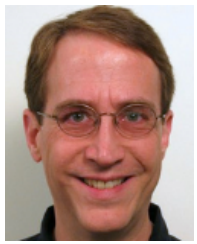
Generating Precise Dependencies for Large Software

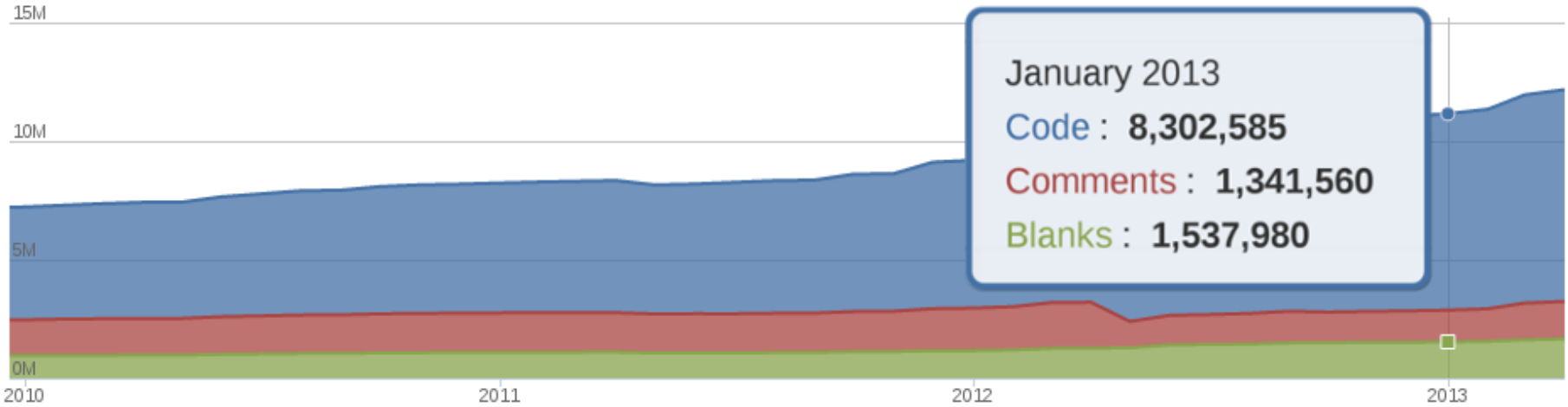


Pei Wang, Jinqiu Yang, Lin Tan
University of Waterloo

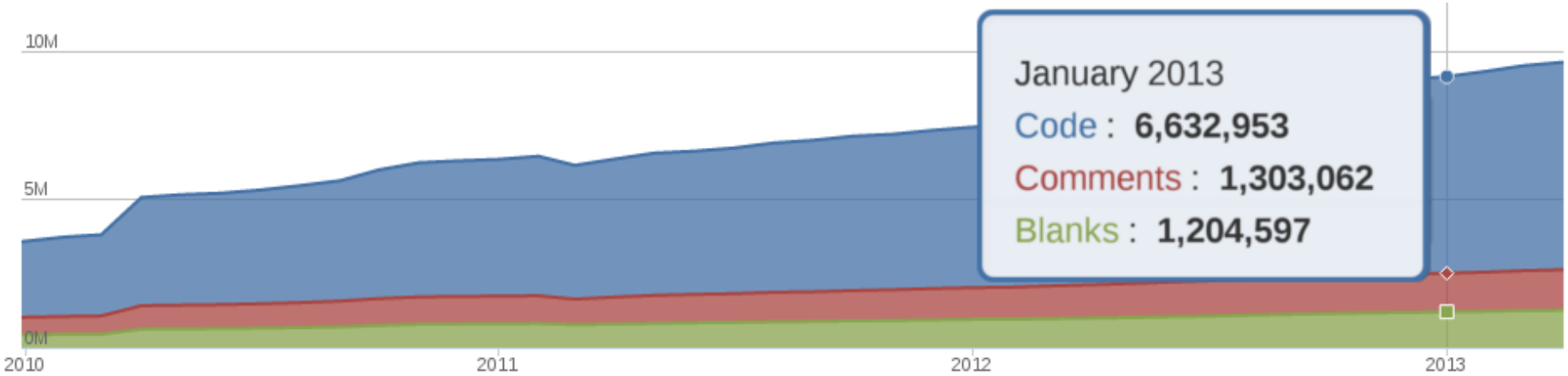



Robert Kroeger, David Morgenthaler
Google Inc.

Code Base Size is Growing



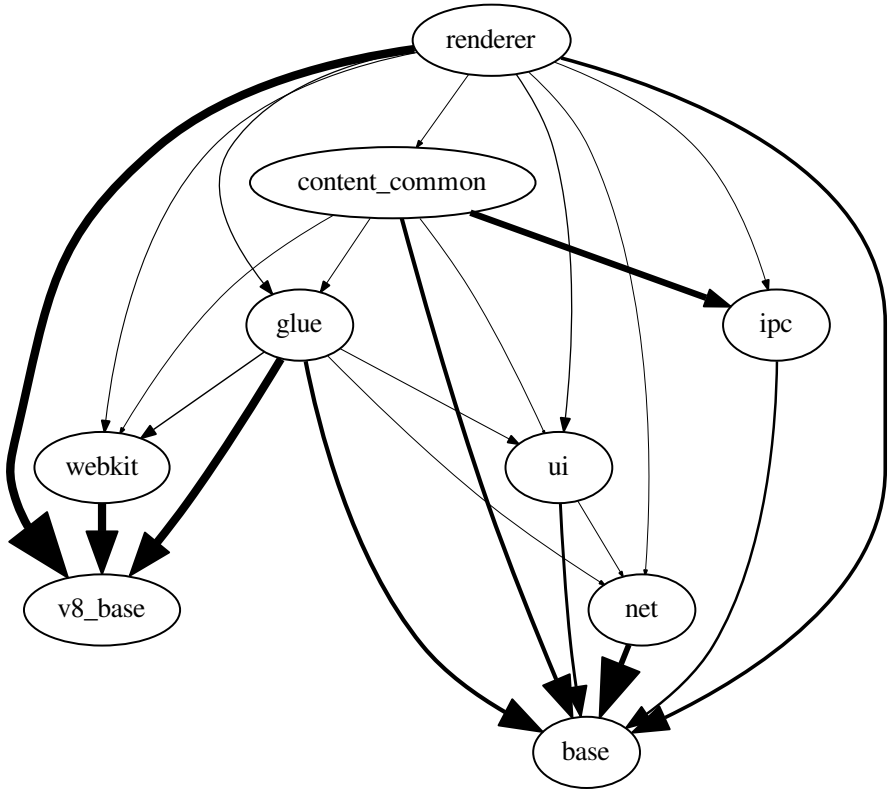
Mozilla Firefox Code Base Size (2010-2013)†



Chromium (Google Chrome) Code Base Size (2010-2013)†

† Data from Ohloh®

Software Complexity is Increasing



Dependencies between Some Key Components of Chromium

By December 2012, Chromium (svn-171054) has 238 modules.

Technical Debt Caused by Increasing Structural Complexity

Technical Debt in Software Development

Compromises made for short term benefits (meeting product release deadline, etc.) but hurting long term maintainability of the software

Two Kinds of Bad Dependencies

- *Inconsistent Dependency:* dependencies violating software design
- *Underutilized Dependency:* only a small portion of a target module is utilized by a client module

Bad Dependencies Tell Us About

- Modularity Violation
- Loosely Coupled Components & Useless Code
- Refactoring Cost

Light-Weight Dependency Analysis is Not Enough

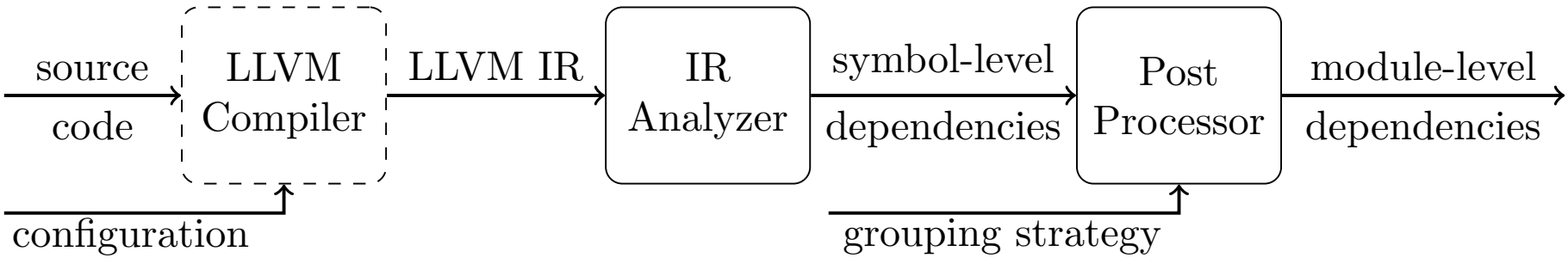
Light-Weight Analysis Techniques

- Pattern Matching
- Abstract Syntax Tree Based Analysis

Challenges in Large-Scale C++ Dependency Analysis

- Function/Operator overloading and default parameters
- Non-standard language syntax
- Implicit call sites
- Templates

Tool Design Overview



Workflow

1. Compile C/C++ source into LLVM Intermediate Representation (IR).
2. Extract symbol-level dependencies from LLVM IR instructions.
3. Group symbol-level dependencies to get module-level dependencies.

Step 2: Symbol-Level Dependency Extraction

- Obtain symbol references by traversing LLVM IR instruction.
- Resolve symbol linkage through a mock linking process.

Example: Non-Standard Syntax Support

```
chromium/src/content/zygote/zygote_main_linux.cc:182:
struct tm* localtime_override(const time_t* timep)
                              __asm__ ("localtime");
```

C++ Code

```
obj.target/content_browser/content/zygote/zygote_main_linux.o:
define %struct.tm* localtime(i64* %timep) nounwind uwtable
```

LLVM IR

Step 3: Module-Level Dependency Analysis

- Group symbols into modules:
  - The grouping strategy can simply be the build configuration of the software and allows user customization.

$$\text{Target-Module-Util} = \frac{\#\text{ of symbols in client's dependency}}{\#\text{ of symbols defined in the target}}$$

- Utilization-related metrics:
  - Pairwise Utilization
  - Overall Utilization

Performance Evaluation

Analysis Scale (Chromium svn-171054)

Lines of C/C++ Code	6 Million
# of Symbols	470,797
# of Symbol References	13,912,651
# of Modules	238

- Analysis time: $\sim$ 123 minutes (3.1GHz Core i5)
  - $\sim$ 88 minutes' compilation time
  - $\sim$ 35 minutes' analysis time
- Peak memory usage: 5.6GB

Preliminary Findings

Partial List of Underutilized Modules in Chromium

Module	# of Symbols	Overall Util†
notifier	181	4.4∼17.1%
ppapi_cpp_objects	1195	17.5∼17.6%
dbus	334	18.9∼18.9%
ppapi_ipc	3228	19.4∼19.4%
remoting_jingle_glue	97	12.4∼19.6%

† *The range shows the impact of virtual function calls.*

A Potential Inconsistent Dependency

The module ***base***, which is not supposed to depend on any other modules, is using a third-party Base64 en-decryption library.

Conclusion

- **Scalable** and **precise** structural dependency extraction and analysis
  - Scales to millions of lines of code
- Full C++ Support
  - Can analyze most salient C++ features
  - Support some non-standard syntax
- Detected potential bad dependencies in Chromium

Future Work

More Advanced Analysis Based on Precise Dependency Data

- Modularity Violation Detection
- Invalid Dependency Injection Diagnosis
- Large-scale Refactoring Assistance