Clang Static Analyzer

A Checker Developer's Guide

Rev. — April 22, 2016

Contents

0 Preface **4**
- 0.1 FAQ: a quick guide through the guide . . . 5

1 Introduction to the Clang Static Analyzer **7**
- 1.1 MainCallChecker — a simple tutorial checker . . . 7
- 1.2 Checker example code explained . . . 10
  - 1.2.1 Declaring a checker class . . . 10
  - 1.2.2 Implementing checker callbacks . . . 10
  - 1.2.3 Throwing bug reports . . . 11
  - 1.2.4 Registering the checker . . . 11
- 1.3 Compiling the checker as a standalone module . . . 12
- 1.4 Further reading . . . 12

2 Kinds of analyses and program representations **13**
- 2.1 Abstract syntax tree . . . 13
- 2.2 Control flow graph . . . 14
- 2.3 Exploded graph . . . 16
- 2.4 Further reading . . . 17

3 AST-based checkers **18**
- 3.1 Path-insensitive checker callbacks . . . 18
  - 3.1.1 check::EndOfTranslationUnit . . . 18
  - 3.1.2 check::ASTCodeBody . . . 18
  - 3.1.3 check::ASTDecl<T> . . . 19
- 3.2 AST visitors . . . 19
  - 3.2.1 Implementing a simple statement visitor . . . 19
  - 3.2.2 Merging statement and declaration visitors . . . 21
- 3.3 AST matchers . . . 21
  - 3.3.1 Implementing a simple AST matcher . . . 22
  - 3.3.2 Re-using matchers . . . 23
  - 3.3.3 Defining custom matchers . . . 24
  - 3.3.4 Matching particular statements . . . 24
- 3.4 Constant folding . . . 25
- 3.5 Further reading . . . 25

4 Path-sensitive analysis **26**
- 4.1 Obtaining information from the program state . . . 26
  - 4.1.1 Obtaining values of expressions . . . 27
  - 4.1.2 A brief introduction to memory regions . . . 27
  - 4.1.3 Iterating over region store bindings . . . 28
  - 4.1.4 Assumptions on symbolic values . . . 28
  - 4.1.5 Operations on symbolic values . . . 29
  - 4.1.6 Using the taint analysis . . . 29
- 4.2 Mutating and splitting the program state . . . 29
  - 4.2.1 Adding transitions to the exploded graph . . . 29
  - 4.2.2 Splitting the state on range constraint assumptions . . . 30
  - 4.2.3 Creating region store bindings . . . 31
  - 4.2.4 Expanding the taint analysis . . . 31
  - 4.2.5 Using program state traits . . . 32
- 4.3 Path-sensitive checker callbacks . . . 35

4.3.1 check::PreStmt<T> . . . 35
4.3.2 check::PostStmt<T> . . . 35
4.3.3 check::PreCall . . . 36
4.3.4 check::PostCall . . . 36
4.3.5 check::Location . . . 37
4.3.6 check::Bind . . . 38
4.3.7 check::EndAnalysis . . . 39
4.3.8 check::EndFunction . . . 39
4.3.9 check::BranchCondition . . . 41
4.3.10 check::LiveSymbols . . . 41
4.3.11 check::DeadSymbols . . . 42
4.3.12 check::RegionChanges . . . 43
4.3.13 check::PointerEscape . . . 45
4.3.14 eval::Assume . . . 45
4.3.15 eval::Call . . . 46
4.4 Implementing bug reporter visitors . . . 47
4.5 Understanding interprocedural analysis . . . 47
4.5.1 Conservative evaluation and invalidation . . . 48
4.5.2 Inlining and stack frames . . . 48
4.6 Further reading . . . 49

5 The symbolic value hierarchy **50**
5.1 Constructing symbolic values . . . 51
5.2 Memory model of the analyzer . . . 51
5.2.1 Memory spaces . . . 53
5.2.2 Untyped base regions . . . 53
5.2.3 Typed base regions with typed values . . . 54
5.2.4 Typed base regions with untyped values . . . 54
5.2.5 Sub-regions of base regions . . . 55
5.3 Concrete values . . . 56
5.3.1 Numeric values . . . 56
5.3.2 Compound values . . . 56
5.4 Special values . . . 57
5.4.1 UndefinedVal . . . 57
5.4.2 UnknownVal . . . 57
5.5 Symbolic expressions . . . 57
5.5.1 Operation symbols . . . 58
5.5.2 Conjured symbols . . . 59
5.5.3 Region value symbols . . . 59
5.5.4 Extent symbols . . . 59
5.5.5 Metadata symbols . . . 60
5.6 Tainted values . . . 60
5.7 Understanding debug dumps . . . 61
5.8 Further reading . . . 61

Index of notions **62**

Index of classes **63**

0. Preface

The author is greatful to everybody who contributed to this guide by finding mistakes and omissions and giving valuable suggestions — and this list would probably grow as new revisions of this guide are released — in particular, to Alexey Sidorin, Julia Trofimovich, and Kirill Romanenkov.

The guide is still incomplete at parts, and it would need updates in case of changes in the analyzer core, which would inevitably occur at times. Additionally, because the author was not a native speaker of English, any suggestions on improving the grammar aspect of the guide are warmly welcome.

Below is a rough to-do list of stuff that is not yet properly explained in the guide, but would be considered useful to have, in no particular order:

1. Direct and default bindings in the region store.
2. Recent changes in work with live and dead symbols.
3. The `WasInlined` attribute of the checker context.
4. The newly introduced `CodeSpaceRegion` memory space.
5. The syntax for adding bug reporter visitors to the report.
6. How to read the `stderr` dump of the program state.
7. The `loc::GotoLabel` value class.
8. How to use the AST parent map.
9. Use `check::ASTDecl<>`, probably for the whole translation unit, instead of `check::EndOfTranslationUnit` for syntax-only checks.
10. Symbols of structural type.
11. A picture of how super-regions of any memory region usually look.
12. Add a picture of how path-sensitive bug reports look, eg. the HTML ones.
13. Describe more Objective-C-related stuff.
14. Using the `SVal` visitor.
15. Writing tests.
16. Coding style needs updating — outdated constructs are used.

The author welcomes suggestions, bug reports, pull-requests, forks, and whatever may come out of it, on github at https://github.com/haonoq/clang-analyzer-guide!

On the other hand, please do not send analyzer-related questions in private messages or on e-mail! The best place to ask questions is the cfe-dev mailing list — http://lists.llvm.org/pipermail/cfe-dev/, because other people would see the question and the answer, and probably even be able to find the discussion later through web search.

0.1. FAQ: a quick guide through the guide

Q: *Okay, so how do I write CSA checkers?*

A: For a step-by-step quick start guide on coding checkers, see subsection 1.2.

Q: *Reference guides are boring. I prefer learning by example. Should I keep reading?*

A: That's not really a reference guide, but rather a free-hand introduction to Clang Static Analyzer. You'd encounter code samples and useful snippets on almost every page. We did not try to copy the official Clang doxygen, which you would definitely refer to during your work on CSA checkers. However, we also strongly advise you to search through the official checker source code for finding how different classes and methods are used in practice.

Q: *This guide is quite big. My checker only needs to find calls to a certain function, and it shouldn't be hard. Do I really need to read the whole guide to implement it?*

A: For finding simple code patterns, an AST matcher would easily do the job. Probably the example in subsection 3.3 is all you need to know.

Q: *Now I have a real problem. My program crashes, due to double-close of* `FlyingElephantDescriptor`*, once in a few weeks, and I badly want to catch and debug it. Please help!*

A: You came to the right place! If you want a checker that finds a program execution paths on which a certain sequence of events, such as double-free, occurs, then you need to implement a path-sensitive checker. You would probably need to read section 4, most importantly subsections 4.1 and 4.2, paying a lot of attention in 4.2.5, and probably look through subsection 4.3 to find the right callback to hook into.

Q: *So, how do I know if I need a path-sensitive or path-insensitive checker?*

A: It depends on what information you need the analyzer core to provide. If you want to understand this matter in-depth, see section 2. Most of the time, you'd pick a path-sensitive checker. Only if your check is really simple, would you want to rely on AST-based checkers.

Q: *My path-sensitive bug report is too short, I cannot figure out what's going on!*

A: By default, the analyzer doesn't draw path through sub-functions that returned before the bug was found. They only show the event of the bug, and decisions that lead to it in the same function or in its direct callers. If you need to highlight other events, probably inside sub-functions, then you need to implement a bug report visitor, as described in subsection 4.4.

Q: *Wait a minute, section 2 also mentions CFG-based analysis. When do I use this one?*

A: Almost never, unless you really know what you're doing, and in this case you almost certainly don't need this guide. Even though it'd be great to have a rough idea of what CFG is and how it looks, most of the time path-sensitive checkers turn out to do the same job much easier.

Q: *I'm reading the guide randomly, and I've no idea what you mean by "GDM".*

A: You can always refer to the alphabetical index at the end of the guide. In fact, because the index is quite short, you may also read through it to find things you missed. The index of classes also highlights most useful methods in CSA classes and points to usage examples for each class or method across the guide.

Q: *Your path-sensitive engine is fantastic! How does it work?*

A: The easiest way to explain how it works is probably say "it constructs an exploded graph". Even though the whole section 4 is about how the path-sensitive engine works, you may also refer to the explanation of the exploded graph in subsection 2.3 for clearer understanding.

1. Introduction to the Clang Static Analyzer

The Clang compiler, based on the LLVM infrastructure, provides much more than a way to turn your C, C++, or Objective-C code into a binary executable file. Clang allows reliably hooking onto the compilation process and obtaining exhaustive information of the data structures the compiler generates on each phase of the compilation. In other words, if you want to know more about your program, the compiler is the best person to ask — and Clang is open to answer your questions. Assuming you ask the right questions, that is.

One of the applications for Clang tools is automatically finding defects in programs, providing much more warnings than your compiler would. For instance, the `clang-tidy` tool finds style issues and unsafe or potentially unportable constructs by observing the syntax used in the program.

Clang Static Analyzer is another tool that finds defects in programs. By exploring the program source code, this particular tool tries to execute parts of the program without compiling them or running the program — as if reading the source code and imagining what would happen if it runs — and reports run-time errors that would occur in such imaginary run-time. Because actual behavior of any real-world program depends on external factors, such as input values, random numbers, and behavior of library components (for which source code is not always available!), the analyzer engine denotes unknown values with algebraic symbols, and performs symbolic computations based on these symbols. It also discovers conditions on the symbolic values that lead the program towards the error.

As a result, Clang Static Analyzer is capable of finding deep bugs that occur only on rare program paths. These paths might have been missed by the manual testers or the automated test suites. Upon finding a bug, the analyzer draws the whole path that lead to the bug, with jump directions on each conditional statement.

However, the analyzer can only find bugs that it has been specifically engineered to find. Otherwise, upon encountering the problem, the analysis runs further and doesn't notice anything. For every particular kind of defects the analyzer finds, such as dereference of a null pointer or buffer overflow, there is a special module — a *checker*, that reacts on such defects during analysis.

So, essentially, the analyzer core is responsible for executing the program in a symbolic manner, and checkers subscribe on events they're interested in, check various assumptions on symbolic values at these events, and throw warnings if these assumptions are found to fail on the given path.

It means that you may want to not only use the analyzer to finds defects, but also adapt it to your particular project. For instance, you may want to enforce rules specific to your project, or find misuses of a specific library API you are using. In order to do that, you may find yourself wanting to write a new checker module for the analyzer. And no matter how easy it may be — because Clang Static Analyzer is a very easy tool — this guide should be able to help you.

1.1. MainCallChecker — a simple tutorial checker

For a quick start, we shall write a simple, though probably not very useful, static analyzer checker. The checker would find violations of the following rule defined by the C++ standard:

> **basic.start.main.3:** The function `main` shall not be used within a program.

In other words, the `main()` function cannot be recursive; the program should never call `main()`, otherwise behavior is undefined. Finding such defect sounds easy at a glance: just see if there's a function call in the

program, and the function has name “main”. Well, not in real life. The programmer may put a pointer to `main` into a variable, pass this pointer around, and then accidentally call a function by pointer, which accidentally turns out to point to `main`:

```
typedef int (*main_t)(int, char **);
int main(int argc, char **argv) {
  main_t foo = main;
  int exit_code = foo(argc, argv);  // actually calls main()!
  return exit_code;
}
```

Example_Test.c

So even in this simple case, the analyzer’s path-sensitive engine has an advantage over a simple syntax-based check. Let’s see if we can detect the error in `Example_Test.c` with the help of the static analyzer.

All right, you got me: even putting the pointer to `main` into a variable actually already means that `main` was “used” within the program. But for educational purposes, we shall find out that in fact it is actually called.

First, let us provide a definition of the checker in the list of checkers. Open up `lib/StaticAnalyzer/Checkers/Checkers.td` in the clang source tree, and add a simple description of the checker somewhere too, say, `alpha.core` package of checkers:

```
...
 HelpText<"Check for assignment of a fixed address to a pointer">,
 DescFile<"FixedAddressChecker.cpp">;

def MainCallChecker : Checker<"MainCall">,
 HelpText<"Check for calls to main">,
 DescFile<"MainCallChecker.cpp">;

def PointerArithChecker : Checker<"PointerArithm">,
 HelpText<"Check for pointer arithmetic on locations other than array elements">,
 DescFile<"PointerArithChecker">;
...
```

Checkers.td

After re-compiling Clang, this would make the checker appear in the list of checkers:

```
~ $ clang -cc1 -analyzer-checker-help
OVERVIEW: Clang Static Analyzer Checkers List

USAGE: -analyzer-checker <CHECKER or PACKAGE,...>

CHECKERS:
...
  alpha.core.FixedAddr            Check for assignment of a fixed address to a p
ointer
  alpha.core.IdenticalExpr        Warn about unintended use of identical express
ions in operators
  alpha.core.MainCall             Check for calls to main
  alpha.core.PointerArithm        Check for pointer arithmetic on locations othe
r than array elements
  alpha.core.PointerSub           Check for pointer subtractions on two pointers
 pointing to different memory chunks
...
```

In this example, "alpha.core.MainCallChecker" is the name of the checker in the registry. Once the checker is registered, it can be enabled via CSA command-line options by the given name, and also the relevant short description line appears in the analyzer checker help. "alpha.core" is the category of the checker. For example, -analyzer-checker alpha.core would enable all checkers in the alpha.core category.

Then, add the checker code to the lib/StaticAnalyzer/Checkers/CMakeLists.txt file, so that its source code got eventually compiled on the next rebuild of Clang:

```
...
  LocalizationChecker.cpp
  MacOSKeychainAPIChecker.cpp
  MacOSXAPIChecker.cpp
  MainCallChecker.cpp
  MallocChecker.cpp
  MallocOverflowSecurityChecker.cpp
...
```

CMakeLists.txt

Finally, write some code in lib/StaticAnalyzer/Checkers/MainCallChecker.cpp that we will soon explain:

```
1  #include "ClangSACheckers.h"
2  #include "clang/StaticAnalyzer/Core/BugReporter/BugType.h"
3  #include "clang/StaticAnalyzer/Core/Checker.h"
4  #include "clang/StaticAnalyzer/Core/PathSensitive/CallEvent.h"
5  #include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
6
7  using namespace clang;
8  using namespace clang::ento;
9
10 namespace {
11 class MainCallChecker : public Checker<check::PreCall> {
12   mutable std::unique_ptr<BugType> BT;
13
14 public:
15   void checkPreCall(const CallEvent &Call, CheckerContext &C) const;
16 };
17 }
18
19 void MainCallChecker::checkPreCall(const CallEvent &Call,
20                                    CheckerContext &C) const {
21   if (const IdentifierInfo *II = Call.getCalleeIdentifier())
22     if (II->isStr("main")) {
23       if (!BT)
24         BT.reset(new BugType(this, "Call to main", "Example checker"));
25       ExplodedNode *N = C.generateErrorNode();
26       auto Report = llvm::make_unique<BugReport>(*BT, BT->getName(), N);
27       C.emitReport(std::move(Report));
28     }
29 }
30
31 void ento::registerMainCallChecker(CheckerManager &Mgr) {
32   Mgr.registerChecker<MainCallChecker>();
33 }
```

MainCallChecker.cpp

After compiling clang, you should be able to run the static analyzer, enable the checker, and see the warning:

```
~ $ clang -cc1 -analyze -analyzer-checker=alpha.core Example_Test.c
Example_Test.c:4:19: warning: Call to main
  int exit_code = foo(argc, argv); // actually calls main()!
                  ^~~~~~~~~~~~~~~
1 warning generated.
```

1.2. Checker example code explained

Now let us figure out how `MainCallChecker` works internally. `MainCallChecker` is a *path-sensitive* checker: it can detect how values flow through variables on different program paths, and understand which execution paths are taken based on these values. We have already demonstrated this in `Example_Test.c`, where we call a function after storing a pointer to this function in a variable `foo`.

1.2.1. Declaring a checker class

A CSA checker is implemented by inheriting from a class template `Checker<...>`, in which template parameters indicate the list of callbacks on which the checker subscribes:

```
10 namespace {
11 class MainCallChecker : public Checker<check::PreCall> {
12   mutable std::unique_ptr<BugType> BT;
13
14 public:
15   void checkPreCall(const CallEvent &Call, CheckerContext &C) const;
16 };
17 }
```

Checker class definitions are usually put into anonymous namespaces to avoid name collisions upon loading multiple checkers into the analyzer.

`MainCallChecker` subscribes to the `check::PreCall` event. The `checkPreCall(...)` callback defined inside the checker will be called every time the path-sensitive engine of the analyzer encounters a function call and is about to analyze it.

1.2.2. Implementing checker callbacks

Now let us look at the implementation of the `checkPreCall(...)` callback:

```
19 void MainCallChecker::checkPreCall(const CallEvent &Call,
20                                     CheckerContext &C) const {
21   if (const IdentifierInfo *II = Call.getCalleeIdentifier())
22     if (II->isStr("main")) {
23       if (!BT)
24         BT.reset(new BugType(this, "Call to main", "Example checker"));
25       ExplodedNode *N = C.generateErrorNode();
26       auto Report = llvm::make_unique<BugReport>(*BT, BT->getName(), N);
27       C.emitReport(std::move(Report));
28     }
29 }
```

The `CallEvent` structure available at the callback contains all the data on the function call event the analyzer core managed to gather for us. In particular, it contains information about the callee function, and values of the arguments.

Because our checker is path-sensitive, this information is a lot more than you may obtain by looking at the syntax tree. In particular, it may know the callee identifier even if a function is called by function pointer, because the analyzer core have predicted the value of this pointer on this execution path. On line 21, we use this to obtain the *identifier* (`IdentifierInfo` structure) for the callee function from the `CallEvent` structure. In case we cannot obtain such info, that is, if `getCalleeIdentifier()` returns a `NULL` pointer, we return from our callback and continue the analysis.

Now, on line 22, we see if the identifier we encountered has the name "`main`". We are only interested in functions with the name "`main`", so all further checks are made under this assumption.

1.2.3. Throwing bug reports

That's it for the checker logic. What remains is to produce a bug report for the user. For this, we use another object available in our callback, the `CheckerContext` structure. This structure is a Swiss Army knife that contains various functions checkers can use to obtain information on the analysis and affect the analysis flow.

There's a variable `BT` in the checker, which stores a "bug type" for the checker — a common way to identify bugs belonging to different checkers. A checker may have multiple bug types; they are traditionally stored and re-used inside the checker for performance. On line 24, the checker initializes its bug type structure `BT` as a bug called "Call to main", within category "Example checker", unless it is already initialized.

On line 25, we use `CheckerContext` to generate a *sink node*, which means that the program would most likely crash after encountering this defect, and it is pointless to continue the analysis beyond this point. The node itself represents a point in the execution path. It is not necessary for the checker to stop the analysis once it finds a defect, if the defect is not critical.

Finally, on line 26, the checker creates a new `BugReport` object. The report is thrown *against* the sink node we generated before. `BugReport` also contains the warning message, which in our case coincides with the bug type name.

Then we pass the report back to the `CheckerContext` using the `emitReport(...)` method. Reports would be gathered together, de-duplicated, and displayed to the user in the preferred manner.

1.2.4. Registering the checker

Finally, there is a small piece of magic code to actually create the checker instance when the analysis starts. You may use this section to disable certain checkers for the whole translation units (eg. checkers for C++-only defects in plain C files), introduce dependencies between checkers, or to set checker options. The code below creates exactly one instance of the `MainCallChecker` to feed upon the events yet to unfold:

```
31 void ento::registerMainCallChecker(CheckerManager &Mgr) {
32   Mgr.registerChecker<MainCallChecker>();
33 }
```

1.3. Compiling the checker as a standalone module

We have been compiling the checker inside the Clang source tree. However, it is possible to compile the checker as a shared plugin library instead. In this case, you don't need to modify `Checkers.td` or `CMakeLists.txt` in order to run the checker; instead, you compile the checker as a standalone library, and load it in run-time.

The syntax for registering the checker changes in the case of compiling as a plugin. You don't need to include the `ClangSACheckers.h` header, but instead you include the `CheckerRegistry.h` header:

```
#include "clang/StaticAnalyzer/Core/CheckerRegistry.h"
```

Then we define an externally visible function in our library that would register the checker dynamically in the analyzer's `CheckerRegistry`:

```
extern "C"
void clang_registerCheckers (CheckerRegistry &registry) {
  registry.addChecker<MainCallChecker>("alpha.core.MainCallChecker",
                                       "Checks for calls to main");
}
```

The clang API version needs to match the plugin API version. Hence, the checker needs to store its version string in the externally visible `clang_analyzerAPIVersionString` variable for the purpose of compatibility checking:

```
extern "C" const char clang_analyzerAPIVersionString[] =
    CLANG_ANALYZER_API_VERSION_STRING;
```

Once you load the checker via the usual clang plugin syntax — `clang -cc1 -load Checker.so` — it would appear as a normal checker in the `-analyzer-checker-help` list, and you should be able to enable it via `-analyzer-checker`.

1.4. Further reading

The quick introduction in this section covers only a very little part of Clang Static Analyzer capabilities. There are various ways of learning how to develop CSA checkers efficiently.

Further sections of this guide should give you an idea of how the analyzer works and what sort of information is available for the checkers to use, and cover various technologies and tricks involved in creating a checker.

However, while developing CSA checkers, you would also inevitably consult the *official LLVM*[1] *and Clang*[2] *documentation*, which contains exhaustive information on classes, functions, and data structures you would regularly encounter.

CSA website contains a quick-start checker development manual[3]. We also cannot avoid mentioning a highly recommended presentation video by CSA developers "Building a Checker in 24 hours"[4], which describes how a slightly more complicated path-sensitive checker works.

[1] `http://llvm.org/doxygen`
[2] `http://clang.llvm.org/doxygen`
[3] `http://clang-analyzer.llvm.org/checker_dev_manual.html`
[4] `http://llvm.org/devmtg/2012-11/videos/Zaks-Rose-Checker24Hours.mp4`

2. Kinds of analyses and program representations

The first decision you usually need to make when you create a checker is whether you need path-sensitivity to implement the desired check. Alternatively, you may implement your check by exploring the of the program on syntax level.

Path-sensitive analysis is usually many times slower than compilation. However, most of the time is taken by the analyzer core to construct the necessary data structures; the checkers are usually lightweight, unless some extremely heavy calculations were explicitly required. So if you are already running at least one path-sensitive checker, then adding another path-sensitive checker would not make the analysis significantly slower.

On the contrary, syntax-only analysis is usually as fast as compilation, or even faster, because code generation doesn't take place. However, syntax-level analysis does not gather enough information for most checks.

The easiest way to understand how different kinds of analyses compare to each other is to see how program is represented from the point of view of each kind of analysis.

As an example, let us construct its *abstract syntax tree*, *control flow graph*, and path-sensitive *exploded graph* for a simple function `foo(...)`, which we put into a file called `test.c` for further reference:

```
1 void foo(int x) {
2   int y, z;
3   if (x == 0)
4     y = 5;
5   if (!x)
6     z = 6;
7 }
```

test.c

2.1. Abstract syntax tree

Clang *abstract syntax tree* (AST) is the structure produced by the *compiler* frontend and serves as the *intermediate representation* of the program used by Clang. Binary code generation takes place based on AST. Unlike AST of the GCC C/C++ compiler, Clang AST contains not only the minimal information necessary to compile the program correctly, but also complete information about the program source code: each element of the tree remembers its source locations and how exactly it was written in the source, even before preprocessing took place. This makes the AST itself usable as the easiest framework for source-level analysis.

Below is a command-line dump of the abstract syntax tree for `test.c`:

```
~ $ clang -cc1 -ast-dump test.c
TranslationUnitDecl <<invalid sloc>> <invalid sloc>
`-FunctionDecl <test.c:1:1, line:7:1> line:1:6 foo 'void (int)'
  |-ParmVarDecl 0x3625c60 <col:10, col:14> col:14 used x 'int'
  `-CompoundStmt <col:17, line:7:1>
    |-DeclStmt <line:2:3, col:11>
    | |-VarDecl 0x3625de0 <col:3, col:7> col:7 used y 'int'
    | `-VarDecl 0x3625e50 <col:3, col:10> col:10 used z 'int'
    |-IfStmt <line:3:3, line:4:9>
    | |-<<<NULL>>>
    | |-BinaryOperator <line:3:7, col:12> 'int' '=='
    | | |-ImplicitCastExpr <col:7> 'int' <LValueToRValue>
    | | | `-DeclRefExpr <col:7> 'int' lvalue ParmVar 0x3625c60 'x' 'int'
    | | `-IntegerLiteral <col:12> 'int' 0
    | |-BinaryOperator <line:4:5, col:9> 'int' '='
    | | |-DeclRefExpr <col:5> 'int' lvalue Var 0x3625de0 'y' 'int'
    | | `-IntegerLiteral <col:9> 'int' 5
    | `-<<<NULL>>>
    `-IfStmt <line:5:3, line:6:9>
      |-<<<NULL>>>
      |-UnaryOperator <line:5:7, col:8> 'int' prefix '!'
      | `-ImplicitCastExpr <col:8> 'int' <LValueToRValue>
      |   `-DeclRefExpr <col:8> 'int' lvalue ParmVar 0x3625c60 'x' 'int'
      |-BinaryOperator <line:6:5, col:9> 'int' '='
      | |-DeclRefExpr <col:5> 'int' lvalue Var 0x3625e50 'z' 'int'
      | `-IntegerLiteral <col:9> 'int' 6
      `-<<<NULL>>>
```

Reading the AST is similar to reading the original program, annotated to display the semantics that weren't necessarily instantly obvious from the raw source code. For instance, you may understand that variable `x` on line 3, on which the `if` statement argument depends, is actually a parameter of `foo(...)`, while variable `y` referenced on line 4 is a local variable declared on line 2 together with `z`.

However, while constructing the AST, the compiler does not try to understand and model what exactly is going on in the program. It does not construct different execution paths through the branch statements, or try to predict how different branches interact.

The best thing you can do with AST is to detect unwanted *code patterns*. For example, you may want to avoid C-style casts in your C++ project, you can easily create an AST-based checker that warns on all C-style casts. A bit more complicated example would be ensuring that return value of some function is always checked for error.

The `security.InsecureAPI` family of checkers may serve as a good example of AST-based checkers in the default distribution of CSA.

However, if you try to catch, say, divisions by zero with an AST-based checker, even if it might be easy to find code patterns like "`y = x / 0`", but it would be much harder to find code like "`z = 0; ...; y = x / z;`". For such checks, a more powerful approach is necessary.

2.2. Control flow graph

Clang *control flow graph* (CFG) is a representation, using graph notation, of all paths that may seem to be possibly traversed through a program during its execution. CFG is constructed separately for every function body. Each node of the CFG represents a *basic block* of statements that do not contain any branch statements, and are therefore executed sequentially. Each basic block ends with a *terminator statement*, which

is a branch statement or a return from the function. Outgoing edges connect the basic block to other blocks that may be reached depending on the run-time value of the terminator branch condition.

CSA provides an easy way of dumping the control flow graph:

```
~ $ clang -cc1 -analyze -analyzer-checker=debug.ViewCFG test.c
Writing '/tmp/CFG-02fc89.dot'...  done.
Running 'xdot.py' program... done.
```

Figure 1 shows the control flow graph for `test.c`, simplified for easier reading.

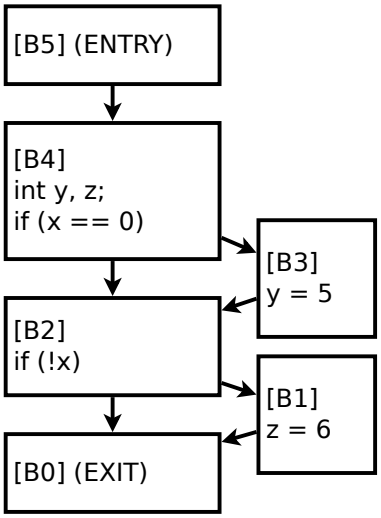

Figure 1: Simplified control flow graph for `test.c`.

CFG-based analysis is useful for creating safe checks, for which it is necessary to consider all possible program paths. For example, if you want to ensure that a certain branch condition always evaluates to `false`, and thus the code below it is "dead", then you'd probably have no choice but to *reach definitions* of all variables referenced inside the condition expression, and CFG would be the right tool for such analysis.

The CFG is constructed relatively easily from the AST. However, CFG-based analysis is often difficult to implement, because CFG does not instantly provide the data flow analysis; additional coding is required to achieve that. Clang framework provides some ready-made CFG-based solutions for checkers "out of the box", such as `LivenessAnalysis`.

The `deadcode.DeadStores` checker is a good example of a CFG-based checker in the default distribution of CSA.

Sometimes CFG-based analysis is used in combination with path-sensitive analysis, when path-sensitive part of the checker is used to find a potential defect location, and later a CFG-based heuristic is implemented in order to improve true positive rate by inspecting other paths to or from the defect. The official `deadcode.UnreachableCode` checker is an example of combining path-sensitive and CFG-based analysis.

However, an attentive reader would instantly find a flaw on Figure 1 that makes CFG-based analysis less useful. By simply looking at the CFG, you would not be able to figure out that once `true` branch is taken in basic block `[B4]`, `true` branch is also inevitably taken in basic block `[B2]`, and vice versa, because branching conditions before these blocks are related. In fact, depending on the initial value of `x`, there are only two ways of reaching the exit block `[B0]` from the entry block `[B5]`: the program either goes through both `[B3]` and `[B1]`, as soon as `x == 0`, or goes through none of them otherwise. This limitation significantly reduces the efficiency of CFG-based analysis.

2.3. Exploded graph

CSA *exploded graph* is the basic data structure of the path-sensitive static analyzer engine. Analyzer core tries to "interpret" the program code, and treats different paths through the CFG, even if they pass through same statements or basic blocks, separately, hence the term "exploded". Exploded graph consists of all paths through the CFG that were explored by the analyzer engine, and carries information regarding the program state on each path in every statement. Nodes of the graph, referred to as *exploded nodes*, are pairs composed of the state of the program and the program point currently being analyzed.

You can display the complete exploded graph for every analysis pass by turning on the special checker called `debug.ViewExplodedGraph`:

```
~ $ clang -cc1 -analyze -analyzer-checker=debug.ViewExplodedGraph test.c
Writing '/tmp/ExprEngine-0528e9.dot'...  done.
Running 'xdot.py' program... done.
```

Exploded graphs are often very large. Exploded graph of `test.c` generated by CSA has over 50 nodes, and is too large to include into this document. Still, figure 2 should give you a rough idea of how it essentially looks.

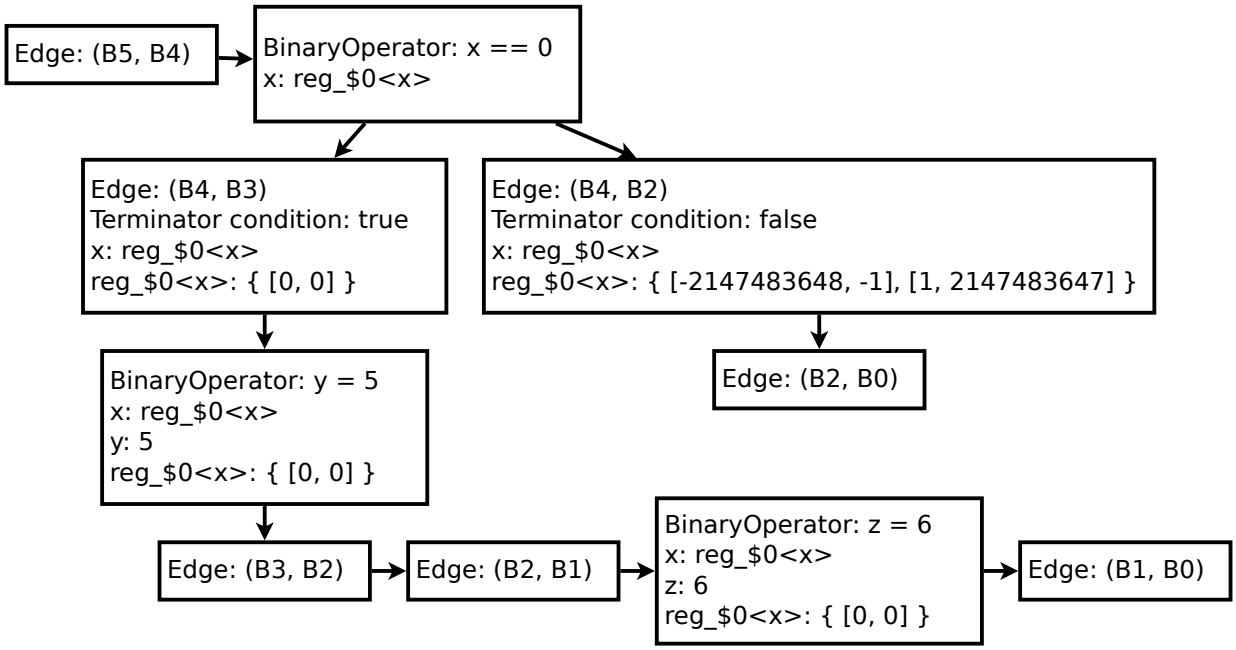

Figure 2: Extremely simplified exploded graph for `test.c`.

Let us see how path-sensitive analysis goes on inside `test.c`. The analyzer starts with emulating the first operation, namely the comparison operator `x == 0`. Because value of `x` is unknown at this point of the analysis (and, in fact, will never be known), this value is represented as a *symbol* `reg_$0<x>`. This symbol is to be understood as "the value stored at the memory region of parameter variable `x` at the beginning of the analysis".

Once the comparison statement is emulated, we reach the terminator of our CFG, namely, the `if` statement. Depending on the terminator condition, we jump to another CFG block, either `[B3]` or `[B2]`. Because we are unsure what branch we take, we split the exploded graph into the two possible paths. On each path, the new node is created by *assuming* that symbol `reg_$0<x>` takes values from a certain *range*: on the `true` branch, it is assumed to be an integer in range $[0, 0]$ (actually, equal to 0), and on the `false` branch, it is assumed to belong to $[-21417483648, -1] \cup [1, 21417483647]$. The range assumed on the symbols would stay inside all

nodes branching off the current node unless the symbol itself is no longer referenced anywhere in the node and gets *garbage-collected*.

On `[B3]`, we execute an assignment operator `y = 5`. This is instantly represented in the node as a *binding*: the value of variable `y` is a *concrete* (non-symbolic) value 5. Then we jump to `[B2]` anyway. However, we are reaching `[B2]` in a different state, hence it is represented by a different node in the exploded graph.

Now, upon reaching `[B2]`, note how we no longer assume anything or try to guess what branch we take from there. Because range of symbol `reg_$0<x>` is still stored inside the node, we already know the truth value of `!x`.

On the `true` branch, assignment `z = 6` is executed. Note how the binding `y = 5` is no longer present in the program state: it was garbage-collected because variable `y` is no longer referenced in further code. Finally, all branches reach the end block `[B0]`, and the analysis stops.

Information stored in the exploded graph is exhaustive, and contains the best assumptions the analyzer core can make about the program execution. Moreover, unlike AST- and CFG-based analysis, path-sensitive CSA checkers rarely read the exploded graph passively, but instead actively participate in its construction, adding their own nodes, bindings, assumptions, leaving checker-specific marks, and splitting paths in the exploded graph at their own will.

2.4. Further reading

Coding AST- or CFG-based checkers is discussed in detail in section 3. If you are interested in coding path-sensitive checkers, jump directly to chapter 4; however, some knowledge of the Clang AST may still be useful. There is a highly recommended introduction to Clang AST available at the official Clang website[5].

[5] http://clang.llvm.org/docs/IntroductionToTheClangAST.html

3. AST-based checkers

Many simple checks can be implemented by looking at the syntax tree of the program and catching unwanted code patterns. Checkers that do not make use of the CSA path-sensitive engine are fast and often have good true positive rate, but are capable of catching only a very limited set of defects.

In this section we proceed to discuss two common technologies for creating syntax-only checkers: AST visitors and AST matchers. Usually having a good command of one of these technologies is sufficient, but sometimes you may want to use both matchers and visitors in the same checker.

AST-based checks are not "the" strength of the Clang Static Analyzer — if you are using only AST-based information, then you could have done this check with any other Clang-based tool. In the official Clang Static Analyzer, there are a few AST-based checks, but normally AST-based checks go to the `clang-tidy` tool.

On the other hand, it is not uncommon to use AST-based checks from within the path-sensitive engine, so that to have a better idea of the syntax behind the path-sensitive analysis events. So, even though in this section we shall deliberately learn how to write AST-only checks, exactly same techniques may make it into a path-sensitive checker, and it is useful to have a good command of them.

3.1. Path-insensitive checker callbacks

Path-sensitive engine of CSA starts working only as soon as at least one checker subscribes to a checker callback that requires path-sensitive analysis to fire. If you are interested only in AST-based checkers, and disable all path-sensitive checkers, the analysis would run significantly faster.

Because AST-based checkers do not participate in construction of the data structures they analyze, only a few AST-only callbacks are defined. The most useful callbacks that do not instantly trigger the path-sensitive engine are `check::EndOfTranslationUnit` and `check::ASTCodeBody`.

3.1.1. check::EndOfTranslationUnit

```
void checkEndOfTranslationUnit(const TranslationUnitDecl *TU,
                               AnalysisManager &AM, BugReporter &BR) const;
```

In this callback, the complete AST of the program is available for analysis.

The entry point for visiting the AST — the declaration of the whole translation unit — is provided as the first argument, `TU`. This callback is commonly used when not only executable code, but also declarations needs to be checked.

3.1.2. check::ASTCodeBody

```
void checkASTCodeBody(const Decl *D,
                      AnalysisManager &AM, BugReporter &BR) const;
```

In this callback, a declaration of a function, code body of which the analyzer would normally analyze, would be provided on every call.

The body of the function that needs to be analyzed is available as `D->getBody()`. This callback is convenient when only executable code needs to be analyzed.

3.1.3. check::ASTDecl<T>

```
void checkASTDecl(const T *D, AnalysisManager &Mgr, BugReporter &BR) const;
```

This callback is called for all AST declarations of type `T` (for example, for all variables if `T` is `VarDecl` or for all class fields if `T` is `FieldDecl`). This is often a convenient simple alternative for declaration visitors.

3.2. AST visitors

The AST visitor mechanism is the most flexible tool for exploring the Clang AST. Clang provides numerous visitors for the AST, with similar syntax. For implementing checkers, two kinds of visitors are mostly useful:

— `ConstStmtVisitor` is widely used for checking code bodies, which is most often exactly what you need,

— `ConstDeclVisitor` is sometimes used for checking declarations outside code bodies (such as global variables).

In order to use a visitor, you need to inherit a class from it and implement visitor callbacks for different kinds of AST nodes. Whenever a callback is not implemented for a particular node, a callback for a more generic node would be called anyway: so, for example, a `CXXOperatorCallExpr` would be visited in one of the following callbacks:

— `VisitCXXOperatorCallExpr(...)`,

— `VisitCallExpr(...)`,

— `VisitExpr(...)`,

— `VisitStmt(...)`,

whichever turns out to be the first one to be defined.

3.2.1. Implementing a simple statement visitor

As an example, let us see if we can rewrite `alpha.core.MainCallChecker` described in section 1 as an AST visitor. This checker would also serve as an example of using the `check::ASTCodeBody` callback.

First, let us declare an AST visitor. The visitor stores references to the `BugReporter` object in order to throw path-insensitive reports, and also the current `AnalysisDeclContext`, which is required for producing diagnostic locations for bug reports. The latter also wraps the original function we are analyzing.

```
namespace {
class WalkAST : public ConstStmtVisitor<WalkAST> {
  BugReporter &BR;
  AnalysisDeclContext *ADC;

  void VisitChildren(const Stmt *S);

public:
  WalkAST(BugReporter &Reporter, AnalysisDeclContext *Context)
      : BR(Reporter), ADC(Context) {}
  void VisitStmt(const Stmt *S);
  void VisitCallExpr(const CallExpr *CE);
};
}
```

The visitor defines two public callbacks: `VisitCallExpr(...)` for special handling of function call expressions, and `VisitStmt(...)` for visiting all other kinds of statements.

These callbacks have one thing in common: they need to visit sub-statements whenever they're done visiting their statement. This operation is often separated into a sub-function called `VisitChildren(...)`:

```
void WalkAST::VisitChildren(const Stmt *S) {
  for (Stmt::const_child_iterator I = S->child_begin(), E = S->child_end();
       I != E; ++I)
    if (const Stmt *Child = *I)
      Visit(Child);
}
```

Now, `VisitStmt(...)` doesn't really need to do anything else:

```
void WalkAST::VisitStmt(const Stmt *S) {
  VisitChildren(S);
}
```

Most of the checker logic is stored in written out in `VisitCallExpr(...)`. We obtain the function declaration for the current call expression, take its identifier, and see if this identifier coincides with `"main"`. If it does, we throw a path-insensitive ("basic") report. Note that unlike path-sensitive checkers, syntax-only checkers do not have the convenient `CheckerContext` wrapper available, so they need to access the `BugReporter` object directly, and also put some effort in obtaining the necessary source locations.

```
void WalkAST::VisitCallExpr(const CallExpr *CE) {
  if (const FunctionDecl *FD = CE->getDirectCallee())
    if (const IdentifierInfo *II = FD->getIdentifier())
      if (II->isStr("main")) {
        SourceRange R = CE->getSourceRange();
        PathDiagnosticLocation ELoc =
            PathDiagnosticLocation::createBegin(CE, BR.getSourceManager(), ADC);
        BR.EmitBasicReport(ADC->getDecl(), "Call to main", "Example checker",
                           "Call to main", ELoc, R);
      }
  VisitChildren(CE);
}
```

That's it for the implementation of the visitor. Now we simply need to create it and give it some code to visit. Since "all code" is a declaration rather than a statement, we subscribe on the `check::ASTCodeBody` callback:

```
namespace {
class MainCallCheckerAST : public Checker<check::ASTCodeBody> {
public:
  void checkASTCodeBody(const Decl *D, AnalysisManager &AM,
                        BugReporter &B) const;
};
}
```

And implement the callback as follows:

```
void MainCallCheckerAST::checkASTCodeBody(const Decl *D, AnalysisManager &AM,
                                          BugReporter &BR) const {
  WalkAST Walker(BR, AM.getAnalysisDeclContext(D));
  Walker.Visit(D->getBody());
}
```

This way the visitor starts from the compound statement that represents the function body, and descends into sub-statements.

Checker is now ready. However, on the example code from chapter 1 it is silent — we're only detecting direct calls now, not calls through function pointers, because only that much is present in the AST. We would warn on a simpler code though:

```
void foo() {
  main(0, 0); // Call to main!
}
```

3.2.2. Merging statement and declaration visitors

Sometimes you'd like to intermix the two visitors together, in order to visit both statements and declarations. In this case, you can inherit your visitor from both visitors:

```
class WalkAST : public ConstStmtVisitor<WalkAST>,
                public ConstDeclVisitor<WalkAST> {
  /* ... */

public:
  using ConstStmtVisitor<WalkAST>::Visit;
  using ConstDeclVisitor<WalkAST>::Visit;

  /* ... */
};
```

3.3. AST matchers

AST matchers are the new API for finding simple code patterns in the Clang AST. They allow writing extremely concise declarative definitions of such patterns — almost as short as describing them in words in a natural language — and provide an interface for taking actions on every pattern found. Being preferable for simple code patterns, for their simplicity and code readability, AST matchers are not as omnipotent as AST visitors.

3.3.1. Implementing a simple AST matcher

As an example, let us see if we can rewrite `alpha.core.MainCallChecker` described in section 1 with the help of AST matchers.

Recall that the checker needs to find calls to functions with the name `"main"`. Knowing just that, we can instantly write a matcher that finds such calls:

```
callExpr(callee(functionDecl(hasName("main")))).bind("call")
```

Which means that complete checker logic now suits into a single line of code! All that remains is to write out the checker bureaucracy and throw the bug report. Note how the `bind(...)` matcher command assigns a name to the AST node it is applied to, for future reference.

The first thing we need to define is the matcher callback. This callback would fire whenever the matcher finds something. Matcher callbacks need to inherit from `MatchFinder::MatchCallback` and implement the method called `run(...)`:

```
namespace {
class Callback : public MatchFinder::MatchCallback {
  BugReporter &BR;
  AnalysisDeclContext *ADC;

public:
  void run(const MatchFinder::MatchResult &Result);
  Callback(BugReporter &Reporter, AnalysisDeclContext *Context)
      : BR(Reporter), ADC(Context) {}
};
}
```

Ideally, the only thing match callback needs to do is throw the basic bug report. This is the case here. However, sometimes matchers cannot cover the whole checker logic, and it is natural to leave some final checks to the callback.

```
void Callback::run(const MatchFinder::MatchResult &Result) {
  const CallExpr *CE = Result.Nodes.getStmtAs<CallExpr>("call");
  assert(CE);
  SourceRange R = CE->getSourceRange();
  PathDiagnosticLocation ELoc =
      PathDiagnosticLocation::createBegin(CE, BR.getSourceManager(), ADC);
  BR.EmitBasicReport(ADC->getDecl(), "Call to main", "Example checker",
                     "Call to main", ELoc, R);
}
```

In the callback, we obtain the call expression by its name, `"call"`, defined via `bind(...)`. Looking at the matcher, we are sure that such call expression is present, so we can assert that we have successfully obtained the statement by name.

Now it is time to define the checker. This time, let us try out the whole-translation-unit matching:

```
namespace {
class MainCallCheckerMatchers : public Checker<check::EndOfTranslationUnit> {
public:
  void checkEndOfTranslationUnit(const TranslationUnitDecl *TU,
                                 AnalysisManager &AM, BugReporter &B) const;
};
}
```

Finally, in the checker callback, we need to construct our matcher and use it to find bugs:

```
void MainCallCheckerMatchers::checkEndOfTranslationUnit(
    const TranslationUnitDecl *TU, AnalysisManager &AM, BugReporter &B) const {
  MatchFinder F;
  Callback CB(B, AM.getAnalysisDeclContext(TU));
  F.addMatcher(
      stmt(hasDescendant(
          callExpr(callee(functionDecl(hasName("main")))).bind("call"))),
      &CB);
  F.matchAST(AM.getASTContext());
}
```

The `matchAST(...)` method of `MatchFinder` lets it match the whole AST of the translation unit. The checker is now done. The output is similar to the visitor version of the checker.

The `ASTContext` structure, which we obtained from the `AnalysisManager`, contains the whole AST of the program, and also various meta-information regarding the AST, such as implementation-specific traits imposed during compilation.

3.3.2. Re-using matchers

If a certain sub-pattern repeats multiple times in your matcher, you can store and re-use it. In the example below, matcher `TypeM` is stored and then re-used twice in two other matchers, which are in turn stored for later use:

```
1 TypeMatcher TypeM = templateSpecializationType().bind("type");
2 DeclarationMatcher VarDeclM = varDecl(hasType(TypeM)).bind("decl");
3 StatementMatcher TempObjM = temporaryObjectExpr(hasType(TypeM)).bind("stmt");
```

3.3.3. Defining custom matchers

Sometimes combining the predefined matchers is not enough to implement the desired check. In this case, it is often convenient to implement a custom AST matcher. Implementing AST matchers is a matter of a few lines of code, and many examples can be found in `ASTMatchers.h`. When implementing a custom AST matcher inside the checker, you need to put it into `clang::ast_matchers` namespace. The example below defines a custom declaration matcher that matches `RecordDecl` nodes that declare unions rather than structures:

```
1 namespace clang {
2 namespace ast_matchers {
3
4 AST_MATCHER(RecordDecl, isUnion) {
5   return Node.isUnion();
6 }
7
8 } // end namespace clang
9 } // end namespace ast_matchers
```

3.3.4. Matching particular statements

As we mentioned before, the `matchAST(...)` method of `MatchFinder` matches the whole AST of the translation unit. Sometimes you want to match only a particular section of the AST. In this case, you can use the `match(...)` method.

For instance, let us try to implement `MainCallChecker` using `check::ASTCodeBody`. Then we need to match `D->getBody()` with the `MatchFinder`.

However, the semantics of `match(...)` is different from semantics of `matchAST(...)`: the former tries to match the statement itself, the latter tries to match its sub-statements as well. So we need to modify our matcher to make it look for sub-statements manually:

```
1  void MainCallCheckerMatchers::checkASTCodeBody(const Decl *D,
2                                                 AnalysisManager &AM,
3                                                 BugReporter &BR) const {
4    MatchFinder F;
5    Callback CB(BR, AM.getAnalysisDeclContext(D));
6    F.addMatcher(
7        stmt(hasDescendant(
8            callExpr(callee(functionDecl(hasName("main")))).bind("call"))),
9        &CB);
10   F.matchAST(*(D->getBody()), AM.getASTContext());
11 }
```

3.4. Constant folding

It is often not obvious from the AST of an expression that this expression actually represents a constant value. This expression may contain casts and references to constant variables, and folding it to the actual value is often non-trivial. There is a ready-made solution in Clang for this problem — just use the `Expr`'s `EvaluateAsInt(...)` method:

```
const Expr *E = /* some AST expression you are interested in */
llvm::APSInt Result;
if (E->EvaluateAsInt(Result, ACtx, Expr::SE_AllowSideEffects)) {
  /* we managed to obtain the value of the expression */
  uint64_t IntResult = Result.getLimitedValue();
  /* ... */
} else {
  /* the expression doesn't fold to into a constant value */
}
```

3.5. Further reading

An introduction to the Clang AST was given on LLVM developer meeting by Manuel Klimek; a video is available![6]

A comprehensive AST matcher reference is available on the official Clang website [7].

While writing AST-based checkers, you would most likely want to consult the official documentation for the various AST nodes: statement nodes inheriting from `Stmt`[8], declaration nodes inheriting from `Decl`[9], and type nodes inheriting from `Type`[10].

[6] http://llvm.org/devmtg/2013-04/videos/klimek-vhres.mov
[7] http://clang.llvm.org/docs/LibASTMatchersReference.html
[8] http://clang.llvm.org/doxygen/classclang_1_1Stmt.html
[9] http://clang.llvm.org/doxygen/classclang_1_1Decl.html
[10] http://clang.llvm.org/doxygen/classclang_1_1Type.html

4. Path-sensitive analysis

Clang Static Analyzer, by design, implements a static analysis method known as *symbolic execution*. This method is based on *abstract interpretation* of the program, and assumes assigning *symbolic values* to program variables, and splitting all possible states of the program into classes of states leading the program across the same paths.

Classes of such *program states* are often defined by *range constraints* imposed on symbolic values involved. However, they may also be different in other ways, or even contain checker-specific differences.

The analyzer also implements a memory model that allows remembering concrete and symbolic values of particular *memory regions* and accessing them at any time during analysis.

The path-sensitive engine of CSA supports *interprocedural analysis*. This means that whenever the analyzer encounters a function call, it tries to model the call and descend into sub-function to continue analysis.

4.1. Obtaining information from the program state

The `ProgramState` is one of the basic structures in path-sensitive analysis. It holds complete information on a momentary state of the program under analysis. By looking into the program state, you can obtain symbolic values of variables stored in memory regions and expressions defined in the current *location context*.

`ProgramState` is *immutable*. Once a `ProgramState` object was created, you cannot modify it; you can only create a new `ProgramState` object that differs from the original `ProgramState` in a certain sense. Also, you never have to access `ProgramState` objects directly, or manage their lifetime manually; they are always wrapped into reference-counting smart pointers called `ProgramStateRef`.

In most path-sensitive checker callbacks, you have a `CheckerContext` object available. One thing it carries is the current program state, which you can easily obtain, for example:

```
void checkEndFunction(CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  /* ... */
}
```

Program state consists of the following traits of the program:

— “Environment”: symbolic values of active expressions;

— “Region Store”: symbolic values of memory regions;

— “Range Constraints”: ranges that symbolic values may take;

— “Taint”: a registry of symbolic values obtained from insecure sources;

— “Generic Data Map”: checker-specific information.

4.1.1. Obtaining values of expressions

The analyzer remembers symbolic values for all expressions it currently needs. Whenever an expression leaves the current context, it is garbage-collected and no longer available. The mapping from expressions to their symbolic values is called the *environment*. You can always obtain the value of an expression if it is available in the environment:

```
const Expr *E = /* some AST expression you are interested in */;
const LocationContext *LC = C.getLocationContext();
SVal Val = State->getSVal(E, LC);
```

If expression `E` is available in the environment, `Val` would be its symbolic value. If `E` is not in the current environment, an `UnknownVal` would be returned. The environment would not try to compute the value for any AST expression; it would only return the value if it is already there. A few things you can surely find in the environment include values of sub-expressions before analyzing the whole expression, and also the value of any expression right after it was analyzed, which is enough for most practical purposes.

4.1.2. A brief introduction to memory regions

Memory regions are symbolic l-values. They may appear during analysis by obtaining symbolic values of pointers:

```
const Expr *E = /* an pointer expression */;
const MemRegion *Reg = State->getSVal(E, LC).getAsRegion();
```

Memory regions can also be obtained directly with declarations of variables:

```
const VarDecl *D = /* a declaration of a variable */;
const MemRegion *Reg = State->getLValue(D, LC).getAsRegion();
```

In both cases, `getAsRegion()` would return a null pointer if the value obtained does not represent any memory region.

Memory regions can contain symbolic values inside them; obtaining such values may be though of as dereferencing memory regions as pointers. The mechanism for dereferencing memory regions is called the *region store*. Each `ProgramState` contains an instance of the store, which carries known bindings of symbolic values to memory regions.

Obtaining the region binding from the program state is as simple as:

```
SVal Val = State->getSVal(Reg);
```

Unlike the environment, the region store tries to produce a sensible binding even if there is no direct binding already available in the current store. In such cases, it would construct and return a symbol representing the unknown value of the region. This means that you can always rely on `getSVal(const MemRegion *)` to produce a sensible symbolic value.

4.1.3. Iterating over region store bindings

The most common operation you usually use the region store for is obtaining bindings for particular memory regions. However, sometimes you may want to list (or, generally speaking, iterate over) all explicit bindings in the store. In the `StoreManager` class, which maintains ownership of the region store instances, there is a mechanism for iterating over bindings in a particular program state, known as the `BindingsHandler`.

In order to use `BindingsHandler`, you need to inherit from it:

```
class Callback : public StoreManager::BindingsHandler {
public:
  bool HandleBinding(StoreManager &SM, Store St,
                     const MemRegion *Region, SVal Val) {
    /* ... */
  }
};
```

The callback should return `false` whenever it needs to stop iterating. Once the callback is defined, you can start iterating:

```
Callback CB;
StoreManager &SM = C.getStoreManager();
SM.iterBindings(State->getStore(), CB);
```

4.1.4. Assumptions on symbolic values

With a program state, you can take any symbolic value and assume a boolean condition on it: whether this value would represent a boolean `true` or a boolean `false`. In order to test an assumption, the value needs to be either defined or unknown; undefined values cannot be tested. Once you are sure that the value is not undefined, you can use the `assume(...)` method of the program state:

```
SVal Val = /* a certain symbolic value */;
Optional<DefinedOrUnknownSVal> DVal = Val.getAs<DefinedOrUnknownSVal>();
if (!DVal)
  return;
if (State->assume(*DVal, true)) {
  /* things to do if Val can possibly be true */
}
if (State->assume(*DVal, false)) {
  /* things to do if Val can possibly be false */
}
```

Note that both statements can actually fire, if the value is not known to be certainly true or certainly false.

4.1.5. Operations on symbolic values

Suppose you have symbolic values `A` and `B`, and you want to assume that `A` is greater than `B`. In order to do that, you need to represent "`A > B`" as a new symbolic value `C` (of boolean type). You can create new symbolic values with a special class called `SValBuilder`:

```
SVal A = /* a certain symbolic value */;
SVal B = /* the other symbolic value */;
ASTContext &ACtx = C.getASTContext();
SValBuilder &SVB = C.getSValBuilder();
SVal C = SVB.evalBinOp(State, BO_GT, A, B, ACtx.BoolTy);
```

4.1.6. Using the taint analysis

A symbolic value is said to be *tainted* if it is known to have been obtained from an untrusted source, such as by reading standard input or file descriptor, or from environment variables. Taint analysis is an efficient method for finding security defects, such as SQL injections, based on detecting usage of tainted values in sensitive function calls.

You can always find out if a certain symbolic value is tainted in a certain program state:

```
ProgramStateRef State = C.getState();
SVal Val = /* a certain symbolic value */;
if (State->isTainted(Val)) {
  /* ... */
}
```

Most of the taint information originates from default built-in CSA checkers, most notably from the checker called `alpha.security.taint.TaintPropagation`. On defining your own sources of taint for your checker and otherwise expanding taint analysis, see 4.2.4.

4.2. Mutating and splitting the program state

Path-sensitive checkers not only observe the symbolic execution of the program by the analyzer core, but also actively participate in modeling the program behavior. A checker may add its own traits to the program state, modify region store bindings or range constraints, or split the state, implying that a certain operation may have multiple distinct results that would eventually make the program take different execution paths.

Note that splitting program states should be done with care. Each program state split effectively doubles the amount of work the analyzer needs to perform for the rest of the current `ExplodedGraph` sub-tree. A large enough amount of splits can quickly degrade the analysis speed.

4.2.1. Adding transitions to the exploded graph

As explained in subsection 2.3, the path-sensitive analyzer engine represents the flow of the analysis in the form of a graph, called the *exploded graph* of the analysis. Nodes of this graph, known as the *exploded nodes*, are defined as ordered pairs that consist of a program point (a single element of the CFG is represented with one program point, or more than one if technically necessary), and a program state. Each statement of the program brings us from one existing node to another newly created node, or probably into multiple other nodes, if there are multiple things that may happen in this statement.

In fact, the exploded graph is not necessarily a tree; it may contain cycles, whenever the program reaches the same program point with the same program state; in this case the analysis of the branch effectively stops, most likely indicating an infinite loop (or, more likely, a bug in one of the checkers).

You cannot modify existing nodes, program points, or program states; they are *immutable*. What you can do, however, is produce a new program state or a new auxiliary program point (or both), and make use of the `CheckerContext` object to *add a transition* to this new state or new point.

Code that changes a single aspect of a program state usually looks as:

```
ProgramStateRef State = C.getState();
State = modifyState(State); // do stuff
C.addTransition(State);
```

If you want to add parallel transitions to multiple alternative nodes, you would probably do something like:

```
ProgramStateRef State = C.getState();
ProgramStateRef State1 = modifyState1(State); // do stuff
ProgramStateRef State2 = modifyState2(State); // do other stuff
C.addTransition(State1);
C.addTransition(State2);
```

Sometimes you want to make a single sequence of transitions, rather than multiple parallel independent branches. In this case, you can use the overridden method that accepts a predecessor node:

```
ProgramStateRef State = C.getState();
State = modifyState1(State); // do stuff
ExplodedNode *N = C.addTransition(State);
State = modifyState2(State, N); // do other stuff
C.addTransition(State2, N);
```

Transitions added in these three code snippets are visualized as the respective graphs on figure 3.

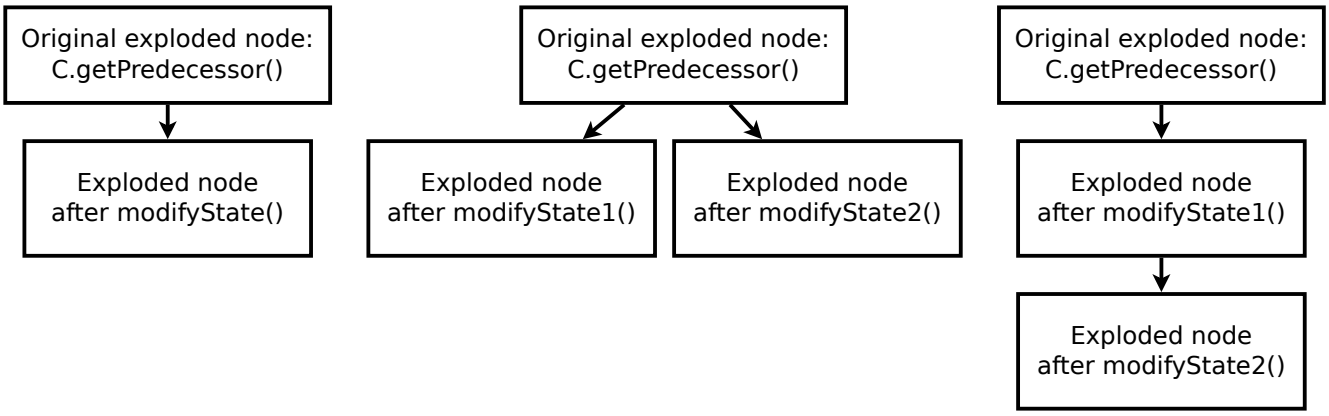

Figure 3: Adding transitions to the exploded graph.

Now let us discuss different mechanisms of mutating the program state available for use in the checkers.

4.2.2. Splitting the state on range constraint assumptions

The `assume(...)` method of `ProgramState` discussed in 4.1.4 operates by returning a new program state with the assumption imposed, or a null `ProgramStateRef` if the assumption cannot be satisfied (because

it contradicts other assumptions already imposed). Thus, what you can do is not only cast it to bool to see if the assumption can be satisfied, but also transition to the newly created state.

For example, if your checker needs to inform the analyzer that a certain function cannot return 0, you can assume its symbolic return value to be non-zero, and add a transition to the assumed state on post-call of this function:

```
SVal Val = Call.getReturnValue();
Optional<DefinedOrUnknownSVal> DVal = Val.getAs<DefinedOrUnknownSVal();
if (!DVal)
  return;
ProgramStateRef State = C.getState();
State = State->assume(*DVal, true);
C.addTransition(State);
```

After such transition, the analyzer would know that this value is non-zero on the whole remaining execution path.

Sometimes you may want to add parallel transitions to both `true` and `false` branches. What is it good for? In fact. the whole idea of symbolic execution is about splitting states. This way you do the same thing that an analyzer does on encountering an `if` statement: instead of considering a single branch on which nothing is known about the symbol, you consider two branches, on each of which something is known. It means that if a checker, in order to report its defect, needs to know exactly that, say, the value is zero, the checker would be able to find such defect on one of the branches. Without a state split, such checker would stay silent, being unable to find a program path on which the defect is certain to exist.

4.2.3. Creating region store bindings

Sometimes you may want to modify the program state by binding a symbolic value to a location. A typical use case would be to manually emulate a function call that the analyzer is unable to model; for example, source code of a certain function is not available, but you can still put your understanding of its specification into the checker and try to emulate its behavior. And then, if you know that the function would write a certain symbolic value into a certain location, you can tell the checker to model it:

```
ProgramStateRef State = C.getState();
SVal Loc = /* Obtain a location */;
SVal Val = /* Obtain a value */;
State = State->bindLoc(Loc, Val);
C.addTransition(State);
```

4.2.4. Expanding the taint analysis

In order to make taint analysis efficient, the analyzer needs to know which events *produce* tainted values, and how taint *propagates* through different events to other symbolic values. Both of these tasks can be extended with the help of checkers.

In order to add sources of taint, subscribe to any checker callback suitable for catching the desired event, and use the `addTaint(...)` method of the `ProgramState`. This method has three overrides, that allow adding taint to different kind of symbolic values.

Adding taint on expressions in the current environment:

```
LocationContext *LC = C.getLocationContext();
ProgramStateRef State = C.getState();
const Expr *E = /* Obtain an expression value of which is untrusted */;
ProgramStateRef NewState = State->addTaint(E, LC);
if (NewState != State) // avoid loops in the exploded graph
  C.addTransition(NewState);
```

Tainting a numeric value:

```
ProgramStateRef State = C.getState();
SVal V = /* Obtain a numeric symbol from an untrusted source */;
if (SymbolRef Sym = V.getAsSymbol()) {
  ProgramStateRef NewState = State->addTaint(Sym);
  if (NewState != State)
    C.addTransition(NewState);
}
```

Adding taint to a pointer to an untrusted data:

```
ProgramStateRef State = C.getState();
SVal V = /* Obtain a symbolic location from an untrusted source */;
ProgramStateRef NewState = State->addTaint(V.getAsRegion());
if (NewState != State)
  C.addTransition(NewState);
```

Note that you cannot mark concrete values as tainted. For example, a symbolic value that represents 32-bit signed integer “0”, cannot be marked as tainted; in fact, the analyzer doesn’t even discriminate between different instances of “0”.

By default, the if a certain symbolic value is marked as tainted, then results of arithmetic operations over it are also marked as tainted. If a region is tainted, then all values derived from it are also tainted; however, if an unrelated value is written into a tainted region, such value is of course no longer considered to be tainted. Also, a region of an element of an array with a tainted symbolic element index is automatically tainted.

However, more complex things may happen to the tainted values. For example, it may be passed into a function that returns another symbolic value, and probably the function cannot be modeled by the analyzer core. In such cases, you need to implement taint propagation: catch the event through which you want the taint to propagate, see if the relevant value is tainted, and then add the taint to the values you want the taint to propagate to.

As mentioned in 4.1.6, most common sources of taint and propagation methods are already defined in the `alpha.security.taint.TaintPropagation` checker, which also contains many examples of working with taint. It is natural, however, for your checker to add your own domain-specific taint sources and propagation methods.

See subsection 5.6 for which exactly kinds of symbolic values can carry taint, and why.

4.2.5. Using program state traits

Checkers are allowed to add their own custom traits to the program state. These traits are stored in a special structure inside the program state, known as the *generic data map* (GDM).

One of the primary use cases for custom program traits is detecting errors that are defined as *sequences of events* rather than one-time events. One of the very obvious examples of such errors is "double-free" kind of errors, which arises when object is destroyed twice, but every single destruction of an object is not yet an defect. In order to create a checker that handles such cases, the common practice is to create a map from symbolic identifiers of objects to their state as seen by the checker (unknown, live, deleted), and store such map within the GDM. Then, the check that would ultimately report the error would be defined as "an object is being destroyed in a program state in which it is marked as deleted".

A typical mistake made by inexperienced authors of CSA checkers is to store such map as a field inside the checker class. This is the very reason why all checker callbacks are `const`-qualified functions: there are very few cases when you need to store anything in the checker state, most of the time *checkers are stateless*. Along the analysis, it is natural for the engine to jump from one branch to another. However, for example, if an object is marked as deleted on one of the program execution branches, it doesn't mean it is deleted on other branches. Which means that member variables of the checker, which are the same for all branches, are not the suitable place for storing information related to the program state; only the program state itself is.

Like the program state itself, GDM is immutable. Which is why, in order to store different data structures inside the GDM, you need to use LLVM immutable containers: `llvm::ImmutableList`, `llvm::ImmutableSet`, `llvm::ImmutableMap`, otherwise performance would suffer significantly, as the program state is copied many times during the analysis.

In order to inject a new trait into the program state, you need to use one of the four predefined macros in the global scope of your checker code (not inside the namespace, and not in a place accessible from multiple Clang translation units).

```
REGISTER_TRAIT_WITH_PROGRAMSTATE(TraitName, Type)
```

Makes the program state carry a trait of type `Type`. You can access the trait and obtain its value in the current state by calling `State->get<TraitName>()`, or obtain a new state with a modified trait value by calling `State->set<TraitName>(NewValue)`. Also, `TraitNameTy` is now a synonym for `Type`.

```
REGISTER_LIST_WITH_PROGRAMSTATE(ListName, ElementType)
```

Makes the program state carry a trait of type `ListNameTy`, which is an LLVM immutable list of elements of type `ElementType`. Apart from working with the whole list via `get<>()` and `set<>()`, you can also easily append items by calling `State->add<ListName>(NewItem)`, or scan the list for items by calling the `State->contains<ListName>(Item)` method template.

```
REGISTER_SET_WITH_PROGRAMSTATE(SetName, ElementType)
```

Makes the program state carry a trait of type `SetNameTy`, which is an LLVM immutable set of elements of type `ElementType`. The set trait supports `add<>()` and `contains<>()` similarly to the list trait, and you can also remove items from the set (which is too heavy of an operation for an immutable list) and obtain a new program state with these items removed from the set by calling `State->remove<SetName>(Element)`.

```
REGISTER_MAP_WITH_PROGRAMSTATE(MapName, KeyType, ValueType)
```

Makes the program state carry a trait of type `MapNameTy`, which is an LLVM immutable map from objects of type `KeyType` to objects of type `ValueType`. This trait supports `remove<>()` by key, it doesn't support `add<>()`, and also `set<>()` and `get<>()` are conveniently overridden: the `State->get<MapName>(Key)` method looks-up the value for the key `Key`, and you can also call the `State->set<MapName>(Key, Value)` method to obtain a new program state with value for `Key` set to `Value`.

If any of the `Type`, `ElementType`, `KeyType`, `ValueType` in these macros is not an integral type, eg. `int` or `bool`, or a pointer type, then there are certain compile-time requirements imposed on these types, necessary for them to qualify as elements of an immutable container. Most importantly, they need to provide a `Profile(...)` method, which allows to use them as LLVM folding-set nodes.

For example, you cannot easily put an `std::string`, or even an `llvm::StringRef`, into an immutable container. You can make a simple wrapper though:

```
class StringWrapper {
  const std::string Str;

public:
  StringWrapper(const std::string &S) : Str(S) {}
  const std::string &get() const { return Str; }
  void Profile(llvm::FoldingSetNodeID &ID) const {
    ID.AddString(Str);
  }
  bool operator==(const StringWrapper &RHS) const { return Str == RHS.Str; }
  bool operator<(const StringWrapper &RHS) const { return Str < RHS.Str; }
};
```

Usage example:

```
REGISTER_SET_WITH_PROGRAMSTATE(MyStringSet, StringWrapper)

void MyChecker::checkPreCall(const CallEvent &Call, CheckerContext &C) {
  ProgramStateRef State = C.getState();
  if (const IdentifierInfo *II = Call.getCalleeIdentifier()) {
    std::string Str = II->getName();
    State = State->add<MyStringSet>(StringWrapper(Str));
    C.addTransition(State);
  }
  if (State->contains<MyStringSet>(StringWrapper("main"))) {
    /* ... */
  }
}
```

Note that `SVal` object provides its own `Profile(...)` method. If you need to store a complex structure, you can implement the `Profile(...)` method by profiling all its fields:

```
void MyStructure::Profile(llvm::FoldingSetNodeID &ID) const {
  ID.AddPointer(Sym);
  Val.Profile(ID);
}
```

4.3. Path-sensitive checker callbacks

Path-sensitive checkers continuously interact with the analyzer core through numerous checker callbacks. Different callbacks fire on different events that happen during analysis.

4.3.1. check::PreStmt<T>

```
void checkPreStmt(const T *S, CheckerContext &C) const;
```

A callback template defined for any AST statement class `T`, that fires every time the analyzer engine *is about to* analyze a statement of class `T`. In this callback, you can obtain values of sub-statements of the statement `T` from the environment.

This callback does not get called on control flow statements (CFG terminators) like `if`. In order to check such statements, subscribe to `check::BranchCondition`.

A typical usage example for this callback can be found in the `core.DivideZero` checker from the default distribution of CSA:

```
1 void DivZeroChecker::checkPreStmt(const BinaryOperator *B,
2                                   CheckerContext &C) const {
3   BinaryOperator::Opcode Op = B->getOpcode();
4   /* ... */
5   SVal Denom = C.getState()->getSVal(B->getRHS(), C.getLocationContext());
6   /* ... */
7 }
```

This checker uses class `BinaryOperator` as template parameter `T`, effectively subscribing on receiving the callback just before every binary operator the analyzer models. Then on line 3 it observes the binary operator's AST to understand the which operation is being modeled, for it is only interested in divisions. Later, on line 5, it obtains the symbolic value for the denominator from the environment.

4.3.2. check::PostStmt<T>

```
void checkPostStmt(const T *S, CheckerContext &C) const;
```

This callback template is similar to `check::PreStmt<T>`, and the only difference is that it fires *after* the statement has been modeled. If `S` is an expression, this callback allows to obtain the symbolic value of `S` itself; however, values of sub-expressions might have been already removed from the environment.

A good example of `check::PostStmt<T>` usage is available in the `unix.Malloc` checker from the default distribution of CSA, which finds memory issues such as leaks or double-free crashes. This checker subscribes on `check::PostStmt<CXXNewExpr>` in order to track symbolic values of pointers allocated with every execution of operator `new` or `new[]`.

4.3.3. check::PreCall

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

This handy callback is simply a more convenient version of `check::PreStmt<CallExpr>`. It fires at exact same moment, before the call is executed, regardless of whether it would be handled with the interprocedural analysis engine or not. The difference is that in `check::PreCall` you possess the `CallEvent` structure from which you can easily obtain symbolic values of the callee, all arguments, and the C++ implicit `this` argument.

In this callback, it is common to try to figure out what function is being called. The easiest way to understand that is to obtain the name of the callee identifier and compare it with the given string. However, string comparison is a heavy operation; it is much faster to store the identifier for the function we want, and then compare identifier pointers.

We have already seen `check::PreCall` used in `alpha.core.MainCallChecker`. You can also find a usage example for `check::PreCall` in `alpha.unix.SimpleStreamChecker` from the default distribution of CSA.

First, let us see how it obtains identifier pointers for faster function lookup:

```
1  void SimpleStreamChecker::initIdentifierInfo(ASTContext &ACtx) const {
2    if (IIfclose)
3      return;
4    IIfclose = &ACtx.Idents.get("fclose");
5  }
```

The `ASTContext.Idents` member variable is the identifier table of the translation unit. You can consult it to find identifiers by string names, store them, and then use for faster lookup.

The implementation of the callback uses `CallEvent` to quickly check if the function being called is the function we are looking for. Then it obtains symbolic value for the argument and makes checks over it:

```
1  void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
2                                         CheckerContext &C) const {
3    initIdentifierInfo(C.getASTContext());
4    if (!Call.isGlobalCFunction())
5      return;
6    if (Call.getCalleeIdentifier() != IIfclose)
7      return;
8    if (Call.getNumArgs() != 1)
9      return;
10   SymbolRef FileDesc = Call.getArgSVal(0).getAsSymbol();
11   /* ... */
12 }
```

4.3.4. check::PostCall

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

Similarly to `check::PreCall`, this is a shortcut callback for `check::PostStmt<CallExpr>`, in which the `CallEvent` structure is available. It fires right after any function call. You can obtain the return value of the call, which is already computed by now, as `Call.getReturnValue()`.

Usage tricks for this callback are quite similar to `check::PreCall`, and these callbacks are often used in pairs. For example, `alpha.unix.SimpleStreamChecker` uses `check::PreCall` for handing `fclose()` calls (where it needs to access the argument) and `check::PostCall` for `fopen()` (where it needs to access the return value). Similarly, the `unix.Malloc` checker uses `check::PreCall` to track `free()` after catching `malloc()` in the `check::PostCall` callback.

4.3.5. check::Location

```
void checkLocation(SVal L, bool IsLoad, const Stmt* S, CheckerContext &C) const;
```

This callback fires every time the program under analysis addresses a certain memory location, either for reading a value from it, or for writing a value into it. Symbolic value `L` would be the l-value (most likely a memory region) being checked, and the `IsLoad` flag is set whenever the access to `L` is read-only. Value `L` is described with statement `S`; if you want to obtain the statement of the access, you'd need to look at its parent statement, probably by using the `ParentMap`. Also, `CheckerContext` is available with its usual functions.

Use this callback whenever you're interested in validating the location rather than the value, that is, whenever accessing the location is "the" event you're interested in. A good usage example is given in the official `core.NullDereference` checker, which subscribes to `check::Location` in order to detect undefined or null-pointer location values.

```
1  void DereferenceChecker::checkLocation(SVal L, bool IsLoad, const Stmt* S,
2                                         CheckerContext &C) const {
3    // Check for dereference of an undefined value.
4    if (L.isUndef()) {
5      if (ExplodedNode *N = C.generateSink()) {
6        /* ... */
7      }
8      return;
9    }
10   DefinedOrUnknownSVal Location = L.castAs<DefinedOrUnknownSVal>();
11   // Check for null dereferences.
12   if (!Location.getAs<Loc>())
13     return;
14   ProgramStateRef State = C.getState();
15   ProgramStateRef NotNullState, NullState;
16   llvm::tie(NotNullState, NullState) = State->assume(Location);
17   if (NullState) {
18     if (!NotNullState) {
19       /* ... */
20     }
21     /* ... */
22   }
23   /* ... */
24 }
```

In the code above, the checker implements a variety of tests in order to explore the nature of the location value and produce different kinds of warnings. You may see how it first detects undefined location values (catches `UndefinedVal`), then tries to assume the location to be null or non-null in the current program state and make decisions based on that. The checker also *splits the program state* in order to discriminate between null and non-null locations if both variants are possible.

4.3.6. check::Bind

```
void checkBind(SVal L, SVal V, const Stmt *S, CheckerContext &C) const;
```

This callback is somewhat similar to `check::Location`. It is called whenever a value is bound to a location, and both the location and the value are available as symbolic values `L` and `V` respectively. Unlike `check::Location`, `check::Bind` does not get called on loads from locations; it only gets called on writes, when a *region binding* appears due to a write operation by the program.

For a substantive example of `check::Bind`, you can see the `alpha.core.BoolAssignment` checker, which checks for assigning values other than 0 or 1 to bool-type variables.

```
1  void BoolAssignmentChecker::checkBind(SVal L, SVal V, const Stmt *S,
2                                        CheckerContext &C) const {
3    // We are only interested in stores into Booleans.
4    const TypedValueRegion *TR =
5      dyn_cast_or_null<TypedValueRegion>(L.getAsRegion());
6    if (!TR)
7      return;
8    QualType valTy = TR->getValueType();
9    if (!isBooleanType(valTy))
10     return;
11   Optional<DefinedSVal> DV = V.getAs<DefinedSVal>();
12   if (!DV)
13     return;
14   ProgramStateRef State = C.getState();
15   SValBuilder &SVB = C.getSValBuilder();
16   DefinedSVal ZeroVal = SVB.makeIntVal(0, valTy);
17   SVal GreaterThanOrEqualToZeroVal =
18     SVB.evalBinOp(State, BO_GE, *DV, ZeroVal, SVB.getConditionType());
19   /* ... */
20   DefinedSVal OneVal = SVB.makeIntVal(1, valTy);
21   SVal LessThanEqToOneVal =
22     SVB.evalBinOp(State, BO_LE, *DV, OneVal, SVB.getConditionType());
23   /* ... */
24 }
```

This checker first inspects the location in order to see if this location is of boolean type. Not every memory region has a type; for example, any void pointer points to a certain memory region, but the analyzer cannot afford making assumptions about the type of values stored in such region. Region that contains values of an explicitly known type is a sub-class of `MemRegion` known as `TypedValueRegion`. The checker aborts unless the region pointed to by `L` certainly has a boolean type. For detailed discussion of various memory region kinds, see subsection 5.2.

Then the checker proceeds to figure out if value `V` is equal to 0 or 1. For that, it creates symbolic comparison values using `SValBuilder`, assumes them to be true or false, and makes decisions based on these assumptions.

4.3.7. check::EndAnalysis

```
void checkEndAnalysis(ExplodedGraph &G, BugReporter &BR, ExprEngine &Eng) const;
```

This callback fires once whenever the path-sensitive analyzer finishes analyzing a certain function code body.

The analysis is reset (and `check::EndAnalysis` callback is called) whenever a function body is fully analyzed. Thus, this callback may be called more than once during analysis of a single translation unit (or, equivalently, more than once during `Checker` object lifetime, or, equivalently, more than once during `clang` run).

This callback fires only once per function code body, rather than once for every branch of the function. This is why `CheckerContext` is not available in this callback, and you cannot obtain the current `ProgramState`. Instead, you have the whole `ExplodedGraph` available. You also have access to the `BugReporter` for throwing bug reports, and to the `ExprEngine` object, which is the unique instance of the analyzer engine.

`check::EndAnalysis` is useful whenever you want to gather statistics across the whole analysis run. One of the extreme examples of using `check::EndAnalysis` is the `deadcode.UnreachableCode` checker:

```
1  void UnreachableCodeChecker::checkEndAnalysis(ExplodedGraph &G,
2                                                BugReporter &BR,
3                                                ExprEngine &Eng) const {
4    /* ... */
5    if (Eng.hasWorkRemaining())
6      return;
7    /* ... */
8    for (ExplodedGraph::node_iterator I = G.nodes_begin(), E = G.nodes_end();
9         I != E; ++I) {
10     /* ... */
11   }
12   /* ... */
13 }
```

This path-sensitive checker finds dead code by understanding which paths were executed during symbolic execution of the function by the engine.

Sometimes the function would be dropped as too complicated; in this case, `hasWorkRemaining()` would return `true`, and the checker would avoid jumping to conclusions. Then the checker proceeds by iterating through the `ExplodedGraph` in order to find which CFG blocks were reached.

4.3.8. check::EndFunction

```
void checkEndFunction(CheckerContext &Ctx) const;
```

This callback fires every time the analyzer leaves the function body. Unlike `check::EndAnalysis`, it fires for every possible return from the function, for every branch of the program execution. Also, when interprocedural analysis is enabled, this callback fires not only when the analysis ends, but also when an analysis of an inlined function call ends.

Consider an example:

```
void bar(int a, int b, int c) {
  if (b) {}
  foo(a);
  if (c) {}
}

void foo(int a) {
  if (a) {}
}
```

In this code, `check::EndFunction` fires twice when analyzing `foo()` in top frame, four times when analyzing `foo()` as called from `bar()`, and eight times when analyzing `bar()`, 14 times total. On the contrary, `check::EndAnalysis` would be called once for `foo()` and once for `bar()`; it would not be called for the pass through `foo()` from inside `bar()`.

You often want to subscribe `check::EndFunction` when you want to find out what remains at the function context at the end of the analysis. One of the examples in the default Clang distribution is the official `core.StackAddressEscape` checker. This checker iterates through all region store bindings in order to find pointers to local variables stored in global variables by the end of the function.

```
void StackAddrEscapeChecker::checkEndFunction(CheckerContext &C) const {
  class CallBack : public StoreManager::BindingsHandler {
  private:
    CheckerContext &C;
    const StackFrameContext *CurSFC;
  public:
    SmallVector<std::pair<const MemRegion*, const MemRegion*>, 10> V;
    CallBack(CheckerContext &CC) :
      C(CC),
      CurSFC(CC.getLocationContext()->getCurrentStackFrame())
    {}
    bool HandleBinding(StoreManager &SMgr, Store Store,
                       const MemRegion *Region, SVal Val) {
      if (!isa<GlobalsSpaceRegion>(Region->getMemorySpace()))
        return true;
      const MemRegion *VR = Val.getAsRegion();
      if (!VR)
        return true;
      /* ... */
      if (const StackSpaceRegion *SSR =
          dyn_cast<StackSpaceRegion>(VR->getMemorySpace())) {
        if (SSR->getStackFrame() == CurSFC)
          V.push_back(std::make_pair(Region, VR));
      }
      return true;
    }
  };
  ProgramStateRef State = C.getState();
  CallBack CB(C);
  C.getStoreManager().iterBindings(State->getStore(), CB);
  /* ... */
}
```

Note the usage of the `StackFrameContext` structure. By comparing the current stack frame with the stack frame of the stack region, the checker understands whether the stack memory region belongs to the same

or to a different stack frame. You almost always need to realize the current `StackFrameContext` when using the `check::EndFunction` callback.

4.3.9. check::BranchCondition

```
void checkBranchCondition(const Stmt *S, CheckerContext &C) const;
```

This callback gets called on every control flow branching that occurs during the analysis of the program. Unlike the `check::PreStmt` and `check::PostStmt` callbacks, which fire for every statement in every CFG basic block, `check::BranchCondition` fires for every CFG terminator instead. Such terminators may include `if` statements, conditional loops, or even short circuits in logical operations `||` and `&&`.

You want to subscribe to this callback whenever you want to figure out what the program uses to make control flow decisions. For example, you may investigate the origin of the symbolic value of the condition, which is available in the environment.

The official `core.uninitialized.Branch` checker relies on this callback to find branch conditions that depend on an undefined value:

```
1 void UndefBranchChecker::checkBranchCondition(const Stmt *S,
2                                               CheckerContext &C) const {
3   SVal Val = C.getState()->getSVal(S, C.getLocationContext());
4   if (Val.isUndef()) {
5     /* ... */
6   }
7   /* ... */
8 }
```

4.3.10. check::LiveSymbols

```
void checkLiveSymbols(ProgramStateRef State, SymbolReaper &SR) const;
```

This callback allows the checker to manually manage garbage collection of range constraints for symbolic expressions. The `SymbolReaper` object is responsible for garbage collection of symbols; you also have access to the current program state in this callback.

Most of the time, unless you really know what you are doing, this callback is only useful for *metadata symbols*. `SymbolMetadata` is a special kind of symbolic expression that is created and managed by the checker itself, and this callback is necessary for managing the lifetime of such symbol.

For example, the `alpha.unix.cstring.OutOfBounds` checker relies on this callback in order to mark metadata symbols that represent string length as live:

```
void CStringChecker::checkLiveSymbols(ProgramStateRef State,
                                      SymbolReaper &SR) const {
  CStringLengthTy Entries = State->get<CStringLength>();
  for (CStringLengthTy::iterator I = Entries.begin(), E = Entries.end();
       I != E; ++I) {
    SVal Len = I.getData();
    for (SymExpr::symbol_iterator SI = Len.symbol_begin(),
                                  SE = Len.symbol_end(); SI != SE; ++SI)
      SR.markInUse(*SI);
  }
}
```

A symbol that represents string length is live whenever the string is the same: even though the memory region that holds the string is the same, changing the value at the null terminator character to a non-null character (or inserting a null character before it) would change the length of a C-style string, and the symbol that represents the old length is no longer necessary and can be released for garbage-collection.

It doesn't mean that the symbol would be instantly deleted; for instance, it would not be deleted as long as it still is stored in another variable in the region store, even if released by the checker.

Metadata symbols would be discussed in detail in 5.5.5.

4.3.11. check::DeadSymbols

```
void checkDeadSymbols(SymbolReaper &SymReaper, CheckerContext &C) const;
```

This callback gets called when the symbol is garbage-collected, and `check::LiveSymbols` didn't prevent that.

On this callback, your checker is notified that this symbol would not be encountered again during further analysis, and you can stop tracking it in your checker-specific data structures. Most likely this assumes removing the symbol information from the GDM of the program state.

This also means that the value represented by the symbol is no longer stored anywhere in the program under analysis; this value is lost forever. For example, if this symbol is a memory address allocated but not freed during analysis, then death of such symbol is a *memory leak*: there's no way for the program to free it once the symbol dies.

Consider `alpha.unix.SimpleStreamChecker`. It uses `check::DeadSymbols` to both clean up its GDM and find file descriptor leaks:

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);
    if (isLeaked(Sym, I->second, IsSymDead, State))
      LeakedStreams.push_back(Sym);
    if (IsSymDead)
      State = State->remove<StreamMap>(Sym);
  }
  ExplodedNode *N = C.addTransition(State);
  reportLeaks(LeakedStreams, C, N);
}
```

4.3.12. check::RegionChanges

```
bool wantsRegionChangeUpdate(ProgramStateRef State) const;
```

```
ProgramStateRef checkRegionChanges(ProgramStateRef State,
                                   const InvalidatedSymbols *Invalidated,
                                   ArrayRef<const MemRegion *> ExplicitRegions,
                                   ArrayRef<const MemRegion *> Regions,
                                   const CallEvent *Call) const;
```

This pair of callbacks allows the checker to monitor all changes in the region store. Unlike `check::Bind` and `check::Location`, this callback also gets called on invalidation, providing the relevant information, such as the optional call event.

As shown on figure 4, `check::RegionChanges` gets called much more often than `check::Location` or `check::Bind`, ensuring exhaustive monitoring of all changes in the store. This callback is also expensive to call, because complete lists of changed symbols and regions are presented. This is why there is an auxiliary callback `wantsRegionChangeUpdate()` that should be defined in order to optimize out the work necessary for calling `checkRegionChanges()` when such work is not necessary.

For example, in the official `alpha.unix.cstring.OutOfBounds` checker, `wantsRegionChangeUpdate()` returns `true` whenever the checker is tracking length of at least one C string:

```
REGISTER_MAP_WITH_PROGRAMSTATE(CStringLength, const MemRegion *, SVal)
/* ... */
bool CStringChecker::wantsRegionChangeUpdate(ProgramStateRef State) const {
  CStringLengthTy Entries = State->get<CStringLength>();
  return !Entries.isEmpty();
}
```

The checker then proceeds with iterating over the `Regions` array in order to remove entries for string length for the changed regions and its sub-regions and super-regions.

	Event	check::Location	check::Bind	check::RegionChanges
1	Load from a variable	✓		
2	Assignment operators	✓	✓	✓
3	Initializations		✓	✓
4	Temporary value creation			✓
5	Default bindings			✓
6	Garbage collection of bindings			✓
7	Invalidation			✓

Figure 4: Comparison of the three region store-related callbacks.

```
ProgramStateRef
CStringChecker::checkRegionChanges(ProgramStateRef State,
                                   const InvalidatedSymbols *Invalidated,
                                   ArrayRef<const MemRegion *> ExplicitRegions,
                                   ArrayRef<const MemRegion *> Regions,
                                   const CallEvent *Call) const {
  llvm::SmallPtrSet<const MemRegion *, 8> InvalidatedRegions;
  llvm::SmallPtrSet<const MemRegion *, 32> SuperRegions;
  for (ArrayRef<const MemRegion *>::iterator
       I = Regions.begin(), E = Regions.end(); I != E; ++I) {
    const MemRegion *MR = *I;
    InvalidatedRegions.insert(MR);
    SuperRegions.insert(MR);
    while (const SubRegion *SR = dyn_cast<SubRegion>(MR)) {
      MR = SR->getSuperRegion();
      SuperRegions.insert(MR);
    }
  }
  CStringLengthTy::Factory &F = State->get_context<CStringLength>();
  for (CStringLengthTy::iterator I = Entries.begin(),
       E = Entries.end(); I != E; ++I) {
    const MemRegion *MR = I.getKey();
    if (SuperRegions.count(MR)) {
      Entries = F.remove(Entries, MR);
      continue;
    }
    const MemRegion *Super = MR;
    while (const SubRegion *SR = dyn_cast<SubRegion>(Super)) {
      Super = SR->getSuperRegion();
      if (InvalidatedRegions.count(Super)) {
        Entries = F.remove(Entries, MR);
        break;
      }
    }
  }
  return State->set<CStringLength>(Entries);
}
```

For better performance, super-regions of invalidated regions are stored in an `llvm::SmallPtrSet`, which is of course problematic with sub-regions. Also note how the checker avoids creating multiple intermediate program states for each region removal, working on the immutable map directly.

4.3.13. check::PointerEscape

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

Whenever a pointer value is assigned to a global variable, or passed into a function that the analyzer cannot model, the pointer is said to "escape". Such pointer cannot be reliably tracked any longer. When a pointer escapes, `check::PointerEscape` is called in order to notify the checkers for escape of pointers they were interested in.

If the pointer escape occurs during invalidation, information on the call event is provided.

Similarly to how `check::DeadSymbols` can be used for detecting resource leaks, `check::PointerEscape` can be used for eliminating false positives in such checks: an escaped pointer could have been freed without us knowing, or value beyond it may have been changed.

In `alpha.unix.SimpleStreamChecker`, this callback is used for finding escaped file descriptors:

```
1  ProgramStateRef
2  SimpleStreamChecker::checkPointerEscape(ProgramStateRef State,
3                                          const InvalidatedSymbols &Escaped,
4                                          const CallEvent *Call,
5                                          PointerEscapeKind Kind) const {
6    if (Kind == PSK_DirectEscapeOnCall && guaranteedNotToCloseFile(*Call)) {
7      return State;
8    }
9    for (InvalidatedSymbols::const_iterator I = Escaped.begin(),
10                                            E = Escaped.end();
11                                           I != E; ++I) {
12     SymbolRef Sym = *I;
13     State = State->remove<StreamMap>(Sym);
14   }
15   return State;
16 }
```

On line 6, a custom check is performed to avoid considering escapes on certain expected kinds of invalidation events — in order to heuristically determine if the function is of interest.

4.3.14. eval::Assume

```
ProgramStateRef evalAssume(ProgramStateRef State, SVal Cond,
                           bool Assumption) const;
```

This callback fires every time a new range constraint appears in the program state. With this callback, checkers can be notified on new constraints imposed over symbols they store internally, or let them help the analyzer with "evaluating" the assumption, together with the constraint manager, modifying the program state. However, before using this callback, see if `check::BranchCondition` may be enough for your purposes.

For example, the `unix.Malloc` uses this callback to find if any of the symbols pointing to allocated memory were constrained to a null-pointer value. As soon as the symbol disintegrates into a concrete value, it is pointless to track such symbol any longer:

```
ProgramStateRef MallocChecker::evalAssume(ProgramStateRef State, SVal Cond,
                                          bool Assumption) const {
  RegionStateTy RS = State->get<RegionState>();
  for (RegionStateTy::iterator I = RS.begin(), E = RS.end(); I != E; ++I) {
    ConstraintManager &CMgr = State->getConstraintManager();
    ConditionTruthVal AllocFailed = CMgr.isNull(State, I.getKey());
    if (AllocFailed.isConstrainedTrue())
      State = State->remove<RegionState>(I.getKey());
  }
  /* ... */
  return State;
}
```

4.3.15. eval::Call

```
bool evalCall(const CallExpr *CE, CheckerContext &C) const;
```

This checker callback allows the checkers to model a function call, overriding the usual interprocedural analysis mechanism. It may be useful for modeling domain-specific library functions, when the source code of the function is not available for analysis.

The callback should return `true` if the checker has successfully modeled the function call, and `false` if the checker would better rely on the analyzer core or on other checkers to evaluate this call.

Usage of this callback is *discouraged* because *only one checker may evaluate any call event*; if two or more checkers, probably developed by different people, accidentally evaluate the same function, behavior of the analyzer is undefined. So, if possible, `check::PreCall` and `check::PostCall` should be considered, and most of the time they are flexible enough to model effects of the call on the program state.

The official `core.builtin.BuiltinFunctions` checker uses this callback in order to emulate behavior of certain compiler built-in functions:

```
bool BuiltinFunctionChecker::evalCall(const CallExpr *CE,
                                      CheckerContext &C) const {
  const FunctionDecl *FD = C.getCalleeDecl(CE);
  if (!FD)
    return false;
  ProgramStateRef State = C.getState();
  const LocationContext *LCtx = C.getLocationContext();
  switch (FD->getBuiltinID()) {
  /* ... */
  case Builtin::BI__builtin_addressof: {
    assert (CE->arg_begin() != CE->arg_end());
    SVal X = State->getSVal(*(CE->arg_begin()), LCtx);
    C.addTransition(State->BindExpr(CE, LCtx, X));
    return true;
  }
  /* ... */
  }
  /* ... */
}
```

4.4. Implementing bug reporter visitors

Usually `BugReporter` does a fairly good job at explaining how exactly was the path-sensitive bug discovered, displaying all events along the symbolic execution path to the user. Sometimes, however, you may want it to mark and display additional events. For instance, when reporting a double-free bug, you may want to let the user know when the first free occurred. In this case, you need to implement a bug reporter visitor, which would walk through the bug report path, as a list of `ExplodedNode`'s, from start to end, and inject path diagnostic pieces along the way.

The syntax for bug reporter visitors is as follows:

```
class MyVisitor : public BugReporterVisitorImpl<MyVisitor> {
  void Profile(llvm::FoldingSetNodeID &ID) const {
    /* ... */
  }
  PathDiagnosticPiece *VisitNode(const ExplodedNode *N,
                                 const ExplodedNode *PrevN,
                                 BugReporterContext &BRC,
                                 BugReport &BR) {
    /* ... */
    if (const Stmt *S = /* Obtain a statement for diagnostic */) {
      PathDiagnosticLocation Pos(S, BRC.getSourceManager(),
                                 N->getLocationContext());
      return new PathDiagnosticEventPiece(Pos, "Message");
    }
    return NULL;
  }
};
```

You need to implement the `Profile(...)` method because bug visitors would be stored in an LLVM folding set of path diagnostic callbacks.

Then you need to implement `VisitNode(...)`. It should identify the node of interest, construct a path diagnostic for it and return it, or return a null pointer if the node should be skipped.

It is not uncommon to identify nodes by statements in their program points. In this case, the static helper method `getStmt(...)` of `PathDiagnosticLocation` class should be useful:

```
const Stmt *S = PathDiagnosticLocation::getStmt(N);
```

Note, however, that there are most likely multiple nodes corresponding to the same statement.

4.5. Understanding interprocedural analysis

The way CSA models function calls is fairly straightforward and transparent for most checkers. The analyzer normally handles function calls by *inlining* the callee code body, proceeding with execution of the callee code right after its arguments were evaluated, and until a return occurs, then finally binding the return value, if any, to the call expression in the environment and continue analysis of the caller.

If the source code of the function body is not available for the analyzer, it tries to evaluate the function *conservatively*. As little as possible is assumed about the function during conservative evaluation, and an *invalidation* of known information usually occurs.

Additionally, CSA checkers may override the evaluation procedure for certain functions, by subscribing on the `eval::Call` callback.

In order to implement certain checks, you may need to understand certain peculiarities of both the inlining procedure and the conservative evaluation procedure.

4.5.1. Conservative evaluation and invalidation

When both inlining and checker-side evaluation of the call fails, the analyzer falls back to conservative evaluation. Such evaluation is relatively simple, as nothing really gets evaluated. Instead, the analyzer needs to drop all information that was formerly known and might have become invalid. The process of erasing such information is called *invalidation*.

Invalidation is mostly handled by the region store. The function may write unknown values to all locations available to it, such as global variables or regions that were passed into it as arguments. New, unconstrained symbolic expressions of type `SymbolConjured` (this type of symbols is discussed in detail in 5.5.2) are created in order to represent these values, and bound to the invalidated regions in the region store.

There are two checker callbacks that let you catch invalidation events in your checker and take action:

— `check::PointerEscape` lets you handle an event of a pointer symbol being passed into a conservatively evaluated function.

— `check::RegionChanges` lets you observe the complete consequences of invalidation, including a list of invalidated regions.

4.5.2. Inlining and stack frames

Inlining function calls is a heavy operation for the analyzer. Every function call needs to be modeled again and again in every new context, and the context-specific exploded graph (which may have different values for variables of the context, and lack branches unreachable in the context) of the callee becomes a sub-graph of the exploded graph of the current analysis.

There are multiple preconditions required for inlining to happen, including:

— Source code of the callee function body needs to be available;

— No checker should evaluate the function call via `eval::Call`;

— If the analysis of the callee reaches maximum exploded node limit, the callee would never be inlined, but evaluated conservatively instead;

— Even though recursion is supported, only a limited number of nested recursive calls would be executed.

Whenever an analyzer inlines a function and descends into it, a new `StackFrameContext` is created. This structure is a kind of `LocationContext` that describes the location of descending into a function during interprocedural analysis. You can obtain the current stack frame via the `getStackFrame()` method of the `CheckerContext`. Very often all you need to know is that if we are inside an inlined function or in top frame; in this case, a convenient `inTopFrame()` method of the `CheckerContext` can be used.

One of the common cases when you need to work with stack frames is the `check::EndFunction` checker callback. This callback fires on every return from the function, however you need to see if it is the end of the analysis or merely a pop from a stack frame.

Sometimes, you may want to rely on symbolic value hierarchy (see section 5) in your checker logic. Then, you would know that the symbolic value that represents a function argument value would be a symbol of type `SymbolRegionValue` for a region of type `VarRegion` for a decl of type `ParmVarDecl`. However, for inlined calls, this is no longer true; for instance, the argument may be an arbitrary `SVal`, whichever was passed to the function in the current caller context. So if you rely on such checks, you would most likely need to code additional checks to understand the status of IPA in the current event.

4.6. Further reading

The method of symbolic execution was first defined in 1976 in an academic article by James C. King[11]. In particular, it describes the idea of the program state, and how the analyzer splits possible program states into equivalence classes.

The implementation of interprocedural analysis in CSA is based on a work by T. Reps, S. Horwitz, and M. Sagiv[12].

[11] James C. King. Symbolic execution and program testing. In: Communications of the ACM, vol. 19. N7. pp. 385–394 (1976)

[12] Precise interprocedural dataflow analysis via graph reachability, T Reps, S Horwitz, and M Sagiv, POPL '95, http://portal.acm.org/citation.cfm?id=199462

5. The symbolic value hierarchy

Symbolic values are the notation CSA uses for describing known and unknown values it encounters during symbolic execution of the program. CSA uses a very complex hierarchy of symbolic values.

The basic class for representing various symbolic values is the `SVal` class. It has various sub-classes which represent different kinds of symbolic values. There are also two auxiliary classes, `MemRegion` and `SymExpr`, that specifically handle memory regions and symbolic expressions respectively.

Objects of `SymExpr` class are also often referred to as *symbols*, and represent unknown numeric values; if a value is known during analysis, it is called a *concrete value*. `MemRegion` objects — "*regions*" — are used for two purposes: as locations for region store bindings in the memory model of the analyzer, and in order to represent pointer values.

The three classes are very much inter-connected. For example, regions may be "based on" symbols and concrete values (for example, a region pointed to by a pointer symbol, or a region of an array element with a known or unknown index), symbols may be "based on" regions (for example, a symbol defined as an initial value of a region).

Additionally, `SVal` sub-classes are split into two large categories: `Loc` for l-values and `NonLoc` for r-values.

Figure 5 illustrates the functional difference between these classes.

	Role	`SVal`	`MemRegion`	`SymExpr`
1	Serve as range constraint keys			✓
2	Serve as region binding keys		✓	
3	Serve as region binding values	✓		
4	Serve as environment values	✓		
5	Carry taint			✓
6	Carry metadata		✓	
7	Serve as metadata values			✓
8	Be stored in the GDM	✓	✓	✓

Figure 5: Comparison of values, memory regions, and symbolic expressions.

It only makes sense to constraint symbols; concrete values are already known, so it's pointless to assign integral range constraints upon them any further, and memory region addresses are never really defined in compile-time. It is also natural that region store works by assigning arbitrary values to regions, and environment works by assigning arbitrary values to AST expressions.

Line 5 of this table may surprise you, as above we have been discussing tainted memory regions, and some methods that work with taint accept arbitrary `SVal`'s. However, when taint analysis works with values other

than symbols, it merely tries to find symbols inside them. We are going to discuss it in detail in subsection 5.6. The concept of metadata symbols would be discussed later in 5.5.5. Finally, GDM can carry pretty much anything, that's what "G" stands for.

5.1. Constructing symbolic values

The `SValBuilder` class provides methods for constructing `SVal` objects. It allows constructing all kinds of `SVal`'s and all kinds of `SymExpr`'s (representing the latter as `SVal`'s if necessary). It also allows evaluating operations on symbolic values.

However, for constructing memory regions, you should be using the `MemRegionManager` object. It is sometimes useful to be able to construct a sub-region (eg. a field region for a structure region with a known declaration, in order to later obtain a value of the field).

You should almost never construct a `SymExpr`. A few rare cases when you want to construct a symbol include creating some sort of `SymbolConjured` during some sort of `eval::Call`, and also constructing `SymbolMetadata` when your checker uses this mechanism. Most of the time, however, you would be receiving all the necessary symbols from the environment or the region store, and rarely even care about their kind.

In any case, you should be using methods of `SValBuilder`, rather than accessing the `SymbolManager` object directly, for constructing all kinds of `SymExpr`'s. These methods would return a `nonloc::SymbolVal` containing the symbol if the symbol requested is of integral type, or a `loc::MemRegionVal` containing a `SymbolicRegion` wrapping the symbol if a pointer type was requested. In both cases, you can call the `getAsSymbol()` method of the resulting `SVal` to obtain the `SymExpr` itself.

5.2. Memory model of the analyzer

`MemRegion` is a segment of memory. When it is stored inside an `SVal` of a pointer type, it represents the address of the first byte of the segment; however, you should still imagine the `MemRegion` object as carrying information about the whole segment.

The `getAsRegion()` method of the `SVal` class works for the following `SVal` kinds:

- `loc::MemRegionVal` — a pointer value described as the address of the first byte of the given region.
- `nonloc::LocAsInteger` — a similar pointer value, just stored inside an integer. This kind of `SVal`'s represents results of pointer-to-integer casts.

Some memory regions are *sub-regions* of other regions. A sub-region is a sub-segment inside a segment. Sub-regions inherit from `SubRegion` class. Every sub-region has a length ("extent"), which may be obtained with the `getExtent(...)` method of `SubRegion`. Extent may be either concrete or symbolic.

Other regions, known as *memory spaces*, do not belong inside any other region.

Each `SubRegion` has exactly one direct super-region obtained via the `getSuperRegion()` method. Memory regions that have a memory space as their direct super-region are called *base regions*. If a region is neither a memory space nor a base region, then there is exactly one base region at the end of its super-region chain. There is a separate family of classes for representing base regions: by looking only at the class of the region, you can determine if it is a base region inside a memory space, or is located inside another base-region.

You can obtain the memory space in which the region belongs with `getMemorySpace()` method, and obtain the base region for any sub-region with `getBaseRegion()`.

Base regions — the direct sub-regions of memory spaces — can be either *typed* or *untyped*. A typed region is a region that holds values of a known type. An untyped region is a region with a value of unknown type, even though you may have a rough idea of what is stored there or where it came from.

For example, consider the following code:

```
1  struct A {
2    int x, y;
3  };
4  struct B: A {
5    int u, v;
6  };
7  struct C {
8    int t;
9    B *b;
10 };
11 void foo(C c) {
12   c.b[5].y; // <-- that
13 }
```

Then the system of regions describing field `y` on line 20 is vaguely depicted on figure 6.

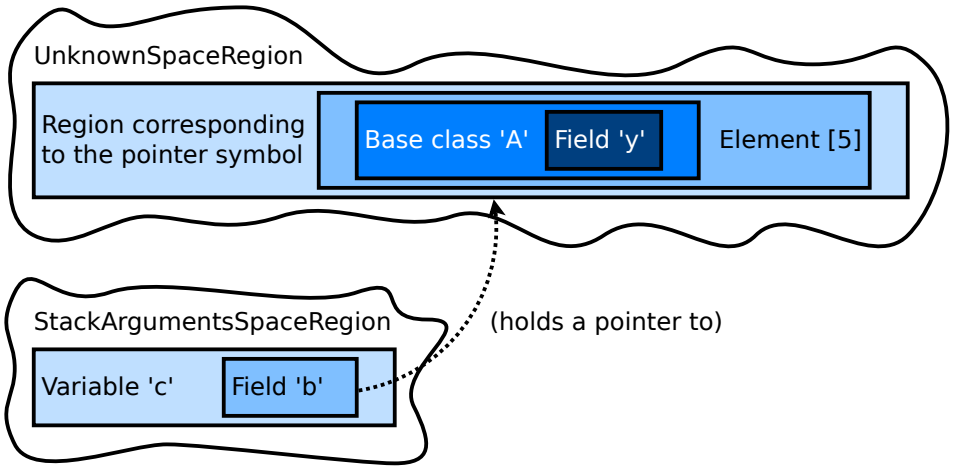

Figure 6: The way the analyzer represents `c.b[5].y`.

If you `dump()` this region to `stderr` during analysis, you would see a pretty print like

```
base{element{SymRegion{reg_$0<c->b>},5 S32b,struct B},A}->y
```

In words, it would be expressed as:

> `FieldRegion` for declaration of member variable `y`,
> inside `CXXBaseRegion` for declaration of class `A`,
> inside `ElementRegion` for element number 5 of type `B`,
> inside `SymbolicRegion` for the pointer symbol of `SymbolRegionValue` kind,
> which represents the initial value of:
>
> `FieldRegion` for declaration of member variable `b`,
> `VarRegion` for declaration of a local variable `c`.

You should be able to understand most of this picture after reading this subsection, with an exception of the SymbolRegionValue thing, which is explained in subsection 5.5.

5.2.1. Memory spaces

Memory spaces inherit from the MemSpaceRegion class. Most memory spaces are "singletons", and there are actually very few of them:

- GlobalsSpaceRegion — a base class for four different memory spaces:
  - NonStaticGlobalSpaceRegion — the single memory space for all non-static global variables, which is split into three:
    - GlobalImmutableSpaceRegion, which consists of globals that cannot be modified,
    - GlobalSystemSpaceRegion, which includes variables that are most likely only modified by system calls, such as errno,
    - GlobalInternalSpaceRegion, which consists other global variables,
  - StaticGlobalSpaceRegion — the memory space for all static global variables,
- HeapSpaceRegion — holding all regions allocated on the heap.
- StackSpaceRegion — a base class for two different memory spaces:
  - StackArgumentsSpaceRegion — memory space of function call arguments,
  - StackLocalsSpaceRegion — memory space for local variables.

  Note that unlike other memory spaces, there may be multiple StackSpaceRegion instances — one for every StackFrameContext.
- UnknownSpaceRegion — whenever the analyzer has no idea where the region is actually stored.

Memory spaces are important because regions are considered different if they are inside different memory spaces, even if all other traits of these regions are equal. For example, regions of the function parameter variable in different calls are different because their memory spaces are defined by different stack frame contexts, even though variable declaration is the same.

5.2.2. Untyped base regions

There are only three kinds of untyped regions:

- AllocaRegion — a region allocated on the stack by calling the alloca() function of the standard C library. This region is untyped because this function allocates raw data.

  AllocaRegion always resides in StackLocalsSpaceRegion.
- SymbolicRegion — a region pointed to by a pointer, value of which is a symbolic expression. This region is untyped, because pointers can be casted freely in C, and you cannot be sure that type of data it points to matches the pointer type.

  SymbolicRegion class deserves special attention due to the fact that pointer symbols, even though their type is a pointer type, are technically NonLoc. So the purpose of the SymbolicRegion class is to express and deliver the Loc part of things — a region that is created after a pointer value, rather than vice versa.

  If a sub-region has SymbolicRegion as its base region, the region is said to have *symbolic base*, and the symbolic region is said to be the symbolic base of that region. The getSymbolicBase() method

of MemRegion returns a pointer to the symbolic base region or a null pointer if the base isn't symbolic. This method is often useful when you need to figure out if a complex sub-region is actually related to a certain pointer symbol.

SymbolicRegion normally resides in UnknownSpaceRegion, because the nature of the pointer is often unknown. However, sometimes the pointer is known to point to heap (for example, if it was returned by the default operator new), and then the region would reside in HeapSpaceRegion. Heap symbolic regions are created with getSymbolicHeapRegion() method of MemRegionManager.

5.2.3. Typed base regions with typed values

Typed regions are regions with a common ancestor class known as TypedValueRegion, though not all its successors are base regions; in fact, sub-regions of base regions are typed as well. Typed regions enjoy much more variety:

— VarRegion is a region of a variable. For every AST global or static variable declaration, one VarRegion is defined. For stack variables, regions can be different inside different function calls, simply by being sub-regions of different StackSpaceRegion memory spaces. Also note that member variable of a class is not at all a base region and is never represented with a VarRegion.

VarRegion may reside in various memory spaces, depending on the nature of the variable declaration.

— CXXThisRegion is the region where the implicit this pointer is stored during a C++ method call. This typed region is always located on the stack, and there is at most one CXXThisRegion for every stack frame context (that is, for every StackArgumentsSpaceRegion space), much like VarRegion's of function parameters.

Note that CXXThisRegion is not the object itself, but merely a stack region holding the pointer. The object itself would be the symbolic value stored in this region. For a top-level call, the object region would be described as the SymbolicRegion of the SymbolRegionValue of CXXThisRegion of the top-level stack frame; in particular, it would be untyped, even though CXXThisRegion itself is always typed. For nested function calls during interprocedural analysis, the current object region may be typed (eg. when CXXThisRegion of the stack frame of the nested call holds a pointer to a VarRegion for a known variable).

— CXXTempObjectRegion represents memory regions of a C++ temporary object. It appears when semantics of C++ require creating an auxiliary invisible object, for example, when creating an object by calling a constructor directly without operator new. This region holds memory of an AST expression that caused it to appear.

CXXTempObjectRegion may reside in StackLocalsSpaceRegion, and it may also sometimes reside in GlobalInternalSpaceRegionKind, when the getCXXStaticTempObjectRegion() method of the MemRegionManager was used for creating it.

— CompoundLiteralRegion represents memory region of an initializer-list ("compound literal") object.

— StringRegion — a region of a string literal.

5.2.4. Typed base regions with untyped values

There are a few special kinds of regions that inherit from TypedRegion but not TypedValueRegion. These regions have well-defined "location" (pointer) type, however the type of the values they store is not defined as the pointee type of the location type.

— `BlockDataRegion` — a base region for representing data stored inside blocks (the non-standard Apple Inc. extension to C and C++). These regions handle both code and data for the block, and implement methods for working with closures.

— `CodeTextRegion` represents memory regions of program code rather than data. There are two subkinds: `FunctionTextRegion` for function code, often used for representing function pointer values in the analyzer, and `BlockTextRegion` for block code.

5.2.5. Sub-regions of base regions

Sub-regions of base regions are always typed, even if the base region is untyped.

— `CXXBaseObjectRegion` is the region of a base class object inside a region of an object of derived class. This region is defined by the base class declaration in the AST.

— `ElementRegion` is a region of an array element inside a solid one-dimensional array. The index of the element is an arbitrary `NonLoc` symbolic value of array index type, which is either a concrete integer, or a symbol. This region also carries type information; `ElementRegion`s of same super-region with same index but different type are considered different.

`ElementRegion` is also used for representing type casts for untyped regions. For example, value of a symbolic pointer casted to type `T*` is represented as element region of value type `T` of a symbolic region over this pointer. If the pointer is casted further to another type `S*`, then this `ElementRegion` may be replaced with another `ElementRegion` of value type `S`.

— `FieldRegion` is a region of a field inside a structure or class or union. Similarly to `VarRegion`, this region is also based on an AST variable declaration.

Note that the `ElementRegion` does not represent a pointer dereference (instead, the `SymbolicRegion` does), and subscripting pointers and arrays is handled completely differently.

For example, consider a function `foo()`:

```
void foo(int *p, int a[5]) {
  /* ... */
}
```

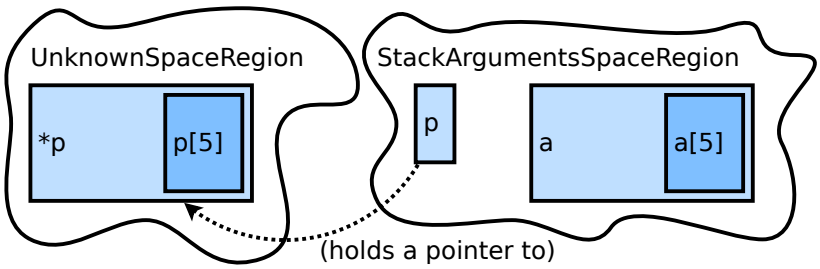

Figure 7: The system of regions that represents `p[5]` and `a[5]` during analysis of `foo()`.

Then `a[5]` would be an `ElementRegion` of `VarRegion` of parameter variable `a`, which lies somewhere in `StackArgumentsSpaceRegion`:

```
element{a,5 S32b,int}
```

However, `p[5]` is an `ElementRegion` of `SymbolicRegion` in `UnknownSpaceRegion`, constructed for the symbolic pointer value of parameter variable `p`:

```
element{SymRegion{reg_$0<p>},5 S32b,int}
```

The respective region hierarchy is displayed on figure 7.

5.3. Concrete values

Concrete values are values known in compile-time. If a value of an integer variable is known to be 42, then there exists a concrete value representing it, and also there are no two different symbolic values that might accidentally represent it; one is enough.

5.3.1. Numeric values

The most primitive concrete value, representing an integer with value known in compile-time. Internally, `nonloc::ConcreteInt` holds an `llvm::APSInt` inside; you can obtain it via `getValue()`:

```
nonloc::ConcreteInt CI = Val.castAs<nonloc::ConcreteInt>();
uint64_t Int = CI.getValue().getLimitedValue();
```

Different instances of `nonloc::ConcreteInt` class have different numeric value, type size, and signedness. However, for each combination of the three, there is only one `nonloc::ConcreteInt` representing it; you cannot discriminate between two 32-bit signed zeros obtained from different sources.

A `loc::ConcreteInt` is a concrete integer representing a known pointer value. Internally it is similar to `nonloc::ConcreteInt`. Usually instances of `loc::ConcreteInt` would be unsigned integers of pointer width. It is very unlikely to know a memory address in compile-time, so the most common value you would see in `loc::ConcreteInt` is 0, representing a null-pointer.

5.3.2. Compound values

The simplest example of a concrete compound value is `nonloc::CompoundVal`, which represents a concrete r-value of an initializer-list or a string. Internally, it contains an `llvm::ImmutableList` of `SVal`'s stored inside the literal.

However, there is another compound value used in the analyzer, which appears much more often during analysis, which is `nonloc::LazyCompoundVal`. This value is an r-value that represents a snapshot of any structure "as a whole" at a given moment during the analysis. Such value is already quite far from being referred to as "concrete", as many fields inside it would be unknown or symbolic. `nonloc::LazyCompoundVal` operates by storing two things:

— a reference to the `TypedValueRegion` being snapshotted (yes, it is always typed), and also

— a copy of *the whole* `Store` object, obtained from the `ProgramState` in which it was created.

Essentially, `nonloc::LazyCompoundVal` is a performance optimization for the analyzer. Because `Store` is immutable, creating a `nonloc::LazyCompoundVal` is a very cheap operation. Note that the `Store` contains all region bindings in the program state, not only related to the region. Later, if necessary, such value can be unpacked — eg. when it is assigned to another variable.

5.4. Special values

This subsection describes two singleton values reserved for special purposes.

5.4.1. UndefinedVal

Whenever it is necessary for the analyzer core to emphasize that the value of something (an expression or a region) is undefined according to the language standard, an `UndefinedVal` is produced. There is only one `UndefinedVal`: you cannot discriminate between two `UndefinedVal`'s obtained from different sources.

Most of the time, whenever an `UndefinedVal` appears, there should be a checker to warn that an undefined behavior has occurred. There are multiple official checkers that throw this type of warnings in the `core.uninitialized` package.

A common example of a situation in which `UndefinedVal` appears is trying to obtain a value of an uninitialized variable.

This is the only value that is banned in the `assume(...)` method of the `ProgramState`.

5.4.2. UnknownVal

Whenever a symbolic execution engine fails to represent a certain value with a symbol, it creates another special value called `UnknownVal`. Like `UndefinedVal`, `UnknownVal` is a singleton value; you cannot discriminate between two `UnknownVal`'s obtained from different sources. However, you can discriminate between `UnknownVal` and `UndefinedVal`.

An `UnknownVal` may appear anywhere, anytime. It often appears when a symbolic expression exceeds its complexity limit. Its appearance in any place, no matter how critical, does not instantly indicate an error in the program, however it most likely indicates a failure of the analyzer core: lack of a distinct symbol for an unknown value defeats the purpose of symbolic execution. Most of the time, the analyzer "conjures" a special symbol for such values, but sometimes it wants to be sure you make completely no assumptions against this value, and thus creates an `UnknownVal`. Most likely you'd want to avoid throwing warnings when you encounter an `UnknownVal` in a checker callback.

5.5. Symbolic expressions

Symbolic expressions, also referred to as symbols, are without doubt the essence of the whole idea behind symbolic execution.

Symbols are *timeless*: a symbolic value cannot "change" during analysis. Once a symbol is created, it represents the same value throughout all analysis. However, when the analysis progresses further through the program, new information may be gathered about this value and stored inside `ProgramState` in the form of *range constraints* imposed over this symbol.

For example, on the `true` branch of an `if` statement, the symbol representing the condition value would be known to be non-zero. On the `false` branch, it would be known to be equal to zero, essentially turning into a concrete value; in fact, the `getSVal(...)` family of methods would internally substitute such symbol with a concrete integer 0.

However, classification of symbolic values is very rarely important. Most of the time, the most important thing you need to know about symbols is that values represented by the same symbol are always equal, while different symbols may or may not represent different values. In fact, most of the analysis would work fairly well even if all symbols everywhere were of `SymbolConjured` type. Here are some of the benefits we have due to possessing a hierarchy of different classes for symbols:

— `RangeConstraintManager` uses symbolic binary-expression classes to significantly simplify constraint conditions, eg. $(x + 3) > 5$ is easily transformed into $x > 2$, which would be hard if the symbol representing $(x + 3)$ didn't remember anything about x or 3.

— Taint propagates automatically from tainted regions to data symbols representing their values, via the reference to the region stored inside the symbol.

— At any moment, we can easily trace the origin of the symbol in a high-level manner. If we want to figure out what conditions are necessary in order to replicate the bug found by the analyzer, we can often do so by looking at the symbols inside the program state.

— It is also often useful for debugging, and sometimes — very rarely — the internal logic of the checker itself would rely on symbol kinds. However, you should know what you're doing; this technique is often misused or used for quick and dirty incorrect heuristics. This subsection should give you a rough idea of what symbols are and what symbols aren't.

Symbol values always have a type, which is an integer or a pointer.

The `getAsSymbol()` method of the `SVal` class works with the following `SVal` kinds:

— `nonloc::SymbolVal` — the value which "is" the symbol "itself".

— `loc::MemRegionVal`, if the region inside it is a `SymbolicRegion` (a "symbolic pointer" — because `nonloc::SymbolVal` is always `NonLoc`, this method represents pointers as `Loc` values). In this case, the method would return the symbol for which the region is constructed. If the optional boolean parameter of `getAsSymbol()` is set to `true`, this method would also work on arbitrary regions with symbolic base.

— `nonloc::LocAsInteger` — extracting a symbolic pointer from the underlying `loc::MemRegionVal`, if any.

5.5.1. Operation symbols

There are three symbols that represent binary operators on other symbols:

— `SymIntExpr` represents result of a binary operation between another symbol and a concrete integer, eg. $x + 5$;

— `IntSymExpr` represents result of a binary operation between a concrete integer and another symbol, eg. $3 > x$;

— `SymSymExpr` represents result of a binary operation between a concrete integer and another symbol, eg. $x * y$. Note that `SymSymExpr` symbols are created by the analyzer very rarely, because they are mostly useless for the `RangeConstraintManager`, which cannot handle complicated constraints. Usually `SymSymExpr`'s appear when one of the operands is tainted, in order to keep taint information.

There is also `SymbolCast`, which represents the result of a cast into a certain type from another symbol.

5.5.2. Conjured symbols

`SymbolConjured` is a fallback when everything else fails: the analyzer failed to make any sense at all from the expression, so it conjured up at least some symbol in order to keep path-sensitivity. Common examples of `SymbolConjured` include return values of functions which were not modeled by the analyzer, because their source code of their body was not available, or for other reasons; it is also used for purposes of invalidation.

5.5.3. Region value symbols

Probably the most primitive "sensible" atomic ("data") symbol, which we have already mentioned a few times, is `SymbolRegionValue`. It represents the value stored in the memory region *at the beginning of the analysis*. This symbol contains a reference to the region.

Consider the following example:

```
1 void foo(int a) {
2   int b = a;
3   a = 1;
4 }
```

At the beginning of the analysis (before line 2), value of `b` is an `UndefinedVal`, while value of `a` is a symbol of class `SymbolRegionValue` representing the value of region of parameter variable `a`.

After line 2, the value of `b` is changed to the `SymbolRegionValue` representing the value of `a`.

After line 3, the value of `a` is changed to a `nonloc::ConcreteInt` with value 1. However, `b` still holds a symbol of `SymbolRegionValue` kind for the region of variable `a`, which still represents the *original* value of `a`, rather than the new concrete value 1.

Another atomic symbol, closely related to `SymbolRegionValue`, is `SymbolDerived`. It represents a value of a region *after* another symbol was written into a direct or indirect super-region. `SymbolDerived` contains a reference to both the parent symbol and the parent region. This symbol is mostly a technical hack. Usually `SymbolDerived` appears after *invalidation*: the whole structure of a certain type gets smashed with a single `SymbolConjured`, and then values of its fields become represented with the help of `SymbolDerived` of that conjured symbol and the region of the field. In any case, `SymbolDerived` is similar to `SymbolRegionValue`, just refers to a value after a certain event during analysis rather than at start of analysis.

5.5.4. Extent symbols

As we mentioned in subsection 5.2, every memory region is a segment of bytes. We are usually interested in the address of the first byte, but sometimes we may try to find out the length ("extent") of the region, which may be known (for plain variable regions) or unknown (usually for symbolic regions, which may actually be arrays of unknown length). When the extent is unknown, it is represented with a special symbol called `SymbolExtent`. This symbol contains a reference for the region.

Of course, if you need to obtain extent of a certain region, you shouldn't be creating a new `SymbolExtent` manually; you can rely on the `getExtent(...)` method of `SubRegion`.

5.5.5. Metadata symbols

Metadata symbols are symbols with a checker-specific meaning, tied to memory regions. The checker may create such symbols and manage their lifetime and garbage collection (via the `check::LiveSymbols` callback). The analyzer core never creates `SymbolMetadata` on its own; only a checker can create such symbols.

You can use `SValBuilder` to create a new `SymbolMetadata`. Here is an example code from the official `alpha.unix.cstring.OutOfBounds` checker which creates metadata symbols that represent string length:

```
SValBuilder &SVB = C.getSValBuilder();
QualType SizeTy = SVB.getContext().getSizeType();
SVal StrLength = SVB.getMetadataSymbolVal(CStringChecker::getTag(),
                                          MR, Ex, SizeTy, C.blockCount());
```

`SymbolMetadata` is made with the following ingredients:

— A symbol tag — a `void*` that uniquely identifies a kind of metadata symbols. In this example, the unique identifier of the checker itself, returned by the static `getTag()` method of the `Checker` object, is being used.

— The parent region, to which the metadata is tied; in our case, it is the region of the string for which the length is defined.

— An AST expression on which the symbol appeared.

— The expected type of the symbol. As expected from an `SValBuilder` method, if this type is `Loc`, then the resulting `SVal` would be of type `loc::MemRegionVal`, carrying a `SymbolicRegion` wrapping the symbol.

— The block count: number of times the CFG block was visited during analysis. This allows discriminating symbols created on the same expression for the same region, whenever it is passed-through multiple times during analysis.

5.6. Tainted values

As mentioned above, any symbols, and only symbols, can carry taint. However, for convenience, other values are said to inherit taint information from symbols on which they rely. Below is the complete list of cases of taint propagation through symbolic value hierarchy:

— Memory regions are said to be tainted in the following cases:

  — A `SymbolicRegion`, when constructed with a tainted pointer symbol;

  — An `ElementRegion`, when constructed with a tainted index value;

  — Any kind of region inherits taint from its super-region.

— Symbols can inherit taint, regardless of their own taint information, in the following cases:

  — `SymbolRegionValue` may inherit taint from its parent region;

  — `SymbolDerived` may inherit taint from its parent symbol, but not from its parent region;

  — All operation symbols inherit taint from their operands.

— `SVal`'s are said to be tainted when a symbol extracted with `getAsSymbol()` or a region extracted with `getAsRegion()` is tainted.

These heuristics significantly simplify taint manipulation in the checkers.

5.7. Understanding debug dumps

All three symbolic value classes allow a convenient `dump()` method useful for debugging. For most of the value kinds, this method produces a recognizable pattern, which can tell a lot of useful information about the value.

Consider an example:

```
reg_$2<element{SymRegion{derived_$1{conj_$0{int},a->ptr}},0 S32b,int}>
```

This `SVal` is a symbol, namely a `SymbolRegionValue`, which is represented by the `reg_$N<...>` wrapper. The number `N` after `$` is the internal symbol counter assigned to each symbol inside the `SymbolManager` object.

This symbol represents the original value of a signed integer in `ElementRegion` with index 0 inside a certain `SymbolicRegion` corresponding to a symbolic pointer. It is uncertain to the analyzer whether this pointer points to an array or to a single integer; however, it is certain that this pointer gets dereferenced in order to obtain the original value.

The pointer itself is a `SymbolDerived`, which is derived for a `FieldRegion` of field `ptr` of some structure variable `a`, from a `SymbolConjured` of type `int`. The `SymbolDerived` itself, being a base for a `SymbolicRegion`, is necessarily a pointer value. However, the type of `SymbolConjured` isn't a pointer; it has most likely appeared as a result of invalidation of structure `a`.

Hence, the value can be described in words as "the value of the integer that was originally stored behind the pointer that appeared in the `a.ptr` field during invalidation".

Consider another example:

```
&base{base{base{d,C},B},A} [as 64 bit integer]
```

This value is a `nonloc::LocAsInteger` that represents a concrete value of a location casted into a 64-bit integer. The location is a `loc::MemRegionVal` (hence prefix '`&`') holding a certain region.

The region itself is the region of a C++ base object of class `A` for an object `d` (which probably belongs to class `D`, and you can check this by dumping the declaration of variable `d`). However, because `D` is not a direct descendant of `A`, you see the whole class inheritance path inside the region hierarchy.

5.8. Further reading

The memory model of Clang Static Analyzer was described in detail in an article by Z. Xu, T. Kremenek, and J. Zhang[13].

[13] Z. Xu, T. Kremenek, and J. Zhang. A memory model for static analysis of C programs. In: ISoLA'10 Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation. pp. 535–548 (2010)

Index of notions

Abstract Interpretation, 26
Abstract Syntax Tree, **13**
AST, *see* Abstract Syntax Tree

CFG, *see* Control Flow Graph
Checker
  AST-based, **18**
    AST Matcher, **21**
    AST Visitor, **19**
  Callback, 18, 35
  Path-sensitive, **26**, 29
Control Flow Graph, **14**, 29
  Basic Block, **14**, 16, 39, 41
  Terminator, **14**, 16, 35, 41

Exploded Graph, **16**, 29
Exploded Node, 16, 29, 47, 48
  Program Point, 16, 47
  Program State, 16, **26**, 26
    Assumption, *see* Range Constraint
    Binding, *see* Region Store
    Environment, **26**, 35, 41, 47, 50
    GDM, *see* Generic Data Map
    Generic Data Map, 26, **32**, 50, 51
    Range Constraint, **16**, 26, 30, 45, 50, 57
    Region Store, 26–28, **31**, 38, 40, 42, 43, 50
    Taint, 26, **29**, 31, 50, 60
    Trait, *see* Generic Data Map
  Sink Node, 11

Garbage Collection, 17, 27, 41–43

Immutability, 26, 30, 33, 56
Interprocedural Analysis, 26, 36, 39, 46, **47**, 54
Invalidation, 43, 45, 48, 59, 61
IPA, *see* Interprocedural Analysis

Symbolic Execution, 26, 29, 31, 39, 50
Symbolic Value, 26, **50**
  Concrete Value, 17, **56**
  Memory Region, 26, 27, **51**
    Base Region, **52**
    Memory Space, **51**, 53
  Symbol, *see* Symbolic Expression
  Symbolic Expression, 16, **57**

Undefined Behavior, 57

Index of classes

AnalysisDeclContext, 19–22
AnalysisManager, 21, 23, 24
  getAnalysisDeclContext(), 21
  getASTContext(), 23, 24
AST_MATCHER(), 24
ASTContext, **23**, 23, 24, 29, 36
  getSizeType(), 60
  Idents, 36

BugReport, **11**, 47
BugReporter, 19, 21, 23, 24, 39, 47
  EmitBasicReport(), 20, 22
  getSourceManager(), 20, 22
BugReporterContext, 47
BugReporterVisitor, **47**
  VisitNode(), **47**
BugType, 11

CallEvent, 11, **36**, 36
  getArgSVal(), 36
  getCalleeIdentifier(), 11, 34, 36
  getNumArgs(), 36
  getReturnValue(), 31, 36
  isGlobalCFunction(), 36
Checker, **10**
  check::ASTCodeBody, **18**, 21, 24
  check::ASTDecl, **19**
  check::Bind, **38**, 43
  check::BranchCondition, **41**, 45
  check::DeadSymbols, **42**, 45
  check::EndAnalysis, **39**
  check::EndFunction, 26, **39**, 49
  check::EndOfTranslationUnit, **18**, 23
  check::LiveSymbols, **41**, 60
  check::Location, **37**, 43
  check::PointerEscape, **45**, 48
  check::PostCall, **36**, 46
  check::PostStmt, **35**
  check::PreCall, 10, **36**, 46
  check::PreStmt, **35**
  check::RegionChanges, **43**, 48
  eval::Assume, **45**
  eval::Call, **46**, 48, 51
  getTag(), **60**
CheckerContext, **11**, 26, 30, 39
  addTransition(), **30**, 31, 32, 43, 46
  blockCount(), 60
  emitReport(), 11
  generateSink(), 11, 37
  getASTContext(), 36
  getCalleeDecl(), 46
  getLocationContext(), 27, 35, 40, 41, 46
  getStackFrame(), **48**
  getState(), **26**, 29, 31, 32
  getStoreManager(), 28, 40
  getSValBuilder(), 29, 38, 60
  inTopFrame(), **48**
CheckerManager, 11
  registerChecker, 11
CheckerRegistry, **12**
  addChecker(), 12
clang_analyzerAPIVersionString, **12**
clang_registerCheckers(), **12**
ConstDeclVisitor, **19**, 21
ConstraintManager, 46
  isNull(), 46
ConstStmtVisitor, **19**, 20, 21

DeclarationMatcher, 23

ExplodedGraph, 16, 39
ExplodedNode, 16, 47
ExprEngine, 39
  hasWorkRemaining(), 39

LivenessAnalysis, 15
LocationContext, 26, 27, 48
  getCurrentStackFrame(), 40
  StackFrameContext, 40, **48**, 53

MatchFinder, **22**
  match(), **24**
  matchAST(), **23**, 24
  MatchCallback, **22**
  MatchResult, 22
    Nodes, 22
MemRegion, 27, 40, 50
  AllocaRegion, **53**
  BlockDataRegion, **55**
  BlockTextRegion, **55**
  CodeTextRegion, **55**
  CompoundLiteralRegion, **54**
  CXXBaseObjectRegion, 52, **55**
  CXXTempObjectRegion, **54**
  CXXThisRegion, **54**
  ElementRegion, 52, **55**, 60, 61
  FieldRegion, 52, **55**, 61

FunctionTextRegion, **55**
getMemorySpace(), 40, **52**
getSymbolicBase(), **53**
GlobalImmutableSpaceRegion, **53**
GlobalInternalSpaceRegion, **53**
GlobalInternalSpaceRegionKind, 54
GlobalsSpaceRegion, 40, **53**
GlobalSystemSpaceRegion, **53**
HeapSpaceRegion, **53**, 54
MemSpaceRegion, **53**
NonStaticGlobalSpaceRegion, **53**
StackArgumentsSpaceRegion, **53**, 55
StackLocalsSpaceRegion, **53**, 53, 54
StackSpaceRegion, 40, **53**, 54
getStackFrame(), 40
StaticGlobalSpaceRegion, **53**
StringRegion, **54**
SubRegion
getBase(), **52**
getExtent(), **51**, 59
getSuperRegion(), **51**
SymbolicRegion, 51, 52, **53**, 55, 60, 61
TypedValueRegion, 38, 54
getValueType(), 38
UnknownSpaceRegion, **53**, 54, 56
VarRegion, 49, 52, **54**, 55
MemRegionManager, **51**, 54

ParentMap, 37
PathDiagnosticEventPiece, 47
PathDiagnosticLocation, 20, 22, 47
createBegin(), 20, 22
getStmt(), 47
ProgramPoint, 16
ProgramState, 16, **26**, 39, 56, 57
add<>(), **33**, 34
addTaint(), **31**
assume(), 28, **30**, 31, 37, 57
contains<>(), **33**, 34
get<>(), **33**, 42–44, 46
getConstraintManager(), 46
getStore(), 28
getSVal(), **27**, 27, 35, 41, 46, 57
isTainted(), **29**
remove<>(), **33**, 43, 45, 46
set<>(), **33**, 44
ProgramStateRef, **26**, 30

RangeConstraintManager, 58
REGISTER_LIST_WITH_PROGRAMSTATE, **33**
REGISTER_MAP_WITH_PROGRAMSTATE, **34**
REGISTER_SET_WITH_PROGRAMSTATE, **33**, 34
REGISTER_TRAIT_WITH_PROGRAMSTATE, **33**

SourceManager, 20, 22
StatementMatcher, 23
Store, 28, 40, 56
StoreManager, **28**, 40
BindingsHandler, **28**, 40
iterBindings(), **28**, 40
SVal, 27, **50**
castAs<>(), 37, 56
DefinedOrUnknownSVal, 28, 31, 37
DefinedSVal, 38
getAs<>(), 28, 31, 37, 38
getAsRegion(), 27, 38, 40, **51**, 60
getAsSymbol(), 36, 51, **58**, 60
isUndef(), 37, 41
Loc, **50**
loc::ConcreteInt, **56**
loc::MemRegionVal, **51**, 58, 61
NonLoc, **50**
nonloc::CompoundVal, **56**
nonloc::ConcreteInt, **56**
nonloc::LazyCompoundVal, **56**
nonloc::LocAsInteger, **51**, 58, 61
nonloc::SymbolVal, 51, **58**
symbol_begin(), 42
symbol_end(), 42
UndefinedVal, 37, **57**, 59
UnknownVal, **57**
SValBuilder, 29, 38, **51**, 51
evalBinOp(), 29, 38
getContext(), 60
getMetadataSymbolVal(), **60**
makeIntVal(), 38
SymbolManager, **51**, 61
SymbolReaper, **41**
isDead(), 43
markInUse(), 42
SymExpr, 16, 50
IntSymExpr, **58**
SymbolConjured, 51, 58, **59**, 61
SymbolDerived, **59**, 60, 61
SymbolExtent, **59**
SymbolMetadata, 41, 51, **60**
SymbolRegionValue, 16, 49, 52, 54, **59**, 60, 61
SymIntExpr, **58**
SymSymExpr, **58**

TypeMatcher, 23