SAMSUNG

Summary-based inter-unit analysis for Clang Static Analyzer

Aleksei Sidorin

2016-11-01

Samsung R&D Institute, Russia

Clang Static Analyzer

SAMSUNG

- Source-based analysis of high-level programming languages (C, C++, Objective-C)
- Simple and powerful Checker API
- Context-sensitive interprocedural analysis with inlining
- This talk is devoted to enhancement of IPA

```
    if (s->msg_callback)
```

1 Taking false branch →

```
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                &s->s3->rrec.data[0], s->s3->rrec.length,
                s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)
```

2 ← Assuming 'hbtype' is equal to 1 →

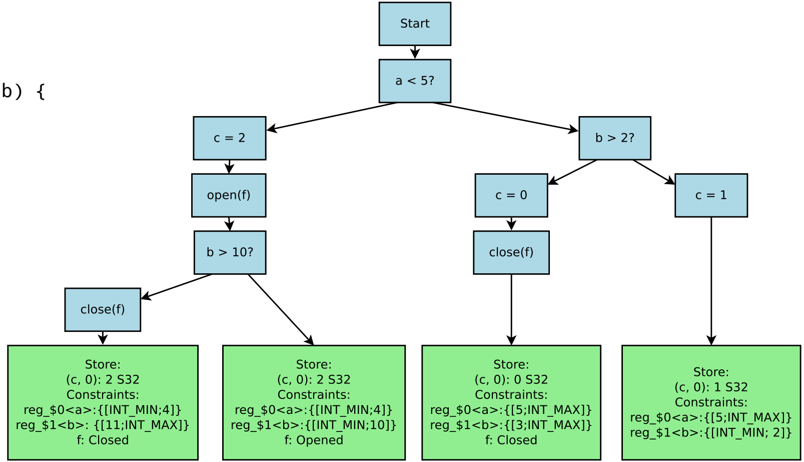
3 ← Taking true branch →

```
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 bytes
         * message type, plus 2 bytes payload length, plus
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);
        memcpy(bp, pl, payload);
```

4 ← Tainted, unconstrained value used in memcpy size

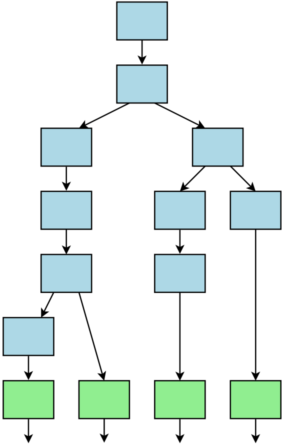
Symbolic execution with CSA

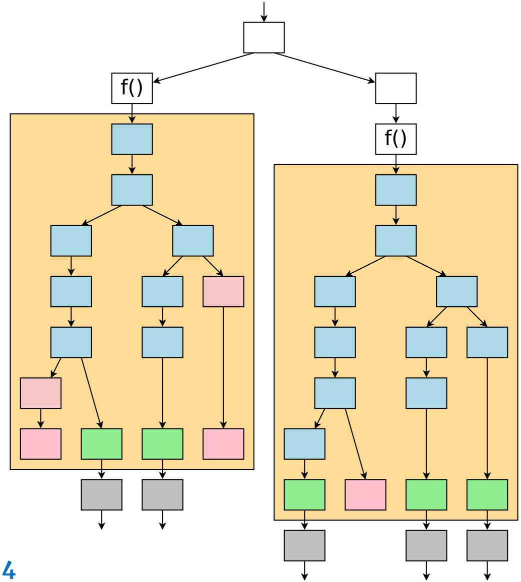
```c
int c;

void func(FILE *f, int a, int b) {
  if (a < 5) {
    c = 2;
    open(f);
    if (b > 10)
      close(f);

  } else {
    if (b > 2) {
      c = 0;
      close(f);
    } else {
      c = 1;
    }
  }
}
```



Analysis with inlining

Callee’s exploded graph

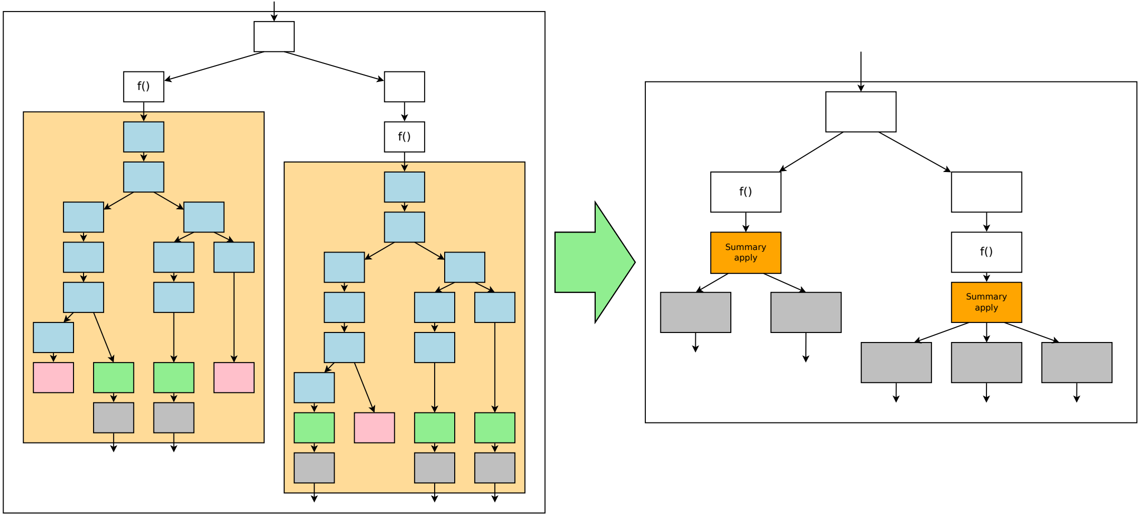




Summary-based analysis

- Don't reanalyze every statement in callee function every time
- Instead, generate only output nodes based on previous analysis of callee function
- Restore effects of function execution using final states of its `ExplodedGraph`
- Remember the nodes in the callee graph where bug *may* occur but we cannot say it definitely
- Check these nodes again while applying a summary with an updated `ProgramState`
- Can be enabled with setting of `-analyzer-config` to `ipa=summary`

Exploded graph with “summary” nodes



Collecting summary

SAMSUNG

- First, we introduced a special callback `evalSummaryPopulate`
- Then, we started extracting the information directly from the state in the final node
- Some additional entries in the `ProgramState` for deferred checks may be still required
- We need to remember the conditions check is performed with

Applying summary

SAMSUNG

For each state of function summary final node:

1. *Actualize* all symbolic values, regions and symbols
   - We replace the symbolic values kept in summary (with their naming in the callee context) with their corresponding values in the caller context
2. Determine if the branch is feasible
   - If all the input ranges of summary branch values have non-empty intersections with ranges of these values in caller, the branch is feasible
   - This intersection of ranges becomes a new range of this value in result branch
3. Invalidate regions that were invalidated in the summary branch
4. Actualize the return value of the function and bind it as the value of call expression
5. Actualize checker-related data

Applying checker summary

SAMSUNG

- Checkers are responsible for their own summary
- A special callback is used in the implementation
- Checkers can update their state to consider changes occurred during function call
- Checkers can perform deferred check if it is not clear in callee context if defect exists or not
- Checkers may split states while applying their summary, as in usual analysis
- Many check kinds may be performed that way

Applying checker summary — example

Source code with double close

```
void closeFile(FILE *f) {
  fclose(f);
}

void doubleClose() {
  FILE *cf = fopen("1.txt", "r");
  closeFile(cf);
  closeFile(cf);
}
```

How checker works

1. Analyze `closeFile()` out of caller context
   - 1.1 Cannot say if it is the second close
   - 1.2 Remember the event node in a separate `ProgramState` trait
   - 1.3 Mark `f` as closed
2. Apply the summary for the first time
   - 2.1 There is a check planned in summary
   - 2.2 Actualization: `f` → `cf`
   - 2.3 `cf` is opened — no actions are required
   - 2.4 Mark `cf` as closed
3. Apply the summary for the second time
   - 3.1 There is a check planned in summary
   - 3.2 Actualization: `f` → `cf`
   - 3.3 `cf` was closed twice! Warn here.

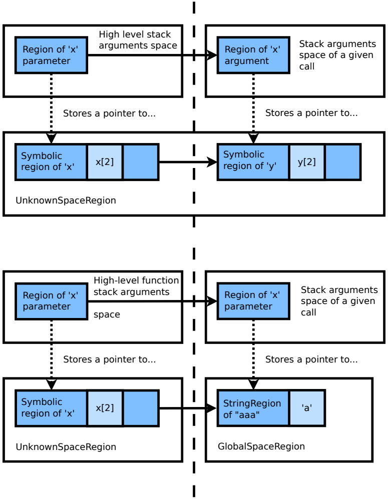
Actualization

SAMSUNG

- We need to know the relation between symbolic values in the caller context and in the callee context
- So, we translate symbolic values from the callee context to the caller context recursively
- All operations on summary applications are done with actualized values
- One symbolic value may contain many references to others
- One of the most complicated parts of summary apply code

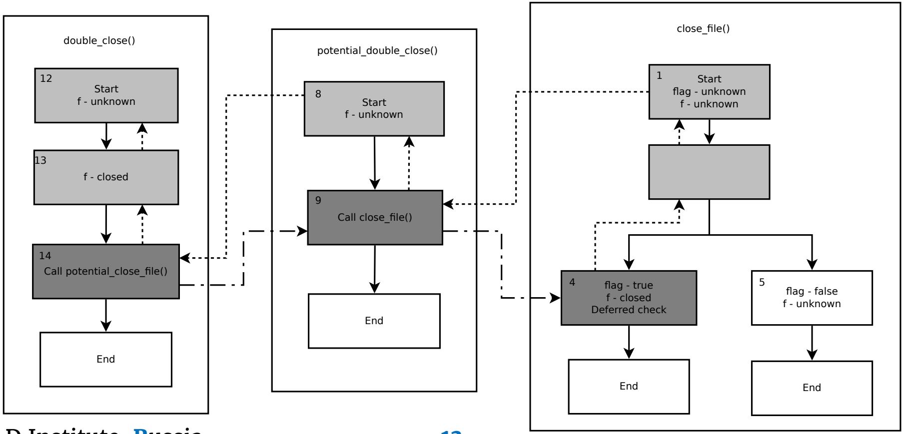
Actualization sample


```
void foo(char *x) {
  if (x[2] == 'a') {}
}


void bar(char *y) {
  foo(y);
  foo("aaa");
}
```



Building interprocedural report

- In summary apply node, we store a pointer to the corresponding final node of callee graph
- For deferred checks, we do the same with the deferred check node



Main results

SAMSUNG

- Faster analysis
  - In the worst case, all the operations with `Store` and `GDM` are repeated while applying a summary
  - But we don't model `Environment` — we don't need it
  - `removeDeadBindings()` is the hottest spot in the whole analyzer code
- More bugs can be found for the same time.

Known issues I

1. Memory optimizations required
   - While using inlining, `ExplodedGraphs` are being deleted after analysis of each function is completed
   - In summary (with current approach), we need to keep the `ExplodedGraphs` of all the callee functions because of deferred checks
   - This leads to much greater memory consumption
2. Checkers should support summary in this implementation
   - Customization of all path-sensitive checkers is… painful
   - Checker writers should know how summary works and be able to use it
   - May lead to mistakes in checker implementation
   - Possible solutions are Smart GDM/Ghost regions or just some ready-for-use templates

Known issues II

3. Limiting analysis time
   - In inlining mode, `max-nodes` setting may be used
   - In summary, every `SummaryPostApply` node corresponds to the whole path in the callee function, but the build time of this node is much greater
   - Currently, we use heuristic of `max-nodes/4`
4. Non-evident warnings may appear
   - In summary, we assume that equivalence classes appear directly while entering the call
   - However, some checkers may be not ready for this
   - Example: DivisionByZeroChecker may report not only div-after-check, but also check-after-div
5. Virtual calls whose object type is unknown are not supported
   - And indirect calls with initially unknown callee as well

Inter-unit analysis prototype

SAMSUNG

Why do we need it?

- To make CSA reason about functions in different translation units
- To decrease a number of functions evaluated conservatively
- To decrease the amount of FPs caused by lack of information about function

How it works?

- Three-stage analysis
  - Build phase: collects information about functions in TUs
  - Pre-analysis: build global call graph and perform topological sorting
  - Analysis: launch `clang` to analyze all the TUs in topological order

Is it usable for other purposes, not CSA-related?

- An open question :)

XTU: build phase

A number of infrastructure tools: some written in Python, some in C++ (`clang`-based)

Usage: `xtu-build.py $build_cmd`

- Intercept compiler calls
  - Currently, we use our `strace`-based solution
  - New interceptor with compilation database building should also be fine
- Dump the information about functions in TU
  - Map function definitions to TUs they located in
  - Dump local call graphs
  - Support multi-arch builds
- Dump ASTs of all translation units

XTU: pre-analysis

SAMSUNG

- Read data generated in the build stage
- Resolve dependencies between functions in different TUs
- Build final mapping between functions and TUs
- Build *global call graph* of the analyzed project
- Sort global call graph in topological order
  - We sort TUs, not functions

XTU: analysis stage

- Launch `clang` for TUs in topological order — in the process pool
- Analyze functions as usually
- If we meet function call with no definition, try to find it in an another TU
- If definition was found:
  - Load corresponding `ASTUnit`
  - Find the function definition
  - Try to import it using `ASTImporter`
  - If import was successful, analyze call as usually
- Generate multi-file report

XTU — toy sample

SAMSUNG

```
% OUT_DIR=.xtu xtu-build.py g++ -c callee.cpp caller.cpp
% xtu-analyze.py --output-dir . --xtu-dir .xtu --enable-checker=core.DivideZero
% cat .xtu/external-map.txt
_Z3divi@x86_64 .xtu/ast/long-path/xtu-sample/callee.cpp.ast
```

report.html

Bug Summary

File:	/media/partition/tmp/xtu-sample/callee.cpp
Location:	line 3, column 13
Description:	Division by zero

Annotated Source Code

```
1
2  int div(int divisor) {
3    return 100/divisor;
       5 ← Division by zero
4  }
```

sub-report.html

```
int div(int);

void caller(int num) {
  if (num == 0) {}
      1 Assuming 'num' is equal to 0 →
    2 ← Taking true branch →
  div(num);
      3 ← Passing the value 0 via 1st parameter 'divisor' →
    4 ← Calling 'div' →
}
```

Pros and cons

SAMSUNG

Good points:

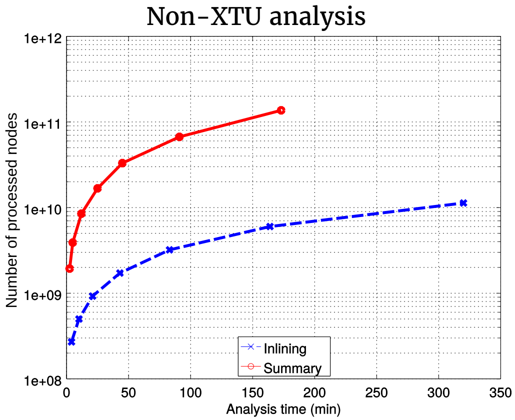
- Transparent analysis — no need in checker support
- All AST information is available without loss

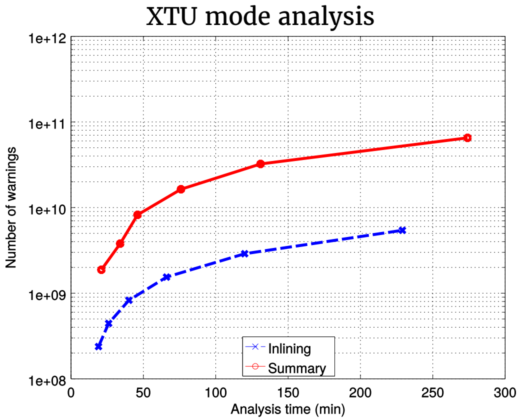
Possible issues:

- Questionable scalability
  - Enough for analyzer but may be not enough for other purposes
- Possible name conflicts
  - Usage of the mangled name for function search is possibly not the best idea
  - We may need to model a linker to avoid name conflicts in large projects
- High disk usage
  - AST dumps consume too much disk space
- May interact with AST-based checkers with changing AST on-the-fly
- Coverage pattern changes too much

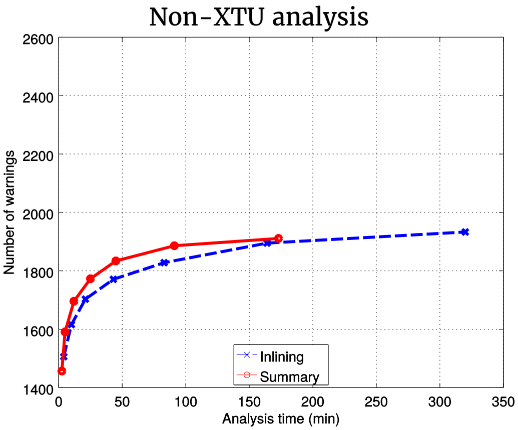
Number of nodes processed per time

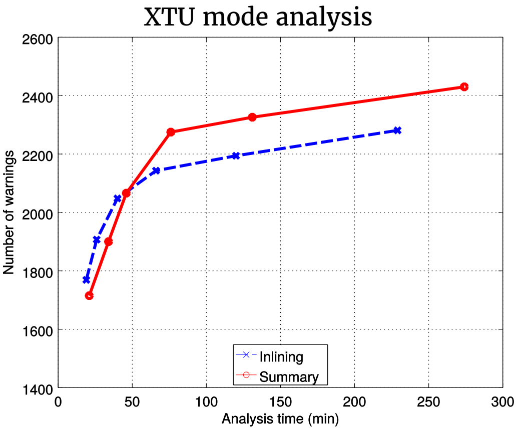
Checkers: ConstModified and IntegerOverflow
Code: AOSP 4.2.1





Unique warnings per time





Acknowledgements

SAMSUNG

- **Artem Dergachev** — for his great input into current design and implementation of summary-based analysis
- **Karthik Bhat** — for the idea of multi-phase analysis
- **Iuliia Trofimovich** — for the implementation of multi-html report
- **Anna Zaks**, **Devin Coughlin**, **Ted Kremenek** — for the help in understanding of different analyzer features and internals
- **Gábor Horváth** — for his investigation of our XTU implementation

Thank you!

SAMSUNG

- Questions?
- Remarks?
- Advice/ideas?

Applying checker summary — example

SAMSUNG

Source code with possible integer overflow

```
char add(int a, int b) {
  return a + b;
}

void overflow(int ca, int cb) {
  if (ca == INT_MAX) {
    if (cb == INT_MAX) {}
    add(ca, cb);
  }
}
```

How checker works

1. Analyze `add()` out of caller context
   - 1.1 Cannot say if overflow happens or not
   - 1.2 Remember the event node in a separate `ProgramState` trait
2. Apply the summary for the first execution branch
   - 2.1 There is a check planned in summary
   - 2.2 Actualization: `a` → `ca`, `b` → `cb`
   - 2.3 `ca == INT_MAX` but `cb != INT_MAX`
   - 2.4 Cannot say if overflow happens or not
   - 2.5 Remember the event node in a separate `ProgramState` trait
3. Apply the summary for the second execution branch
   - 3.1 There is a check planned in summary
   - 3.2 Actualization: `a` → `ca`, `b` → `cb`
   - 3.3 `ca == INT_MAX` and `cb == INT_MAX`
   - 3.4 It's an overflow! Warn here.