Exploits generation on C code using formal methods

Guillaume Cluzel
(guillaume.cluzel@trust-in-soft.com)

TrustInSoft

27th June 2023

TrustInSoft

Company founded in 2013 by former researchers of the CEA.

Fabrice Derepas

Benjamin Monate

Pascal Cuoq

Company's goal: commercialize a tool that provides mathematical guarantees on C and C++ programs: *TrustInSoft Analyzer*.

Company activity: 80 % tool licenses, 20 % service.

The need for secure communications

Multiple parameters are involved to ensure privacy:

The need for secure communications

Multiple parameters are involved to ensure privacy:

- the cryptographic algorithm

The need for secure communications

Multiple parameters are involved to ensure privacy:

- the cryptographic algorithm
- the communication protocol

The need for secure communications

Multiple parameters are involved to ensure privacy:

- the cryptographic algorithm
- the communication protocol
- the protocol/algorithm implementations

The need for secure communications

Multiple parameters are involved to ensure privacy:

- the cryptographic algorithm
- the communication protocol
- the protocol/algorithm implementations
- ...

The need for secure communications

Multiple parameters are involved to ensure privacy:

- the cryptographic algorithm
- the communication protocol
- **the protocol/algorithm implementations**
- ...

TrustInSoft Analyzer's goal

Guarantee the absence of *bugs* in the implementations of these protocols and algorithms.

A first exemple of bug: Heartbleed

- Heartbleed is a **security vulnerability** in OpenSSL.
- It affects the **TLS heartbeat** extension.
- Introduced in 2012 and reported in 2014. The vulnerability **existed for more than 2 years** in the OpenSSL implementation.

  $\rightarrow$ *CVE-2014-0160*

- The faulty line:

```
memcpy(bp, pl, payload);
```

`pl` is a pointer not necessary valid on `payload` bytes. It can result in buffer overflow and an attacker can read more data than it should.

Undefined behaviors

The heartbleed vulnerability exploits an **Undefined Behavior**.

"Undefined Behaviors" is the name for list of behaviors that are **proscribed** by the C norm and whose result is **unpredictable**.

Example

- "The value of a pointer to an object whose lifetime has ended is used".
- "If the quotient a/b is not representable, the behavior of both a/b and $a\%b$"

Detect undefined behaviors

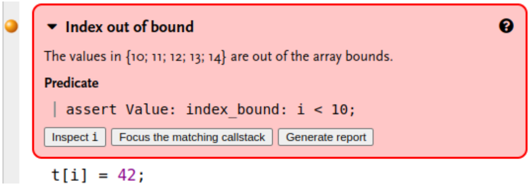
TrustInSoft Analyzer uses a plugin called **Value** to capture **all** the undefined behaviors.

Based on **abstract interpretation** to execute a program with additional checks.

```c
int t[10];

void setter(int i) {
  t[i] = 42;
}

int main() {
  setter(5);
  setter(tis_interval(8, 14));
}
```



Abstract interpretation I

Abstract interpretation is a technique to execute programs with **abstract values**.

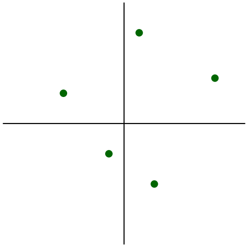


Figure – Testing a two variables program with unit tests.

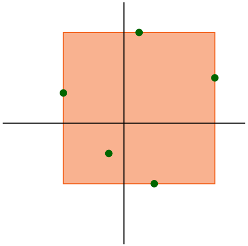


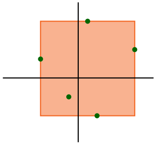
Figure – Analyzing a program with abstract interpretation. The values of the variables are all the values in the square.

⇒ Pascal Cuoq and Raphaël Rieu-Helft. "Result graphs for an abstract interpretation-based static analyzer". In: *28èmes Journées Francophones des Langages Applicatifs*. 2017.

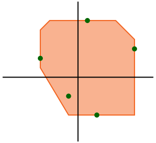
Abstract interpretation II

The idea behind abstract interpretation is to have a **correspondance** between a concrete lattice $(\mathcal{D}, \subseteq)$ and an abstract lattice $(\mathcal{D}^\sharp, \subseteq^\sharp)$ that over-approximates the concrete lattice and has "good properties".

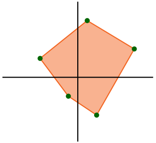
- There are **non-relational abstract domains** that only keep an abstract value for each variable.
  - *Example:* lattice of signs, intervals, ...
- **relational abstract domains** that keep relations between the program variables.
  - *Example:* Octogons, polyhedra, ...



Intervals



Octogons



Polyhedra

TrustInSoft choices

TrustInSoft analyzer uses intervals to represent the values of the variables because:

- It makes the analysis **faster** than with relational domains on **huge code bases** (+100K LOC).
- **The results are easily understandable for a C programmer.**
- Configuring an analysis is **rather easy** to have almost **no false** positive and **eliminate all the UBs**.

Successes

- mbed TLS is immune to certain types of Common Weaknesses Enumerations. It ensures the absence of Heartbleed-like errors if deployed accordingly to the Secure Deployment Guide.
- Used by highly-renowned industrial actors like Thales, Safran, Mitsubishi, Sony Interactive Entertainment.

Is it enough?

Is it enough?

NO...

Is it enough?

NO...

What if the code does not contain any UB but still it does not do what it is supposed to do?

Is it enough?

NO...

What if the code does not contain any UB but still it does not do what it is supposed to do?

It can still be considered as a bug, even if it does not result in a runtime failure.

Functional specifications

To overcome this problem, it is possible to add functional specifications to C functions. These annotations can be:

- Logic specifications: predicates, logic functions, etc.
- Function contracts as pre- and postconditions.
- Assertions in the source code.
- Loop invariants.

Specification language

The language used to write these specifications is **ACSL** (**A**NSI/ISO **C S**pecification **L**anguage).

An example of function specification: abs

The abs() function computes the absolute value of the integer argument v.

```
int abs(int v) {
  int tmp;
  if (v >= 0) tmp = v;
  else tmp = -v;
  return tmp;
}
```

An example of function specification: abs

The abs() function computes the absolute value of the integer argument v.

```
/*@
ensures
    \result == ((v >= 0) ? v : -v);
*/
int abs(int v) {
  int tmp;
  if (v >= 0) tmp = v;
  else tmp = -v;
  return tmp;
}
```

The **ensures** keyword expresses the post-condition of the function.

An example of function specification: abs

The abs() function computes the absolute value of the integer argument v.

```
/*@
requires v != INT_MIN;
ensures
    \result == ((v >= 0) ? v : -v);
*/
int abs(int v) {
  int tmp;
  if (v >= 0) tmp = v;
  else tmp = -v;
  return tmp;
}
```

The **requires** keyword specifies the acceptable values at the entry of the function, *i.e.* a property that must be true when the function is called.

The **ensures** keyword expresses the post-condition of the function.

An example of function specification: abs

The abs() function computes the absolute value of the integer argument v.

```
/*@ requires v != INT_MIN;
    behavior positive:
      assumes v >= 0;
      ensures \result == v;
    behavior negative:
      assumes v < 0;
      ensures \result == -v; */
int abs(int v) {
  int tmp;
  if (v >= 0) tmp = v;
  else tmp = -v;
  return tmp;
}
```

The **behaviors** can be used to improve the readability of a contract when the function changes its behavior depending of its input.

Weakest precondition calculus

Definition (Hoare Triple)

A **Hoare Triple** is a triple denoted $\{P\}s\{Q\}$ where P and Q are logic propositions and s is a statement. P is called the *precondition* and Q the *postcondition*.

The Value plugin is able to prove functional specifications. But only the most simple ones.

The notion of weakest precondition (WP) calculus was originally proposed by Dijkstra.
It is able to reduce the problem of proving that a triple $\{P\}s\{Q\}$ that is derivable using the classical Hoare logical rules to the proof of a mathematical formula.

Functional proofs in the industry

Tools implementing the WP calculus have already been successfully used:

- Methode B (Atelier B)
- SPARK (*Ada*)
- Frama-C and its WP plugin (*C*)
- Why3 (*WhyML and used as a library for other languages*)
- Dafny
- KeY (Java)
- ....

... but mainly by formal methods experts.

Functional proofs in the industry

Tools implementing the WP calculus have already been successfully used:

- Methode B (Atelier B)
- SPARK (*Ada*)
- Frama-C and its WP plugin (*C*)
- Why3 (*WhyML and used as a library for other languages*)
- Dafny
- KeY (Java)
- ....

... but mainly by formal methods experts.

The goal is to **overcome the limitations encountered by WP** and **make program proofs accessible** to as many C developers as possible.

The J^3 plugin

Thanks to the past experiences, we collaborate on a new plugin called J^3 with the **Inria TOCCATA Team**.

We closely work with **Claude Marché**.

- This plugin is **based on Why3**.
- It should handle **low-level code**.
- Give the most **precise feedbacks**.

  Especially, we want to **generate counterexamples** when a goal is not proven, and **explain the counterexamples**.

How J^3 works

1 **Parsing**

The C code is parsed by the Kernel to produce an intermediate AST.

How J^3 works

1 Parsing

The C code is parsed by the Kernel to produce an intermediate AST.

2 Translation

J^3 transforms the C AST and the program annotations to a Why3 AST.

How J³ works

1 Parsing

The C code is parsed by the Kernel to produce an intermediate AST.

2 Translation

J³ transforms the C AST and the program annotations to a Why3 AST.

3 WP calculus

Why3 generates the verification conditions for the generated AST.

How J³ works

1 Parsing

The C code is parsed by the Kernel to produce an intermediate AST.

2 Translation

J³ transforms the C AST and the program annotations to a Why3 AST.

3 WP calculus

Why3 generates the verification conditions for the generated AST.

4 SMT solvers

SMT solvers try to answer whether the VC is valid or not.

How J³ works

1 Parsing

The C code is parsed by the Kernel to produce an intermediate AST.

2 Translation

J³ transforms the C AST and the program annotations to a Why3 AST.

3 WP calculus

Why3 generates the verification conditions for the generated AST.

4 SMT solvers

SMT solvers try to answer whether the VC is valid or not.

5 Prover results

The results of the provers are parsed by Why3 to collect as much information as possible.

How J^3 works

1 Parsing

The C code is parsed by the Kernel to produce an intermediate AST.

2 Translation

J^3 transforms the C AST and the program annotations to a Why3 AST.

3 WP calculus

Why3 generates the verification conditions for the generated AST.

4 SMT solvers

SMT solvers try to answer whether the VC is valid or not.

5 Prover results

The results of the provers are parsed by Why3 to collect as much information as possible.

6 Result

J^3 outputs understandable results for a C developer.

Retrieve information from the provers

An assertion in the code can be either proven or not.

For a non proven goal:

- The provers can suggest a **model**.
- The model can be seen as a **counterexample for the property** we are trying to prove in the C code.

$\Rightarrow$ *The work presented in the next slides is still a work in progress.*

The need for counterexamples

There are many reasons why the specification cannot be proven:

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers

```
/*@ requires \abs(x) <= 1.0;
    ensures
      \abs(\result - \exp(x)) <= 0x1p-4; */
float exp() {
    return
      0x1.fa62ffd643d6ep-1 +
      0x1.2158a22d91de9p+0 * x +
      0x1.1babaa64d94dbp-1 * x * x;
}
```

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers

```
/*@ requires \abs(x) <= 1.0;
    ensures
      \abs(\result - \exp(x)) <= 0x1p-4; */
float exp() {
    return
      0x1.fa62ffd643d6ep-1 +
      0x1.2158a22d91de9p+0 * x +
      0x1.1babaa64d94dbp-1 * x * x;
}
```

Counterexamples cannot help as the specification is correct.

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers
- The specification can be **too weak** to prove the goals

```
/*@ ensures \result >= 0; */
int two() { return 2; }

void foo() {
    int x = two();
    /*@ assert x == 2; */
}
```

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers
- The specification can be **too weak** to prove the goals

```
/*@ ensures \result >= 0; */ // result = 0;
int two() { return 2; }

void foo() {
    int x = two();          // x = 0;
    /*@ assert x == 2; */ // x = 0;
}
```

The counterexample can be exploited to show that the specification of `foo` is too weak.

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers
- The specification can be **too weak** to prove the goals
- The **specification can be wrong**
- The **code can be wrong**

```
/*@ requires -20 <= x <= 40;
    ensures \result >= 0 */
int foo(int x) {
    return x;
}
```

The need for counterexamples

There are many reasons why the specification cannot be proven:

- The specification can be **too hard to prove** for the SMT solvers
- The specification can be **too weak** to prove the goals
- The **specification can be wrong**
- The **code can be wrong**

```
/*@ requires -20 <= x <= 40; // x = -3;
    ensures \result >= 0 */  // result = -3;
int foo(int x) {     // x = -3;
    return x;        // x = -3;
}
```

The counterexample can be seen as an ***exploit****. Indeed, the counterexample shows how to exploit a defect of the code. Security teams can* ***exploit them to evaluate the severity of a bug****.*

Checking the counterexamples

The main problem with counterexamples is that **they can be wrong**... And showing to a user a wrong counterexamples is more confusing than not showing anything. It is necessary to **discard the wrong counterexamples**.

One solution consists in checking the counterexample with **"giant-steps execution"**

Giant-steps-execution

At each program point, the intermediate assertions and the function contracts of the functions that are called are checked with the values given by the counterexample.

The giant-steps execution

We want to check if the counterexample is correct for the assertion

```
/*@ ensures \result >= 0; */ // result = 0;
int two() { return 2; }

void foo() {
    int x = two();         // x = 0;
    /*@ assert x == 2; */ // x = 0;
}
```

The giant-steps execution

We want to check if the counterexample is correct for the assertion

```
/*@ ensures \result >= 0; */ // result = 0;
int two() { return 2; }

void foo() {
    int x = two();          // x = 0;
    /*@ assert x == 2; */ // x = 0;
}
```

→ The assertion evaluates to True (for the first line)

→ The assertion evaluates to True (for the `int x = two();` line)

The giant-steps execution

We want to check if the counterexample is correct for the assertion

```
/*@ ensures \result >= 0; */ // result = 0;  → The assertion evaluates to True
int two() { return 2; }

void foo() {
    int x = two();          // x = 0;         → The assertion evaluates to True
    /*@ assert x == 2; */ // x = 0;           → The assertion evaluates to False
}
```

The counterexample is **good** as it does not respect the final assertion but it respects all the intermediate assertions.

Classification of the counterexamples

In addition, it is possible to get even more feedback from the counterexample. We can get a **classification of the counterexample** by completing the giant-step execution with a **concrete execution**.

- **BadCE** One of the two did not go well. The counterexample is bad.
- **GoodCE** The two executions went well.
- **Subcontract weakness** A subcontract of a function or a loop invariant is too weak to prove a property.

⇒ Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. “Explaining Counterexamples with Giant-Step Assertion Checking”. In: *6th Workshop on Formal Integrated Development Environments (F-IDE 2021)*. Ed. by José Creissac Campos and Andrei Paskevich. Electronic Proceedings in Theoretical Computer Science. May 2021.

Concrete execution

We want to check if the counterexample is correct for the assertion with a concrete execution

```
/*@ ensures \result >= 0; */ // result = 2;
int two() { return 2; }

void foo() {
    int x = two();  // x = 2;
    /*@ assert x == 2; */  // x = 2;
}
```

The assertion can be proved with the values computed by the concrete execution.

Concrete execution

We want to check if the counterexample is correct for the assertion with a concrete execution

```
/*@ ensures \result >= 0; */ // result = 2;
int two() { return 2; }

void foo() {
    int x = two();  // x = 2;
    /*@ assert x == 2; */  // x = 2;
}
```

The assertion can be proved with the values computed by the concrete execution.

The automatic system classifies it as a **subcontract weakness**.

Mixing the proof techniques

Some future research focus.

1. Large analysis campaigns can be run with Value (abstract interpretation plugin).
2. When an undefined behavior is detected:
   - It can be a real bug.
     We want to be able to generate an exploit thanks to the J^3 plugin
   - It can be a false positive due to imprecision of Value.
     We want to be able to remove the alarm.

Mixing the proof techniques

Some future research focus.

1. Large analysis campaigns can be run with Value (abstract interpretation plugin).
2. When an undefined behavior is detected:
   - It can be a real bug.
     We want to be able to generate an exploit thanks to the J^3 plugin
   - It can be a false positive due to imprecision of Value.
     We want to be able to remove the alarm.

It requires to find a way to **automatically generate** pre-, postconditions and loop invariants.

Conclusion

Using counterexamples is a promising way to provide feedback to the user by:

- Generating exploits and counterexamples.
- Providing additional information to help the user to prove their code.

This technique deserves to be more widely used, in particular in addition to other verification techniques.