just add cross-compiler options for some GCC plugin)
- unsound tool (wrong negative: tool gives OK on buggy program)
- only modest results can be expected (many false positives and missing alarms)
- software failures should be acceptable and expected
- still requires a “whole program” capable tool
- suitable for **non-critical IoT systems** (but should be avoided for safety-critical IoT systems)

The halting problem and Rice's theorem

The **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running (i.e., halt) or continue to run forever.

from Wikipedia

Rice's theorem states that *all* non-trivial, semantic properties of programs are undecidable.
⇒ There exists no automatic method that decides with generality non-trivial questions on the behavior of computer programs.

from Wikipedia

The halting problem is **provably unsolvable** ☹

It is the essential limitation of static analysis

Overview

Source code - socially

The **source code** is the ***preferred*** **form on which software developers *work***.

Software development is a "social" activity.
We write source code for *humans* [2], not mostly for computers.

Source code may include building scripts (e.g. `Makefile`-s, shell scripts or tests)
Practical importance of **version control** e.g. **`git`** [3]

[2] Perhaps just us next month!

[3] `git` was first developed by Linus Torvards in 2005.

Source code - examples

- hand-written C source and header files (before preprocessing): `*.c` and `*.h`
- build automation configuration: `Makefile`, etc...
- generators for *generated* files, perhaps as simple as:

```
const char bismon_timestamp[]="Sun 30 Sep 2018 06:13:23 PM MEST";
const unsigned long bismon_timelong=1538324003L;
const char bismon_lastgitcommit[]=
      "f157d8ecbdde start coding emit_jsstmt°basiclo_while";
const char bismon_lastgittag[]="heads/master";
const char bismon_checksum[]="8452084c28c0580f65a57f8b404f9bc7";
const char bismon_directory[]="/ssdhome/basile/bismon";
const char bismon_makefile[]="/ssdhome/basile/bismon/Makefile";
```

which is generated by a few lines (e.g. of some `Makefile`).

Practical importance of *meta-programming*: using or writing scripts or programs emitting C code [4] (parser generators à la `bison`, glue generators à la SWIG, ...)

Most IoT projects are likely to have some *generated* C code in 2018!

Your vendor IDE might not be able to handle them. So use a real build automation tool like `make`, `ninja`....

[4] Or any other code consumed by a compiler or interpreter!

source code for your GCC compiler

Your GCC compiler (e.g. `gcc 8.2` in october 2018, see **gcc.gnu.org**) consumes C files (and many other languages: C++, Objective C, Fortran, Go, perhaps D) and does not care if that “source” file is generated or not. It emits [5] object code.

⇒ *source code* does not mean the same thing for compilers and for developers.

The C compiler proper `cc1` sees quite early *preprocessed* text (and have skipped all your comments!).

[5] Actually, the `cc1` emits assembler code, but `gcc` runs `as` after `cc1`.

Showing and sharing your source code

- in a formal process (e.g. avionics): the (proprietary) source code is "audited" and "evaluated" by some *external* entity (€€ costly €€).
- informally, by some distant colleague (outside of your team) - peer reviews
- informally, code reviews in your team (biased, since they know your code)
- open source / free software communities (reusability of code chunks)

Linus' law: **given enough eyeballs, all bugs are shallow** (1999, mostly true today).

⇒ source code analysis is often an *aid* to code reviewers or developers

take-away message:

showing your source code *increases* its value and quality

even for pre-α quality prototype code;

hiding source code should be "discouraged" and is ineffective and will become "counter-productive".

When subcontracting software development, request the source code!

IoT industry not used to show source code

That will cost thousands of human lives and billions of €

Hiding source code is a **bad practice** that should change.

Static source code analysis is an aid to external code reviewers and developers (might help to avoid “code smells”)

The IoT industry [6] should get used to show the source code!

Q: How many human lives (killed by bugs), how many billions € of loss, are needed to make that generally happen and become standard practice?

Human language -e.g. in comments- matter too! future DECODER H2020 project mixing natural language processing, machine learning, and static source code analysis

[6] Actually, all software intensive industries!

Overview

Simpler CHARIOT approach to static analysis

A “simpler” approach, because formal methods expertise is *not* required (like what VESSEDIA partners have and need).

Expected audience of the CHARIOT static analyzer, tentatively called *bismon* [7]

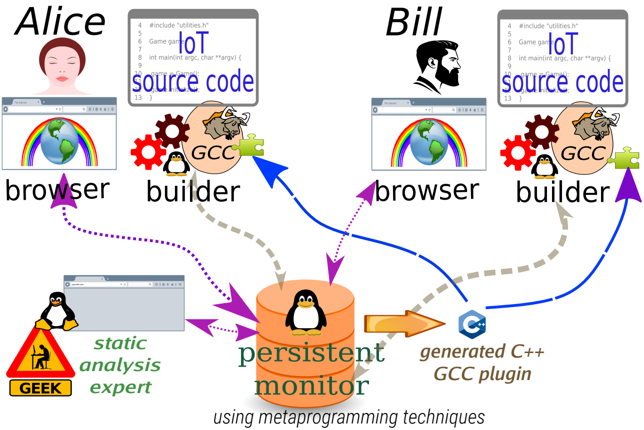
- a **small team of developers** (e.g. 2 to 10 persons) working on the *same* IoT firmware. That team is **trustful** (and its members trust each other) and **well behaving** (no malicious behavior).
- a reasonably sized firmware project source code (e.g. less than 300 KLOC)
- one (or a few) *simple* program properties to check (e.g. stack overflow)
- the firmware developers are *all* using **Linux** and a **recent GCC cross-compiler** (accepting GCC plugins) : GCC 8 (and soon GCC 9) [8] at least.
- these developers have all configured their build system to use the *same* (given) GCC plugin. The C++ code of that GCC plugin would be generated by *bismon*.

[7] A bad temporary name, please suggest constructively a better one.

[8] The version of GCC matters a lot and is important for plugins. All IoT developers use the *same* GCC cross-compiler on their Linux machine.

The *bismon* persistent monitor

Persistence : the monitor keep its data [9] (by loading it at start and dumping it before exiting) from one run to the next one (Typically the monitor would run the whole day).



[9] Notably intermediate results related to static analysis.

bismon is still work in progress

Unreleased, but incomplete pre-α GPLv3+ code on **github.com/bstarynk/bismon** (in september 2018):

- TODO: choosing a simple illustrative open-source firmware example (with partners)
- TODO: choosing a simple static analysis goal (probably stack overflow) to focus on
- runtime (naive GC) and **persistence** : working
- multi-threaded agenda machinery with tasklets : working
- client HTTP (and management of contributors) with login form : mostly working
- meta-programming approach:
  - (non-bootstrapped) generation of internal C code
  - generation of JavaScript (in browser), half done
  - TODO: generation of HTML5
  - TODO: generation of GCC plugins in C++ (leverage on GCC MELT experience)
  - TODO: analysis of GCC code (to ease plugins generation)
- TODO: “single page application” web interface (above CodeMirror & JQuery) - so *bismon* is not yet usable in september 2018 by others than me