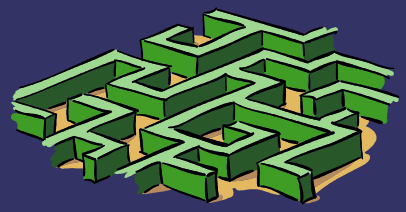
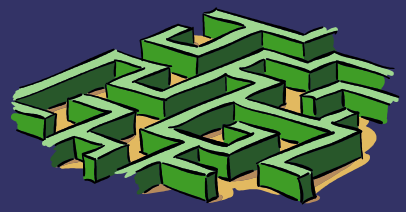
Importance of Early Bug Detection for Improving Program Reliability and Reducing Development Costs

Sergey Ignatchenko



Disclaimer

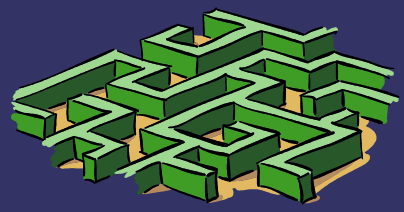
- All data and especially opinions within this presentation represent an inherently subjective point of view and should be taken internally only with a pinch of salt. Your mileage may vary.



Bugs, bugs, bugs...

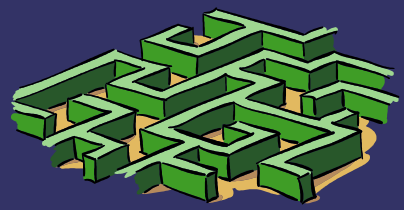
Imagine if every Thursday your shoes exploded if you tied them the usual way. This happens to us all the time with computers, and nobody thinks of complaining.

-- Jeff Raskin --

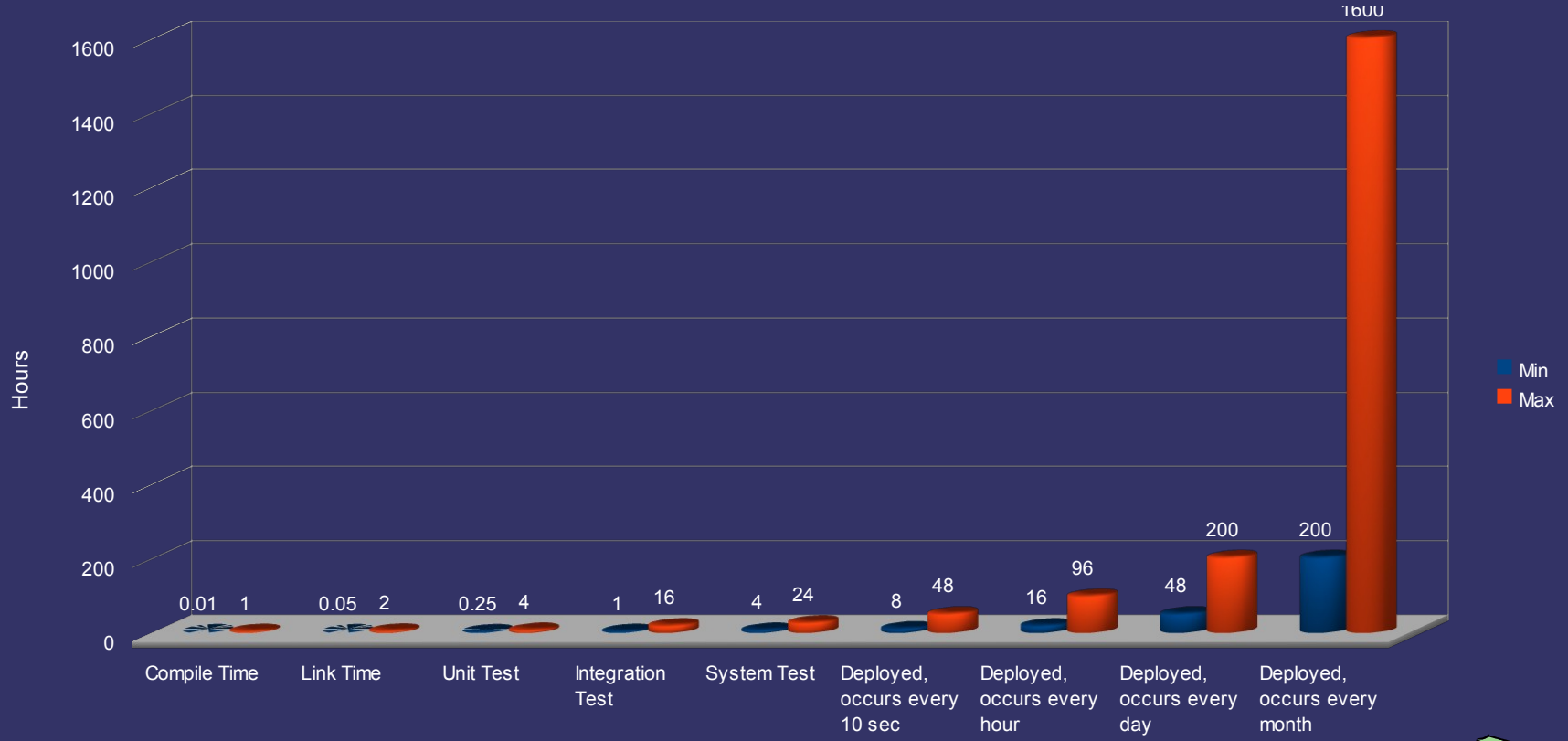


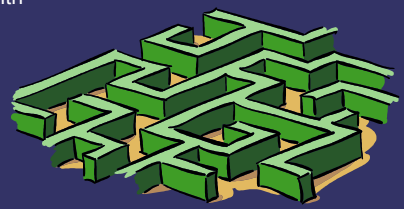
Bug Life Cycle

- Bug mysteriously emerges...
- Bug goes through compiler
- Bug makes it through linker
- Bug is not detected during unit-test
- Bug survives integration test
- Bug further survives system test
- Bug is deployed
- Bug is reported
- Bug is reproduced
- Bug is identified
- Bug is fixed



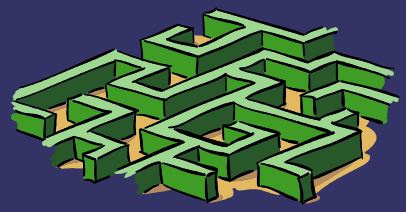
How Does Time Necessary to Identify Bug Depend on the Stage When It Is Detected





Especially Nasty Bugs

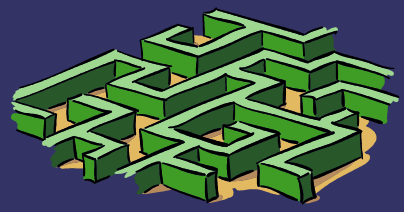
- Multithreading Bugs (easy to make, difficult to reproduce)
- Security Bugs (nobody cares to report)
- 3rd-party Bugs (nobody expects them there)



Nothing Too New...

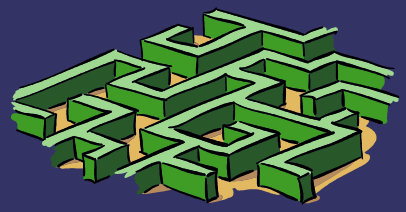
The more bug is allowed to live, the more damage it makes:

- Decreases program reliability
- Decreases customer satisfaction
- Increases development/maintenance costs



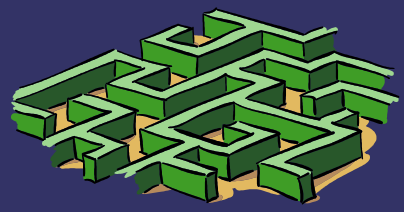
Existing Solutions

- Production-mode logging and reporting
- Asserts
- Self-restricting techniques like 'const'
- Static code analysis/LINT



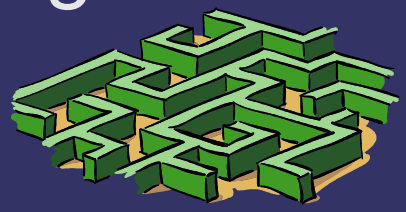
Existing Solution: Production Mode Logging and Reporting

- Trivial to implement (reporting can be tricky for client-side or standalone apps, but logging is still trivial)
- Can be implemented regardless of programming language used
- Comprehensive logging enables post-mortem analysis
- Exceptions e-mailed to developers can be a bit too much, but it helps to reduce number of bugs fairly quickly, and often allows to fix the bug even before somebody complains.



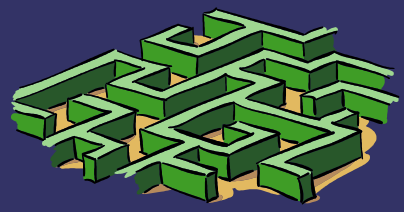
Existing Solutions: asserts

- Extremely helpful not only to detect that there is a bug, but also to identify it
- The most convenient in C/C++, but can be used in other languages too
- Cannot have too many asserts
- Contrary to popular belief, in C/C++ it is often useful to leave some asserts even in production mode
- Close cousins: checked builds, debug-mode libraries, and tools like Valgrind.



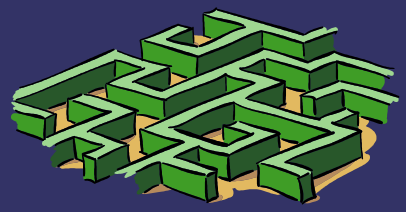
Existing Solutions: const, private, etc.

- Compile-time detection (as early as it gets)
- As they don't change generated binary in any way, it is essentially a tool which helps programmer to protect himself from his own mistakes
- Availability depends on language, but most modern languages support similar concepts one way or another



Existing Solutions: static code analysis/LINT

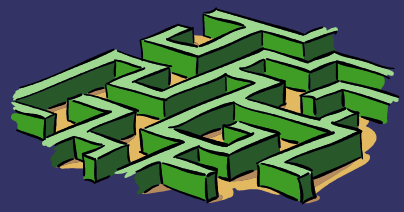
- LINT itself is a C/C++ tool, but there are similar tools for static code analysis for other languages
- Compile-time detection (as early as it gets)
- Detects “suspicious” constructs which definition is vague, and varies from project to project
- Usually tuning it for your project takes some time, but it is still worth it.



Static Code Analysis: what we would like to be detected?

Example 1:

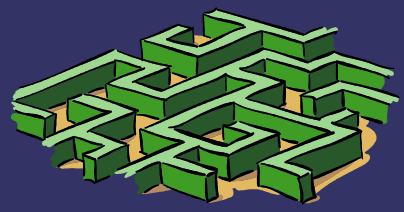
```
void f( const char* src ) {
  char buf[ 32 ];
  strncpy( buf, src, 64 );
}
```



Static Code Analysis: what we would like to be detected?

Example 2:

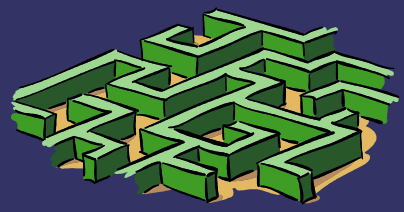
```
struct X {
  string a; string b; int c; string long_text;
  bool operator < ( const X& other ) {
    if ( a != other.a ) return a < other.a;
    if ( b != other.b ) return b < other.b;
    if ( c != other.c ) return c < other.c;
    return true; // we don't care what to return
                 // if all of (a,b,c) are the same
  }
};
set< X > set_of_x;
```



Static Code Analysis: what we would like to be detected?

Example 2:

```
struct X {
  string a; string b; int c; string long_text;
  bool operator < ( const X& other ) {
    if ( a != other.a ) return a < other.a;
    if ( b != other.b ) return b < other.b;
    if ( c != other.c ) return c < other.c;
    return true; // bug!!
   }
};
set< X > set_of_x;
```



Static Code Analysis: what we would like to be detected?

Example 3:

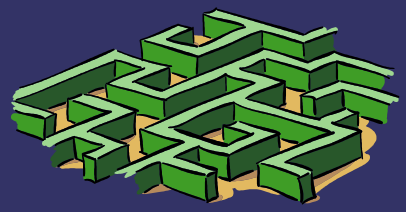
```
struct X {
  Mutex mx;
  string s; // all access should be protected
                // by mutex mx
  };

void f( X& x ) {
  // Lock lock ( mx ); // erroneously omitted
  string tmp = x.s; // problem: we're
                       // accessing s without lock
  }
```

Static Code Analysis: we would like to detect it, but can we?

Bad: neither compiler nor LINT can understand the comment below:

```
string s; // all access should be protected
          // by mutex mx
```

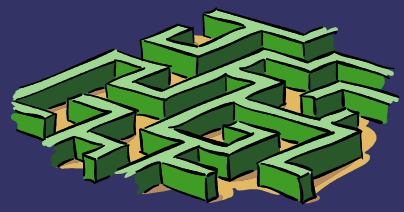
Good: we can make it. Sort of.



Static Code Analysis: C+-

C+-:

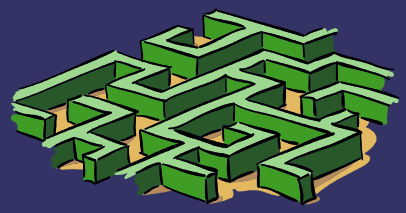
- Soon to be released as an open source
- Extensible language with DIY extensions
- Compiles into C/C++ code (in the future – also in Java)
- Some extensions can be “restrictive”
- One of extensions we already have is @protected_by extension.



Static Code Analysis: Example 3 revisited

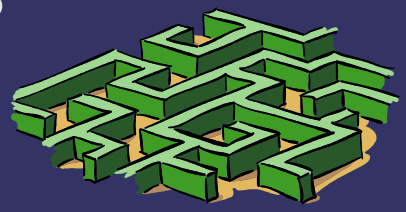
Example 3 in C+-:

```
struct X {
  Mutex mx;
  string s @protected_by( mx );
  };

void f( X& x ) {
  // Lock lock ( mx ); // erroneously omitted
  string tmp = x.s;// compiler generates error
                   // because of protected_by
                   // specification
  }
```



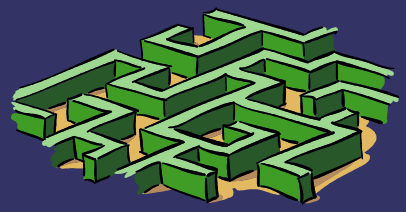
C+-: Other Extensions

- Already has an extension to enforce safe parameters for “printf”
- Also as an extension to prevent buffer overflows in functions like strncpy() (Example 1)
- Should be rather easy to add more extensions
- We hope that open source community will contribute many extensions (including those aimed to enforce certain safe practices).
- In any case, use of all extensions is optional.



Advanced Static Analysis: an Important Tool in Early Bug Detection

- In general, may need extending language to allow specifying certain concepts explicitly.
- Can help detect certain classes of bugs as early as it gets (compile-time).
- Has a big potential of improving of overall quality of code.



Computers can't do anything smart for you, but they can do a lot of stupid work instead of you, freeing you time to do something smart.