This Page Is Under Construction

Checker Developer Manual

The static analyzer engine performs path-sensitive exploration of the program and relies on a set of checkers to implement the logic for detecting and constructing specific bug reports. Anyone who is interested in implementing their own checker, should check out the Building a Checker in 24 Hours talk (slides video) and refer to this page for additional information on writing a checker. The static analyzer is a part of the Clang project, so consult Hacking on Clang and LLVM Programmer's Manual for developer guidelines and send your questions and proposals to cfe-dev mailing list.

Getting Started

- To check out the source code and build the project, follow steps 1-4 of the Clang Getting Started page.
- The analyzer source code is located under the Clang source tree:
  $ cd llvm/tools/clang
  See: include/clang/StaticAnalyzer, lib/StaticAnalyzer, test/Analysis.
- The analyzer regression tests can be executed from the Clang's build directory:
  $ cd ../../../; cd build/tools/clang; TESTDIRS=Analysis make test
- Analyze a file with the specified checker:
  $ clang -cc1 -analyze -analyzer-checker=core.DivideZero test.c
- List the available checkers:
  $ clang -cc1 -analyzer-checker-help
- See the analyzer help for different output formats, fine tuning, and debug options:
  $ clang -cc1 -help | grep "analyzer"

Static Analyzer Overview

The analyzer core performs symbolic execution of the given program. All the input values are represented with symbolic values; further, the engine deduces the values of all the expressions in the program based on the input symbols and the path. The execution is path sensitive and every possible path through the program is explored. The explored execution traces are represented with ExplodedGraph object. Each node of the graph is ExplodedNode, which consists of a ProgramPoint and a ProgramState.

ProgramPoint represents the corresponding location in the program (or the CFG). ProgramPoint is also used to record additional information on when/how the state was added. For example, PostPurgeDeadSymbolsKind kind means that the state is the result of purging dead symbols - the

analyzer's equivalent of garbage collection.

ProgramState represents abstract state of the program. It consists of:

- Environment - a mapping from source code expressions to symbolic values
- Store - a mapping from memory locations to symbolic values
- GenericDataMap - constraints on symbolic values

Interaction with Checkers

Checkers are not merely passive receivers of the analyzer core changes - they actively participate in the ProgramState construction through the GenericDataMap which can be used to store the checker-defined part of the state. Each time the analyzer engine explores a new statement, it notifies each checker registered to listen for that statement, giving it an opportunity to either report a bug or modify the state. (As a rule of thumb, the checker itself should be stateless.) The checkers are called one after another in the predefined order; thus, calling all the checkers adds a chain to the ExplodedGraph.

Representing Values

During symbolic execution, SVal objects are used to represent the semantic evaluation of expressions. They can represent things like concrete integers, symbolic values, or memory locations (which are memory regions). They are a discriminated union of "values", symbolic and otherwise. If a value isn't symbolic, usually that means there is no symbolic information to track. For example, if the value was an integer, such as 42, it would be a ConcreteInt, and the checker doesn't usually need to track any state with the concrete number. In some cases, SVal is not a symbol, but it really should be a symbolic value. This happens when the analyzer cannot reason about something (yet). An example is floating point numbers. In such cases, the SVal will evaluate to UnknownVal. This represents a case that is outside the realm of the analyzer's reasoning capabilities. SVals are value objects and their values can be viewed using the .dump() method. Often they wrap persistent objects such as symbols or regions.

SymExpr (symbol) is meant to represent abstract, but named, symbolic value. Symbols represent an actual (immutable) value. We might not know what its specific value is, but we can associate constraints with that value as we analyze a path. For example, we might record that the value of a symbol is greater than 0, etc.

MemRegion is similar to a symbol. It is used to provide a lexicon of how to describe abstract memory. Regions can layer on top of other regions, providing a layered approach to representing memory. For example, a struct object on the stack might be represented by a VarRegion, but a FieldRegion which is a subregion of the VarRegion could be used to represent the memory associated with a specific field of that object. So how do we represent symbolic memory regions? That's what SymbolicRegion is for. It is a MemRegion that has an associated symbol. Since the symbol is unique and has a unique name; that symbol names the region.

Let's see how the analyzer processes the expressions in the following example:

```
int foo(int x) {
  int y = x * 2;
  int z = x;
  ...
}
```

Let's look at how x*2 gets evaluated. When x is evaluated, we first construct an SVal that represents the lvalue of x, in this case it is an SVal that references the MemRegion for x. Afterwards, when we do the lvalue-to-rvalue conversion, we get a new SVal, which references the value **currently bound** to x. That value is symbolic; it's whatever x was bound to at the start of the function. Let's call that symbol $0. Similarly, we evaluate the expression for 2, and get an SVal that references the concrete number 2. When we evaluate x*2, we take the two SVals of the subexpressions, and create a new SVal that represents their multiplication (which in this case is a new symbolic expression, which we might call $1). When we evaluate the assignment to y, we again compute its lvalue (a MemRegion), and then bind the SVal for the RHS (which references the symbolic value $1) to the MemRegion in the symbolic store.
The second line is similar. When we evaluate x again, we do the same dance, and create an SVal that references the symbol $0. Note, two SVals might reference the same underlying values.

To summarize, MemRegions are unique names for blocks of memory. Symbols are unique names for abstract symbolic values. Some MemRegions represents abstract symbolic chunks of memory, and thus are also based on symbols. SVals are just references to values, and can reference either MemRegions, Symbols, or concrete values (e.g., the number 1).

Idea for a Checker

Here are several questions which you should consider when evaluating your checker idea:

- Can the check be effectively implemented without path-sensitive analysis? See AST Visitors.
- How high the false positive rate is going to be? Looking at the occurrences of the issue you want to write a checker for in the existing code bases might give you some ideas.
- How the current limitations of the analysis will effect the false alarm rate? Currently, the analyzer only reasons about one procedure at a time (no inter-procedural analysis). Also, it uses a simple range tracking based solver to model symbolic execution.
- Consult the Bugzilla database to get some ideas for new checkers and consider starting with improving/fixing bugs in the existing checkers.

Once an idea for a checker has been chosen, there are two key decisions that need to be made:

- Which events the checker should be tracking. This is discussed in more detail in the section Events, Callbacks, and Checker Class Structure.
- What checker-specific data needs to be stored as part of the program state (if any). This should be minimized as much as possible. More detail about implementing custom program state is given in section Custom Program States.

Checker Registration

All checker implementation files are located in clang/lib/StaticAnalyzer/Checkers folder. The steps below describe how the checker SimpleStreamChecker, which checks for misuses of stream APIs, was registered with the analyzer. Similar steps should be followed for a new checker.

1. A new checker implementation file, SimpleStreamChecker.cpp, was created in the directory lib/StaticAnalyzer/Checkers.
2. The following registration code was added to the implementation file:

```
void ento::registerSimpleStreamChecker(CheckerManager &mgr) {
  mgr.registerChecker<SimpleStreamChecker>();
}
```

3. A package was selected for the checker and the checker was defined in the table of checkers at include/clang/StaticAnalyzer/Checkers /Checkers.td. Since all checkers should first be developed as "alpha", and the SimpleStreamChecker performs UNIX API checks, the correct package is "alpha.unix", and the following was added to the corresponding UnixAlpha section of Checkers.td:

```
let ParentPackage = UnixAlpha in {
...
def SimpleStreamChecker : Checker<"SimpleStream">,
  HelpText<"Check for misuses of stream APIs">,
  DescFile<"SimpleStreamChecker.cpp">;
...
} // end "alpha.unix"
```

4. The source code file was made visible to CMake by adding it to lib/StaticAnalyzer/Checkers/CMakeLists.txt.

After adding a new checker to the analyzer, one can verify that the new checker was successfully added by seeing if it appears in the list of available checkers:
$clang -cc1 -analyzer-checker-help

Events, Callbacks, and Checker Class Structure

All checkers inherit from the Checker template class; the template parameter(s) describe the type of events that the checker is interested in processing. The various types of events that are available are described in the file CheckerDocumentation.cpp

For each event type requested, a corresponding callback function must be defined in the checker class (CheckerDocumentation.cpp shows the correct function name and signature for each event type).

As an example, consider SimpleStreamChecker. This checker needs to take action at the following times:

- Before making a call to a function, check if the function is fclose. If so, check the parameter being passed.
- After making a function call, check if the function is fopen. If so, process the return value.
- When values go out of scope, check whether they are still-open file descriptors, and report a bug if so. In addition, remove any information about

them from the program state in order to keep the state as small as possible.

- When file pointers "escape" (are used in a way that the analyzer can no longer track them), mark them as such. This prevents false positives in the cases where the analyzer cannot be sure whether the file was closed or not.

These events that will be used for each of these actions are, respectively, PreCall, PostCall, DeadSymbols, and PointerEscape. The high-level structure of the checker's class is thus:

```
class SimpleStreamChecker : public Checker<check::PreCall,
                                           check::PostCall,
                                           check::DeadSymbols,
                                           check::PointerEscape> {
public:

  void checkPreCall(const CallEvent &Call, CheckerContext &C) const;

  void checkPostCall(const CallEvent &Call, CheckerContext &C) const;

  void checkDeadSymbols(SymbolReaper &SR, CheckerContext &C) const;

  ProgramStateRef checkPointerEscape(ProgramStateRef State,
                                     const InvalidatedSymbols &Escaped,
                                     const CallEvent *Call,
                                     PointerEscapeKind Kind) const;
};
```

Custom Program States

Checkers often need to keep track of information specific to the checks they perform. However, since checkers have no guarantee about the order in which the program will be explored, or even that all possible paths will be explored, this state information cannot be kept within individual checkers. Therefore, if checkers need to store custom information, they need to add new categories of data to the ProgramState. The preferred way to do so is to use one of several macros designed for this purpose. They are:

- REGISTER_TRAIT_WITH_PROGRAMSTATE: Used when the state information is a single value. The methods available for state types declared with this macro are get, set, and remove.
- REGISTER_LIST_WITH_PROGRAMSTATE: Used when the state information is a list of values. The methods available for state types declared with this macro are add, get, remove, and contains.
- REGISTER_SET_WITH_PROGRAMSTATE: Used when the state information is a set of values. The methods available for state types declared with this macro are add, get, remove, and contains.
- REGISTER_MAP_WITH_PROGRAMSTATE: Used when the state information is a map from a key to a value. The methods available for state types declared with this macro are add, set, get, remove, and contains.

All of these macros take as parameters the name to be used for the custom category of state information and the data type(s) to be used for storage. The data type(s) specified will become the parameter type and/or return type of the methods that manipulate the new category of state information. Each of these methods are templated with the name of the custom data type.

For example, a common case is the need to track data associated with a symbolic expression; a map type is the most logical way to implement this. The key for this map will be a pointer to a symbolic expression (SymbolRef). If the data type to be associated with the symbolic expression is an integer, then the custom category of state information would be declared as

```
REGISTER_MAP_WITH_PROGRAMSTATE(ExampleDataType, SymbolRef, int)
```

The data would be accessed with the function

```
ProgramStateRef state;
SymbolRef Sym;
...
int currentlValue = state->get<ExampleDataType>(Sym);
```

and set with the function

```
ProgramStateRef state;
SymbolRef Sym;
int newValue;
...
ProgramStateRef newState = state->set<ExampleDataType>(Sym, newValue);
```

In addition, the macros define a data type used for storing the data of the new data category; the name of this type is the name of the data category with "Ty" appended. For REGISTER_TRAIT_WITH_PROGRAMSTATE, this will simply be passed data type; for the other three macros, this will be a specialized version of the llvm::ImmutableList, llvm::ImmutableSet, or llvm::ImmutableMap templated class. For the ExampleDataType example above, the type created would be equivalent to writing the declaration:

```
typedef llvm::ImmutableMap<SymbolRef, int> ExampleDataTypeTy;
```

These macros will cover a majority of use cases; however, they still have a few limitations. They cannot be used inside namespaces (since they expand to contain top-level namespace references), and the data types that they define cannot be referenced from more than one file.

Note that ProgramStates are immutable; instead of modifying an existing one, functions that modify the state will return a copy of the previous state with the change applied. This updated state must be then provided to the analyzer core by calling the CheckerContext::addTransition function.

Bug Reports

When a checker detects a mistake in the analyzed code, it needs a way to report it to the analyzer core so that it can be displayed. The two classes used to construct this report are BugType and BugReport.

BugType, as the name would suggest, represents a type of bug. The constructor for BugType takes two parameters: The name of the bug type, and the name of the category of the bug. These are used (e.g.) in the summary page generated by the scan-build tool.

The BugReport class represents a specific occurrence of a bug. In the most common case, three parameters are used to form a BugReport:

1. The type of bug, specified as an instance of the BugType class.
2. A short descriptive string. This is placed at the location of the bug in the detailed line-by-line output generated by scan-build.
3. The context in which the bug occurred. This includes both the location of the bug in the program and the program's state when the location is reached. These are both encapsulated in an ExplodedNode.

In order to obtain the correct ExplodedNode, a decision must be made as to whether or not analysis can continue along the current path. This decision is based on whether the detected bug is one that would prevent the program under analysis from continuing. For example, leaking of a resource should not stop analysis, as the program can continue to run after the leak. Dereferencing a null pointer, on the other hand, should stop analysis, as there is no way for the program to meaningfully continue after such an error.

If analysis can continue, then the most recent ExplodedNode generated by the checker can be passed to the BugReport constructor without additional modification. This ExplodedNode will be the one returned by the most recent call to CheckerContext::addTransition. If no transition has been performed during the current callback, the checker should call CheckerContext::addTransition() and use the returned node for bug reporting.

If analysis can not continue, then the current state should be transitioned into a so-called *sink node*, a node from which no further analysis will be performed. This is done by calling the CheckerContext::generateSink function; this function is the same as the addTransition function, but marks the state as a sink node. Like addTransition, this returns an ExplodedNode with the updated state, which can then be passed to the BugReport constructor.

After a BugReport is created, it should be passed to the analyzer core by calling CheckerContext::emitReport.

AST Visitors

Some checks might not require path-sensitivity to be effective. Simple AST walk might be sufficient. If that is the case, consider implementing a Clang compiler warning. On the other hand, a check might not be acceptable as a compiler warning; for example, because of a relatively high false positive rate. In this situation, AST callbacks checkASTDecl and checkASTCodeBody are your best friends.

Testing

Every patch should be well tested with Clang regression tests. The checker tests live in clang/test/Analysis folder. To run all of the analyzer tests,

execute the following from the clang build directory:

```
$ bin/llvm-lit -sv ../llvm/tools/clang/test/Analysis
```

Useful Commands/Debugging Hints

Attaching the Debugger

When your command contains the -cc1 flag, you can attach the debugger to it directly:

```
$ gdb --args clang -cc1 -analyze -analyzer-checker=core test.c
$ lldb -- clang -cc1 -analyze -analyzer-checker=core test.c
```

Otherwise, if your command line contains --analyze, the actual clang instance would be run in a separate process. In order to debug it, use the -### flag for obtaining the command line of the child process:

```
$ clang --analyze test.c -\#\#\#
```

Below we describe a few useful command line arguments, all of which assume that you are running clang -cc1.

Narrowing Down the Problem

While investigating a checker-related issue, instruct the analyzer to only execute a single checker:

```
$ clang -cc1 -analyze -analyzer-checker=osx.KeychainAPI test.c
```

If you are experiencing a crash, to see which function is failing while processing a large file use the -analyzer-display-progress option.

To selectively analyze only the given function, use the -analyze-function option:

```
$ clang -cc1 -analyze -analyzer-checker=core test.c -analyzer-display-progress
ANALYZE (Syntax): test.c foo
ANALYZE (Syntax): test.c bar
ANALYZE (Path,  Inline_Regular): test.c bar
ANALYZE (Path,  Inline_Regular): test.c foo
$ clang -cc1 -analyze -analyzer-checker=core test.c -analyzer-display-progress -analyze-function=foo
ANALYZE (Syntax): test.c foo
ANALYZE (Path,  Inline_Regular): test.c foo
```

Note: a fully qualified function name has to be used when selecting C++ functions and methods, Objective-C methods and blocks, e.g.:

```
$ clang -cc1 -analyze -analyzer-checker=core test.cc -analyze-function=foo(int)
```

The fully qualified name can be found from the -analyzer-display-progress output.

The bug reporter mechanism removes path diagnostics inside intermediate function calls that have returned by the time the bug was found and contain no interesting pieces. Usually it is up to the checkers to produce more interesting pieces by adding custom BugReporterVisitor objects. However, you can disable path pruning while debugging with the -analyzer-config prune-paths=false option.

Visualizing the Analysis

To dump the AST, which often helps understanding how the program should behave:

```
$ clang -cc1 -ast-dump test.c
```

To view/dump CFG use debug.ViewCFG or debug.DumpCFG checkers:

```
$ clang -cc1 -analyze -analyzer-checker=debug.ViewCFG test.c
```

ExplodedGraph (the state graph explored by the analyzer) can be visualized with another debug checker:

```
$ clang -cc1 -analyze -analyzer-checker=debug.ViewExplodedGraph test.c
```

Or, equivalently, with -analyzer-viz-egraph-graphviz option, which does the same thing - dumps the exploded graph in graphviz .dot format.

You can convert .dot files into other formats - in particular, converting to .svg and viewing in your web browser might be more comfortable than using a .dot viewer:

```
$ dot -Tsvg ExprEngine-501e2e.dot -o ExprEngine-501e2e.svg
```

The -trim-egraph option removes all paths except those leading to bug reports from the exploded graph dump. This is useful because exploded graphs are often huge and hard to navigate.

Viewing ExplodedGraph is your most powerful tool for understanding the analyzer's false positives, because it gives comprehensive information on every decision made by the analyzer across all analysis paths.

There are more debug checkers available. To see all available debug checkers:

```
$ clang -cc1 -analyzer-checker-help | grep "debug"
```

Debug Prints and Tricks

To view "half-baked" ExplodedGraph while debugging, jump to a frame that has clang::ento::ExprEngine object and execute:

```
(gdb) p ViewGraph(0)
```

To see the ProgramState while debugging use the following command.

```
(gdb) p State->dump()
```

To see clang::Expr while debugging use the following command. If you pass in a SourceManager object, it will also dump the corresponding line in the source code.

```
(gdb) p E->dump()
```

To dump AST of a method that the current ExplodedNode belongs to:

```
(gdb) p C.getPredecessor()->getCodeDecl().getBody()->dump()
```

Additional Sources of Information

Here are some additional resources that are useful when working on the Clang Static Analyzer:

- Clang doxygen. Contains up-to-date documentation about the APIs available in Clang. Relevant entries have been linked throughout this page. Also of use is the LLVM doxygen, when dealing with classes from LLVM.
- The cfe-dev mailing list. This is the primary mailing list used for discussion of Clang development (including static code analysis). The archive also contains a lot of information.
- The "Building a Checker in 24 hours" presentation given at the November 2012 LLVM Developer's meeting. Describes the construction of SimpleStreamChecker. Slides and video are available.

Useful Links

- The list of Implicit Checkers