CLANG STATIC ANALYSIS TOOLSET

Industrial Experiences & The CodeChecker Solution

Daniel Krupp (daniel.krupp@ericsson.com)

WHY STATIC ANALYSIS & WHY CLANG?

- Defect detection with Static Analysis is a cheap extension to testing
- Can catch bugs with no test coverage

- **Impressive** checker framework
- Working with Clang Static Analyzer since 2013
- Large potential user base at Ericsson ~5000 developers

We had difficulties to make it work in practice

CHECKING TECHNIQUES

1. Text Pattern Matching – (CppCheck...)
2. AST Matchers - (CppCheck, Clang AST matchers...)
3. Symbolic Execution – (Coverity SA, Clang Static Analyzer...)

How can we measure the precision of the checkers?

1. **False positive rate:** False Reports / All Reports
2. **False negative rate:** Non-reported defects / All existing defects

The lower these values, the better.

1. TEXT PATTERN MATCHING

```
#include <stdlib.h>
#define ZERO 0
int getNull(int a) {
  return a?0:1;
}
int getInput() __attribute__((notzero));
void test(int b)
{
  int a,c;
  double *d;
  switch (b){
    case 1:  a = b / 0; break;
    case 2:  a = b / ZERO; break;
    case 3:  d = (double*)
malloc(sizeof(d)); free(d); break;
    case 4:  c = b-4;
             a = b / c; break;
    case 5: a = b / getNull(b); break;
    case 6: a = b / getInput(); break;
    };
}
```

Found

Found if simple preprocessor statements are resolved.

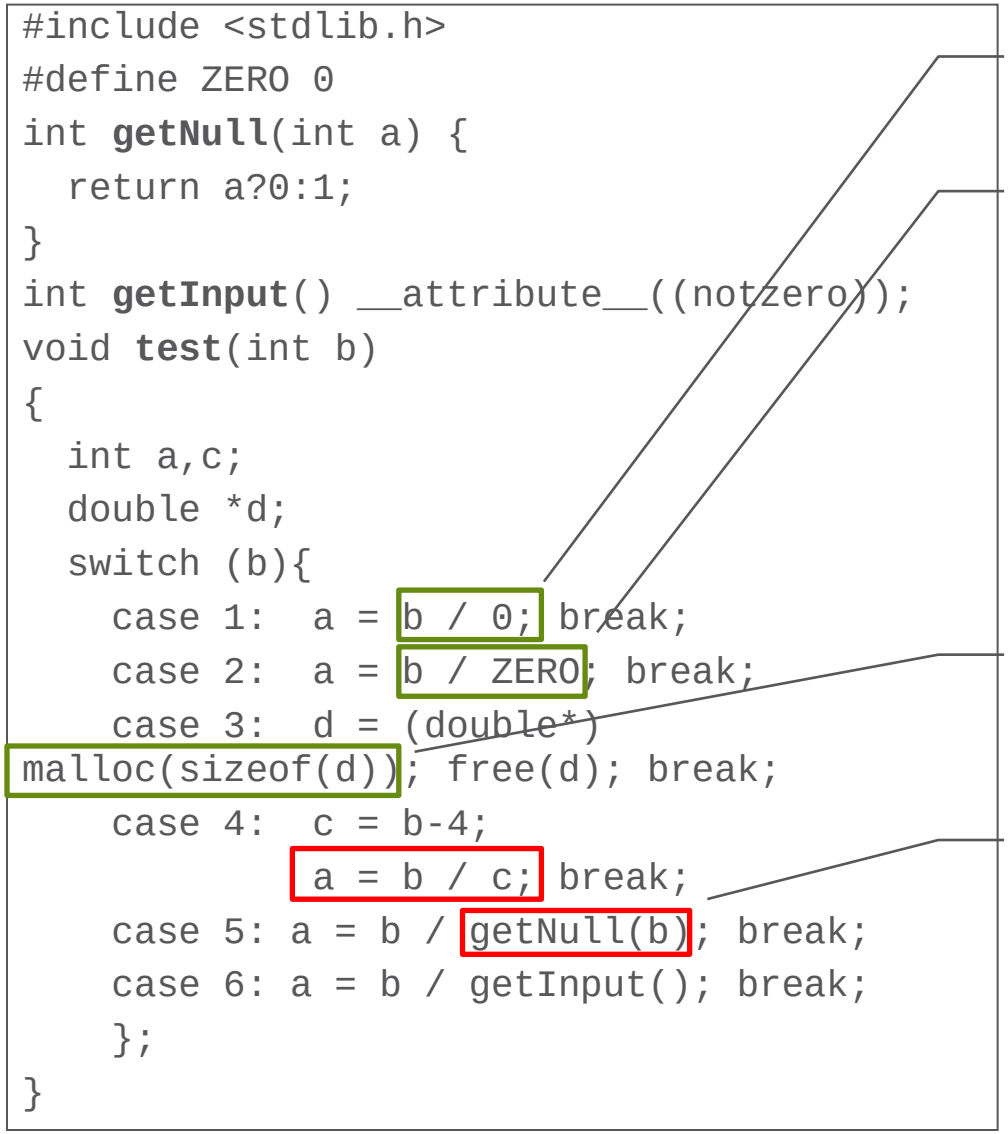
Token:Match(tok,"/ 0");

Not found as type resolution cannot be used.

Not found as symbolic expressions are not evaluated.

Flow insensitive

2. AST MATCHERS



```
#include <stdlib.h>
#define ZERO 0
int getNull(int a) {
  return a?0:1;
}
int getInput() __attribute__((notzero));
void test(int b)
{
  int a,c;
  double *d;
  switch (b){
    case 1:  a = b / 0; break;
    case 2:  a = b / ZERO; break;
    case 3:  d = (double*)
malloc(sizeof(d)); free(d); break;
    case 4:  c = b-4;
             a = b / c; break;
    case 5: a = b / getNull(b); break;
    case 6: a = b / getInput(); break;
    };
}
```

Found

Found as all preprocessor statements are resolved.

```
BUILD_MATCHER() {        return
binaryOperator(hasOperatorName("/"),
hasRHS(integerLiteral(equals(0)).bind(
KEY_NODE)));
    }
```

Found as type resolution can be used. (**size_of checker**)

Not found as symbolic expressions are not evaluated.

Flow insensitive

3. SYMBOLIC EXECUTION I

```
#include <stdlib.h>
#define ZERO 0
int getNull(int a) {
  return a?0:1;
}
int getInput() __attribute__((notzero));
void test(int b)
{
  int a,c;
  double *d;
  switch (b){
    case 1:  a = b / 0; break;
    case 2:  a = b / ZERO; break;
    case 3:  d = (double*)
malloc(sizeof(d)); free(d); break;
    case 4:  c = b-4;
             a = b / c; break;
    case 5: a = b / getNull(b); break;
    case 6: a = b / getInput(); break;
    };
}
```

Path Sensitive

Context Sensitive

As value of c evaluated and stored along the execution path.

Internal function calls are followed (context passed), variable constraints are stored, possible paths are executed.

Without context sensitivity, this is undecidable.

3. SYMBOLIC EXECUTION II

```
#include <stdlib.h>
#define ZERO 0
int getNull(int a) {
  return a?0:1;
}
int getInput() __attribute__((notzero));
void test(int b)
{
  int a,c;
  double *d;
  switch (b){
    case 1:  a = b / 0; break;
    case 2:  a = b / ZERO; break;
    case 3:  d = (double*)
malloc(sizeof(d)); free(d); break;
    case 4:  c = b-4;
             a = b / c; break;
    case 5: a = b / getNull(b); break;
    case 6: a = b / getInput(); break;
    };
}
```

```
//model hint

int getInput(){
      int unkown();
      int x = unkown();
      return x==0? 1:x;
}
```

configuration variable:

divisionByZero.optimistic

Shall this be reported?

If getInput cannot return 0, this is a false positive.

CHECKERS WE IMPLEMENTED

- **AST Matchers** (33, 5 contributed to Clang already)
  - Rule of three
  - Suspicious size of
  - Static assert

  ...
- **Preprocessor Matchers** (1)
  - Missing header guard
- **Symbolic Execution**(7)
  - Uninitialized class member
  - Return address of local variable

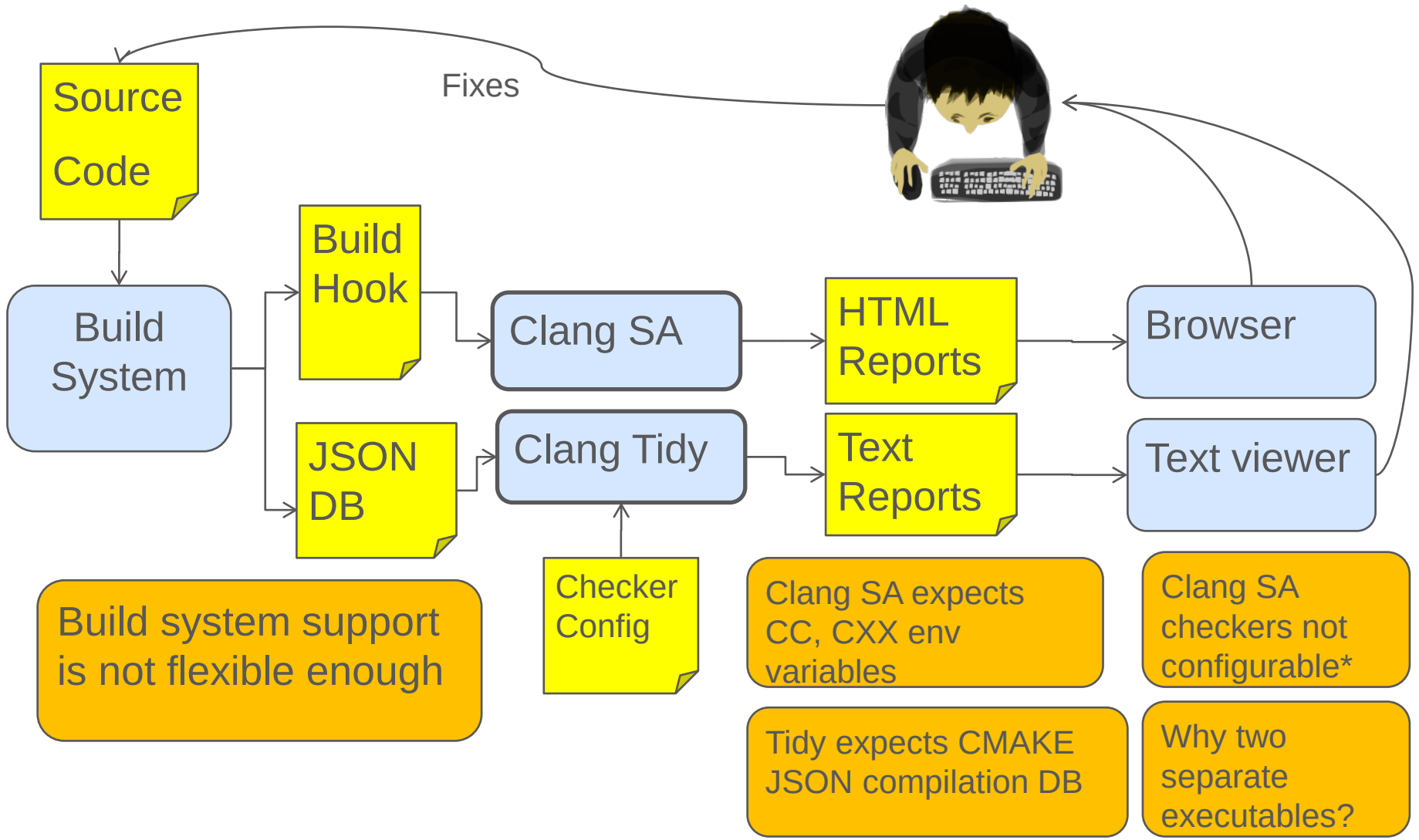
  …

CURRENT INFRASTRUCTURE

1. Clang Diagnostic
   - Fast, flow sensitive analysis
2. Clang Static Analyzer
   - Symbolic Execution (path, context sensitive)
3. Clang Tidy
   - AST Matchers (flow insensitive)
   - Preprocessor Matchers
   - Can call Clang Static Analyzer checkers
   - Can call Clang Diagnostic checkers

LET'S TAKE A BIRDS-EYE PERSPECTIVE!

CURRENT TOOLSET



+BUILD LOGGING

Fixes

Source Code

Build Log DB

Build System

Build Logger

Clang SA

Clang Tidy

Checker Config

HTML Reports

Text Reports

Browser

Text viewer

LD_PRELOAD loads

logger.so

Catches exec* call family

Universally works with all build systems

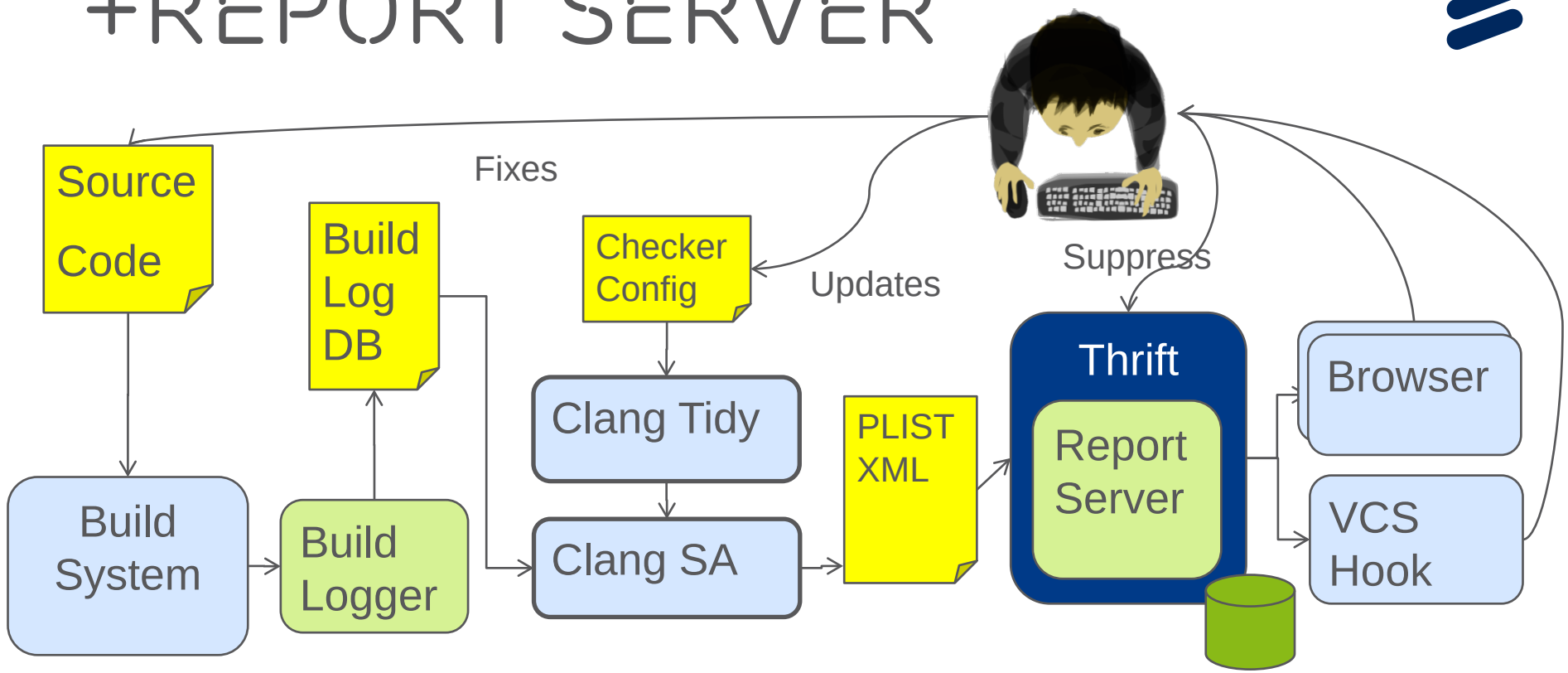
Tidy: Text warnings
Clang SA: HTML,PLIST

HTML reports are not scalable

HTML/Text reports are not manageable

Different users need different viewers!

+REPORT SERVER



Report Server

- Stores the reports in SQL DB
- Open web-service interface [Thrift generated]
- Many viewers can connect

Filterable, orderable report listing

False positive suppression

Severity

Diff view: show new/resolved bugs only

Many viewers: Eclipse, web, scripts

+DOCUMENTATION

Source Code

Fixes

Build Log DB

Checker Config

Updates

Suppress

Build System

Build Logger

Clang Tidy

Clang SA

PLIST XML

Thrift

Report Server

Browser

VCS Hook

Checkers Need Consistent Documentation

Doxygen template

Content

- Problem description
- Consequences, tips to fix
- Limitations, known false positives
- Configuration options
- Annotations
- Model hints

DEMO

VISION

Extend the Static Analysis toolset to cover the „Big Picture”

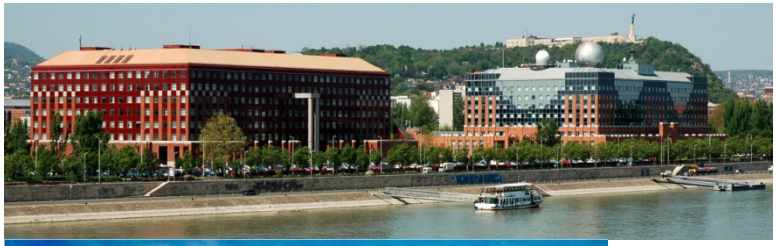
Instead of several expensive custom solutions

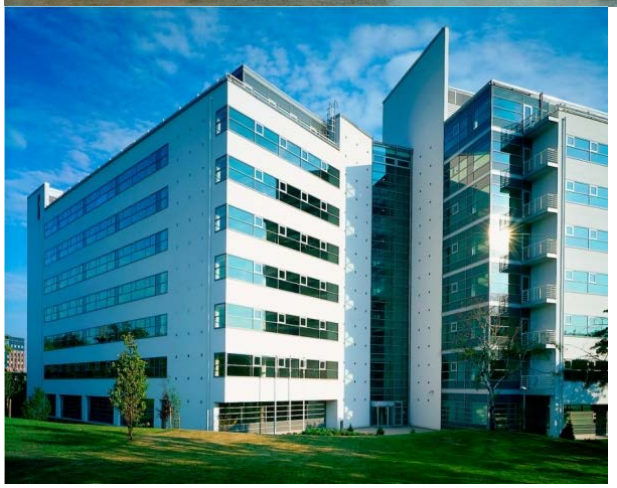
FUTURE WORK

- **Open the source** of the Viewers (standalone web, eclipse)
- **Open the source** of the Report Server
- **Open the source** of the Build Logger

- Introduce severity levels to checkers
- Implement PLIST support into Clang Tidy (under review)
- Clang-static analyzer checkers could use each others' results – introduce dependency management
- Introduce Confidence levels to checkers (todo in Tidy)
- Cross Translation Unit Checker Framework

WHO WE ARE





- Ericsson Software Lab @ Budapest, Hungary
  - 1 PhD
  - 2 Msc
- ELTE University, Budapest
  - 4 interns Bsc/MSc students
- Contributions to Clang
  - 5 accepted Checkers (30 more to come)
  - GSOC 2014 on Cross TU Analysis
  - Several patches to Tidy and Static Analyizer

QUESTIONS

CONTACTS

Daniel Krupp daniel.krupp@ericsson.com
Zoltán Porkoláb zoltan.porkolab@ericsson.com

Credits

Gábor Horváth gabor.a.horvath@ericsson.com
Bence Babati bence.babati@ericsson.com
György Orbán gyorgy.orban@ericsson.com
Szabolcs Sipos szabolcs.sipos@ericsson.com
Boldizsár Tóth boldizsar.toth@ericsson.com

REFERENCES

[1] Par Emanuelsson and Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools, Linköping University, Report number 2008:3

[2] CPPCheck

[3] Clang Tidy

[4] Clang Static Analyzer

Used Figures

› Dragon Boat

› Programmer

ERICSSON