Security & safety of autonomous vehicles: a case study with TrustInSoft Analyzer

Alexandre Hamez, **David Wagner**, Fabien Lheureux, Stéphane Zimmermann, **Benjamin Monate**

June 3rd, 2019

TrustInSoft

TrustInSoft

- Startup incorporated in 2013
- Created by former CEA LIST researchers
    - Initial Frama-C creators
    - Frama-C lead developers for 10 years
- Application of formal methods to sensitive software
    - Safety: aero, auto, energy, rail
    - Cyber-Security: aero, auto, cryptography, cyber, defense, telco
- Software publisher of `TrustInSoft Analyzer`
- Technical expertise in source code assessment
- Training

TrustInSoft Analyzer

- TIS-Kernel: fork of Frama-C with stability in mind
  - C with non-standard extensions
  - Scalability and maintainability
  - *Value*: evolutions toward predictability and ease of usage
  - *WP*: mostly unchanged but porting evolutions from public Frama-C releases
  - Increase precision and scalability of dependency analysis
  - Available under GPL
- TIS-Analyzer: companion tools and methodologies
  - C++ support from C++98 up to almost C++17
  - Support for I/O modeling (filesystem, network)
  - Support for automatic analysis configuration
  - Web-based GUI dedicated to efficient methodologies
  - JSON API for continuous integration
  - Training, methodology and standard materials

TrustInSoft Analyzer on a test suite

Required inputs :

- Source code
- Tests suite (software)

Outcome :

- Precision: each alarm is a true bug
- Minimizing the time to setup up the analysis
- Easy to integrate in the development process
- Possibility to combine with fuzzers

Finds tons of serious bugs with minimal efforts

TrustInSoft Analyzer with generalized tests

Using generalized values on test suites to increase coverage

```c
/* Standard test driver */
void test_driver(void)
{
  int a, b, c;
  a = 22;
  b = 17;
  c = is_prime_factor(a, b);
  return;
}
```

```c
/* Generalized test driver */
void test_driver(void)
{
  int a, b, c;
  a = tis_interval(0, INT_MAX);
  b = tis_interval(0, INT_MAX);
  c = is_prime_factor(a, b);
  return;
}
```

Generalized `test_driver`:

- Equivalent to $2^{31} * 2^{31} = 2^{62}$ tests cases
- Explore all the combinations of `a` and `b` at the same time

TRUST IN SOFT

easy MILE

Challenges

- Incremental verification is a must: standard metrics do not apply
- Fixing bugs takes more time that finding them: process integration
- Continuous Integration of formal verification
- Inherently, most of the C++ data structures are relational:
    - Difficult when the analysis is not precise
    - Incremental generalization allows to counterbalance this difficulty

EasyMile

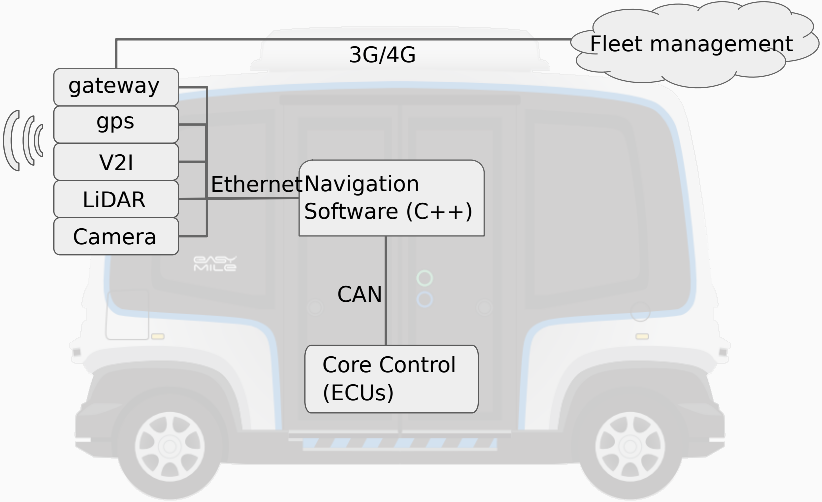
EasyMile

- Founded in 2014
- Software vendor
- Vertical solution: from embedded to fleet management software
- 170 employees
- In 5 years:
  - 230 deployments
  - 600 000 km

Development process for autonomous vehicles at EasyMile

- As opposed to regular cars:
    - We have few ECUs
    - The logic is implemented on “off-the-shelf” x86 PCs
    - The development environment is familiar to Linux C/C++ devs (i.e. without the traditional “embedded software” constraints)
- Safety:
    - ISO 26262 for the low-level safety layer
    - Our approach is:
        - The low-level safety layer (C) prevents any collision (through an emergency stop)
        - The higher-level (C++) software does the actual navigation
- Cybersecurity concerns:
    - SAE J3061
- Availability:
    - The target is: 24/7

Architecture



Why C++...

... rather than C or Ada?

- Performant
- Modern C++ introduces ways of preventing some classes of programming mistakes
- Automatic memory management, without garbage collection
- Technical ecosystem (standard library, ROS, Eigen)
- Developer/recruitment pool

Feedback: C++ Navigation software - 2018

TIS for C++ specificites:

- The code is first translated to C; implicit constructs are explicited, such as:
  - constructors/automatic destruction;
  - virtual methods;
  - copy elision; ...
- The STL is huge
- Name mangling, templates

Results

- Eight days over a one month period
- 3 developers learning TIS Analyzer
- 9 unit tests have been analyzed, some of which have been generalized
- Bugs found:
    - 2 in our code (one of them repeated several times in a lib)
    - 3 in our dependencies (boost, eigen, gtsam)
    - 1 in LLVM's C++ standard library (`https://bugs.llvm.org/show_bug.cgi?id=39354`)
- Use of uninitialized memory
- Use of dangling references
- Use of invalid iterator
- Year 2262 bug

TRUST IN SOFT

easy MILE

Results, example (1)

```cpp
#include <cstdint>

template <size_t N>
void f(std::uint8_t (&bytes)[N], size_t offset)
{
  for (auto i = 0; i < offset; ++i)
  {
    const auto mask = std::uint8_t{1} << (i % 8);
    auto& byte = bytes[i / 8];

    byte |= mask; // Use of uninitialized memory here
  }
}

int main() {
  std::uint8_t x[4]; // Uninitialized memory allocated here
  f(x, 3);
}
```

```c
/*@ assert Value: initialisation: \initialized(byte);*/
*byte = (unsigned char)((int)*byte | mask);
```

Results, example (2)

```cpp
auto first = edges.begin();
auto last = std::prev(edges.end());
// Invalid iterator when "edges" is empty
// The above line is UB even if we never enter
// the loop
for (auto it = first; it != edges.end(); ++it)
{
  f(*it);
}
```

TRUST IN SOFT

easy MILE

Difficulties related to our use-case

- Floating point computation
    - NaN and infinity are treated as errors
    - whereas we regard them as special values/limit cases
    - work in progress to change this behaviour
- Lots of trial and error to get the full sources needed by the analysis
    - one of our analysis involved 35 source files
    - tooling work ongoing to automate this
- System calls are specified (using ACSL)
    - but they sometimes need to be stubbed to return arbitrary values
    - in that case, all stubbed system calls involved need to be consistent
    - and doing so reduces the scope for which the program is proven UB-free

System calls difficulty example (1)

Year 2262 bug using `clock_gettime`:

```
struct timespec {
   long tv_sec;
   long tv_nsec;
};

/// Overflows April 11th, 2262 at 23:47:16
long nanoseconds = tv_sec * 1'000'000'000;
```

Initial workaround attempt:

```
int clock_gettime(clockid_t clk_id, struct timespec *tp) {
  constexpr auto year_2262_bug = /* y2262_bug_time - epoch as nanoseconds */
  tp->tv_sec = tis_long_interval(0, year_2262_bug - 1);
  tp->tv_nsec = tis_long_interval(0, one_sec_in_ms - 1);
  return 0;
}
```

TRUST IN SOFT

System calls difficulty example (2)

The code involved in the analysis then uses `gmtime_r` to split a unix timestamp into a date description:

```
/*@ requires \valid(__result);
  assigns *__result \from *__timer;
  assigns \result \from __result;
  ensures \result == __result;
  ensures \initialized(__result);
*/
struct tm *gmtime_r(const time_t *__timer,
                    struct tm *__restrict __result);
```

TRUST IN SOFT

System calls difficulty example (3)

However, the ACSL specification leads to overapproximation (e.g. hours greater than 23).

First take at stubbing `gmtime_r`:

```
result.tm_sec = tis_interval(0, 60);
result.tm_min = tis_interval(0, 59);
result.tm_hour = tis_interval(0, 23);
result.tm_mday = tis_interval(1, 31);
result.tm_mon = tis_interval(0, 11);
result.tm_year = tis_interval(0, 2038 - 1900);
result.tm_wday = tis_interval(0, 6);
result.tm_yday = tis_interval(0, 365);
tis_make_unknown((char *)&result.tm_isdst, sizeof(result.tm_isdst));
```

That will not do: this contains non-sense dates, such as February 31st.

TRUST IN SOFT

easy MILE

System calls difficulty example (3)

Final stub: let's us return an arbitrary date, 2018-10-18 (a Thursday, during daylight savings time), 14:18:11.

```
result.tm_sec = 11;
result.tm_min = 19;
result.tm_hour = 14;
result.tm_mday = 18;
result.tm_mon = 10-1;
result.tm_year = 2018-1;
result.tm_wday = 4-1;
result.tm_yday = 291-1;
result.tm_isdst = 1;
```

But `clock_gettime` must now be consistent with `gmtime_r` and return the same date:

```
tp->tv_sec = 1539872351;
tp->tv_nsec = 0;
```

Conclusions

- Putting static analysis tools and best practices in place prevented many bugs or helped us detecting them early
- When TIS Analyzer finds a bug, it is quite easy to understand
    - On the other hand, the tuning needed to refine false positives requires some skill and some time
- Generally speaking, the more maintainable the code is, the easier it is to analyze, encouraging us to:
    - Move side effects/syscalls outside of the library code
    - Analyze library code and user code (side-effects) separately
    - Simplify the implementation of some algorithms or code constructs
    - But this requires the ability to modify that code
- TIS Analyzer ensures us the code is free of any undefined behaviour
- Future standards for autonomous vehicles are not known but we're getting ahead

Questions ?