Optimizing for Bugs Fixed

The Design Principles behind the Clang Static Analyzer

Anna Zaks, Manager of Program Analysis Team @ Apple

What is This Talk About?

- LLVM/clang project
- Overview of the Clang Static Analyzer
- Driving factors behind its design
- The story of ARC
- Program Analysis vs Language Evolution

LLVM/clang Project

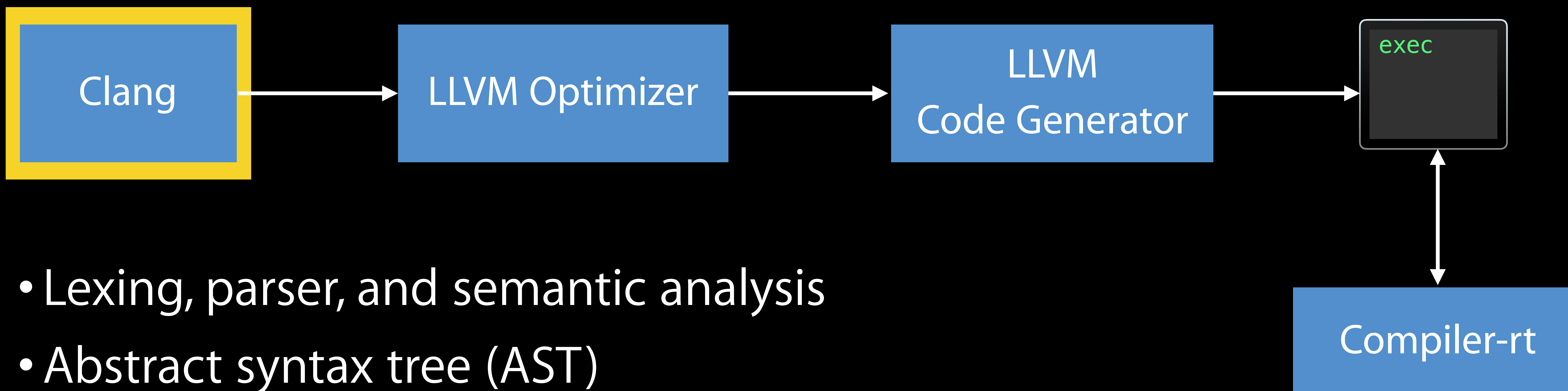
LLVM/clang is a Great Compiler

- Compiler for C/C++/Objective-C
- Great diagnostics
- Open source

LLVM is Much More than a Great Compiler!

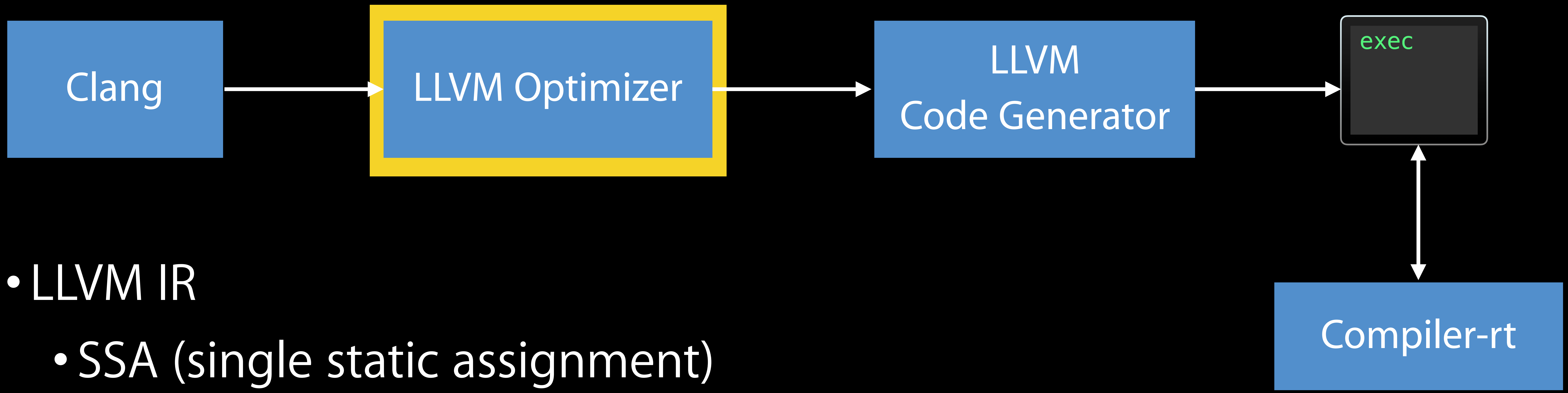
- Extensible modular infrastructure
- Great platform for building tools and analysis
- Including bug finding tools!
- Widely used by academic projects
- Over 300 papers published

Overview of the Main Components



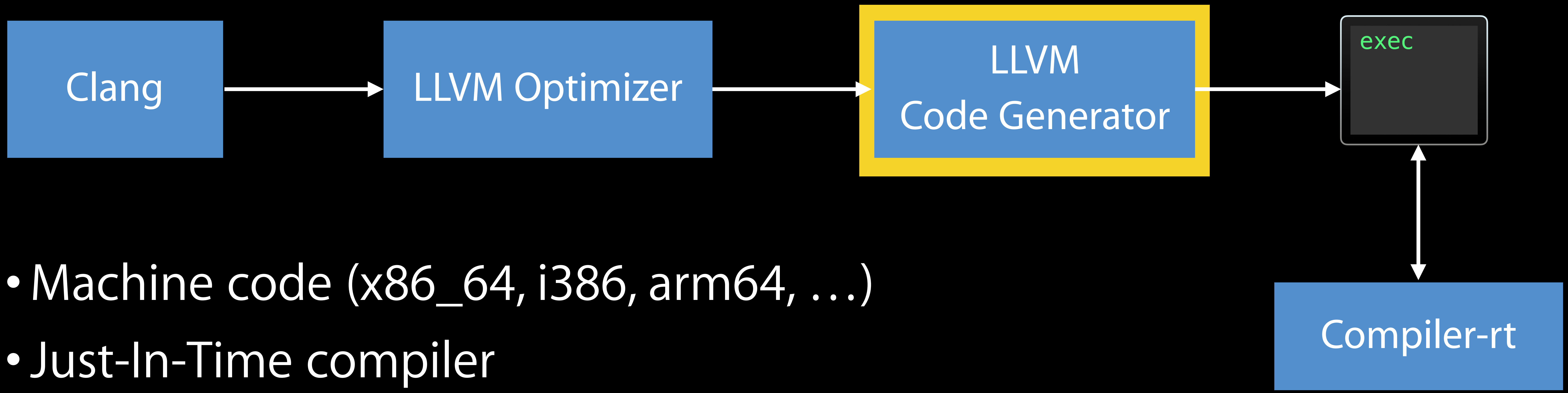
- Lexing, parser, and semantic analysis
- Abstract syntax tree (AST)
    - Code indexing and cross-references
    - Code completion
- Source-level control flow graph (CFG)
    - Clang Static Analyzer

Overview of the Main Components



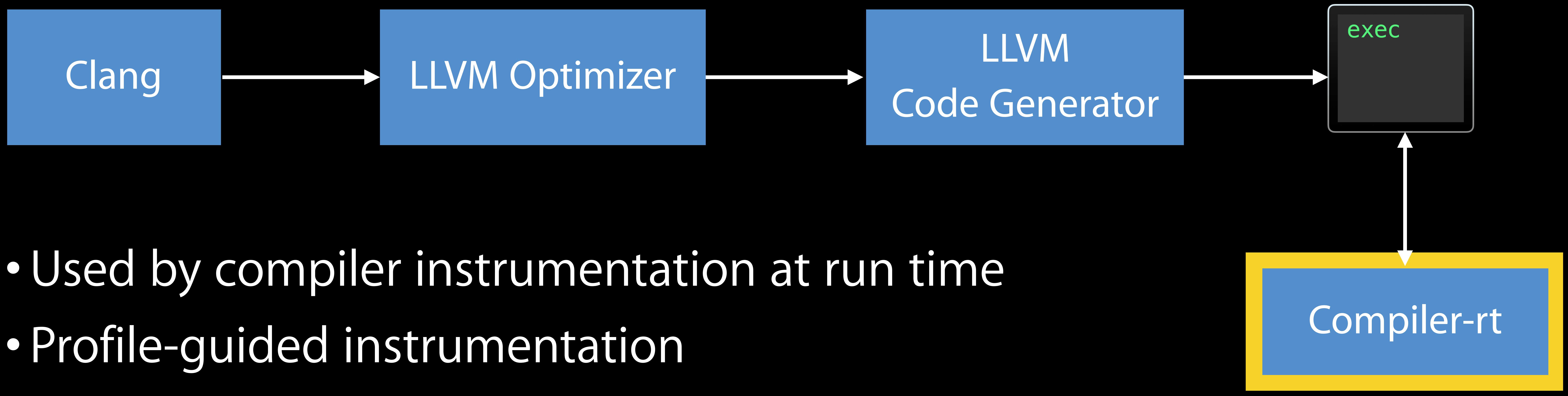
- LLVM IR
  - SSA (single static assignment)
  - CFG (control flow graph)
  - Access to analysis such as dominance, loops
- Analysis, transformation, and optimization passes

Overview of the Main Components



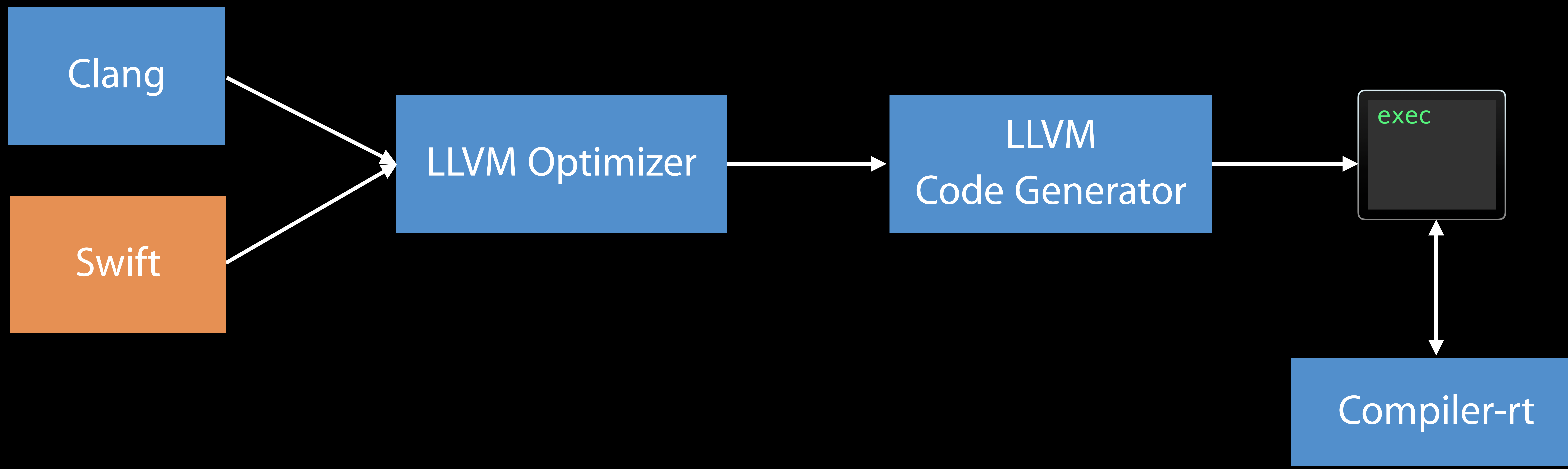
- Machine code (x86_64, i386, arm64, …)
- Just-In-Time compiler

Overview of the Main Components



- Used by compiler instrumentation at run time
- Profile-guided instrumentation
- Support for run-time bug-finding tools
  - Address Sanitizer
  - Thread Sanitizer

Swift Compiler



Finding Bugs with LLVM

- Compiler Warnings
  - Explore with `-Weverything`
- clang-tidy allows pattern matching on the AST
  - Great linter
  - Coding standards (CERT, LLVM, Google)
- Sanitizers
  - Find bugs at run time with low overhead
  - Provide great diagnostics
  - Need to execute the code path that triggers the problem
- Clang Static Analyzer
  - Deeper path-sensitive analysis

Clang Static Analyzer

Clang Static Analyzer

- Works with C/C++/Objective-C
- Extensive set of Objective-C checks
- Integrated into Xcode IDE
- Widely used by iOS/macOS developers

Path-Sensitive Static Analysis

- Finds bugs without running program or tests
- Detects bugs that go undetected during testing
- Finds corner-case, hard to reproduce bugs

Check that a File is Closed on *each* Path

```c
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Check that a File is Closed on *each* Path

```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;        Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Check that a File is Closed on *each* Path

```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)                    1. Assuming 'Data' is null
            return;                   2. Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Symbolic Execution

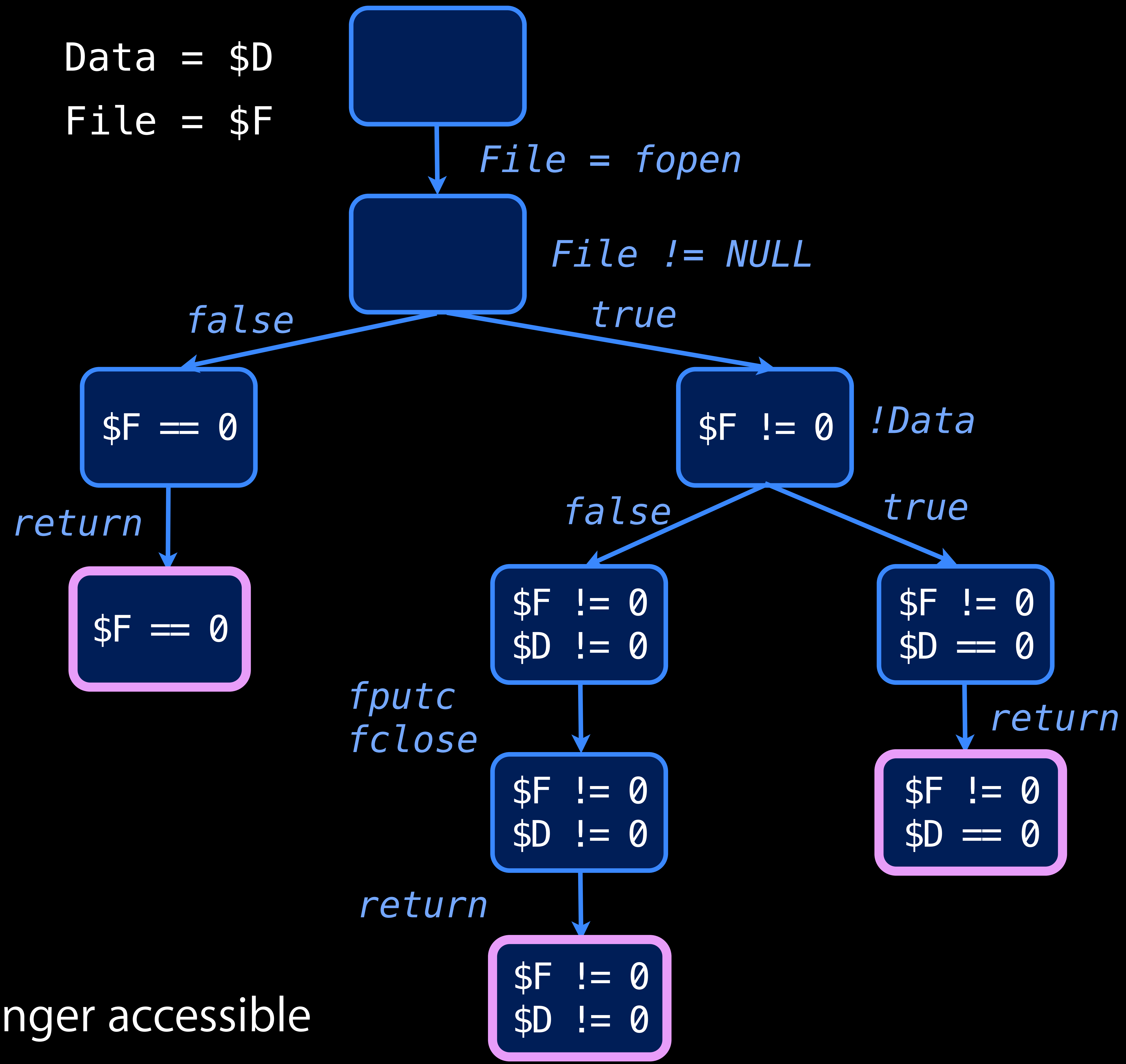
- Simulates execution of all paths through the program
- Uses symbols instead of the concrete values
- Collects the constraints on symbolic values along each path
- Uses constraints to determine feasibility of paths
- Computes a set of reachable program states

Inspired by academic work on symbolic execution and graph reachability:

- *James C. King* Symbolic execution and program testing 1976
- *Thomas Reps, Susan Horwitz, Mooly Sagiv*. Precise interprocedural dataflow analysis via graph reachability 1995

Builds a Graph of Reachable Program States

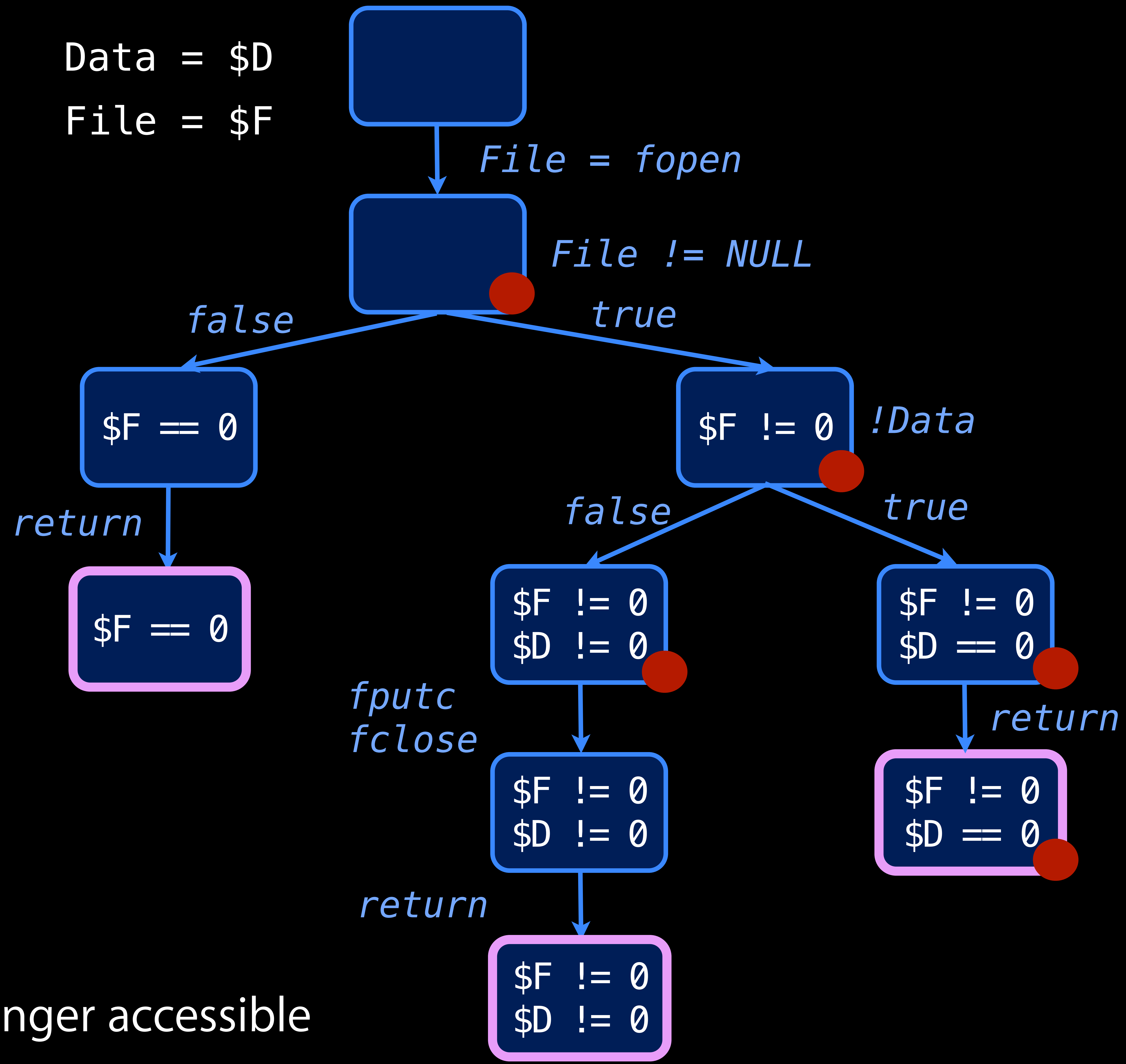
```
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```



Denotes the node where File is no longer accessible

Builds a Graph of Reachable Program States

```c
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```

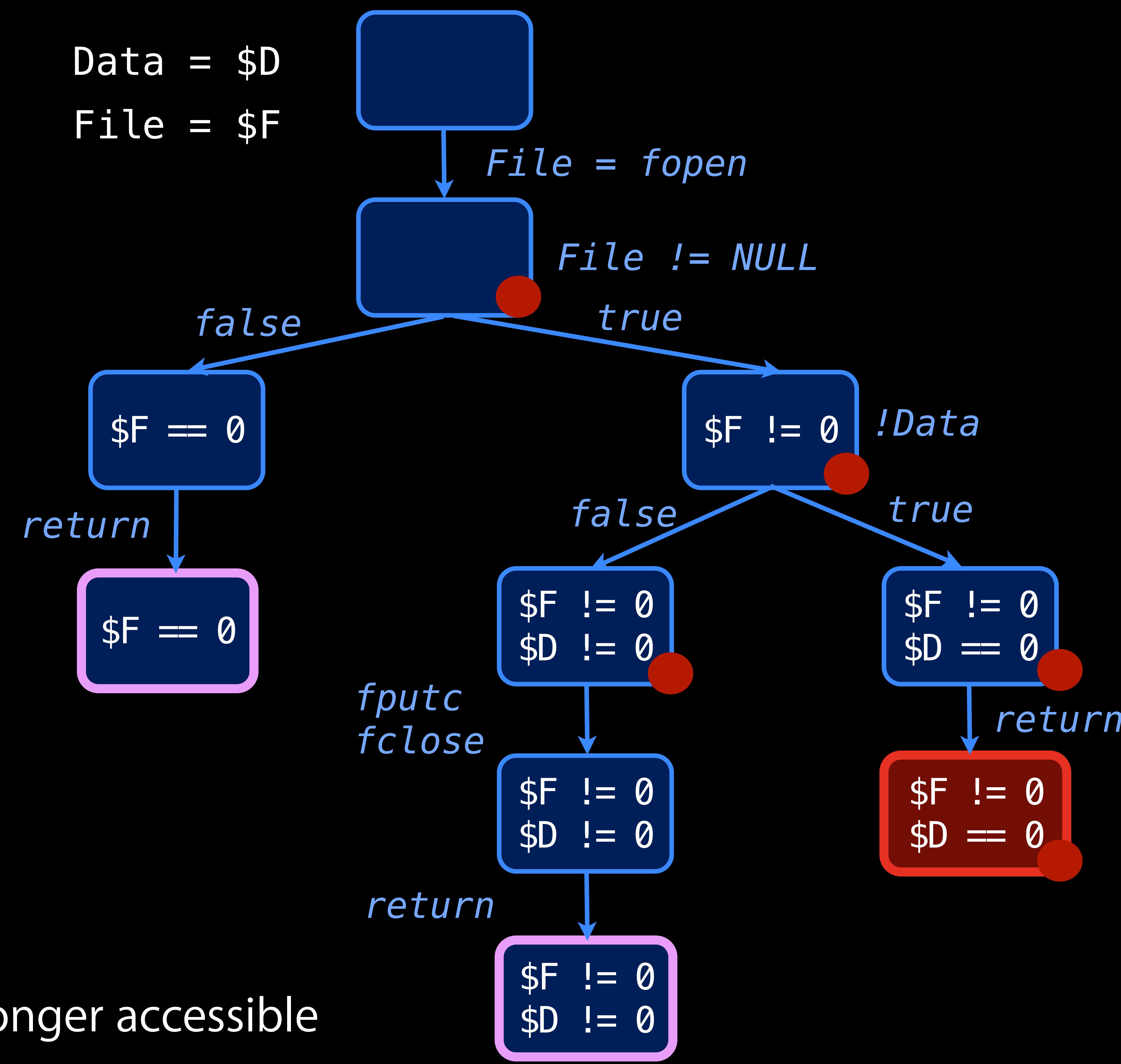


Denotes that the file is open

Denotes the node where File is no longer accessible

Finding a Bug ~ Graph Reachability

```
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```

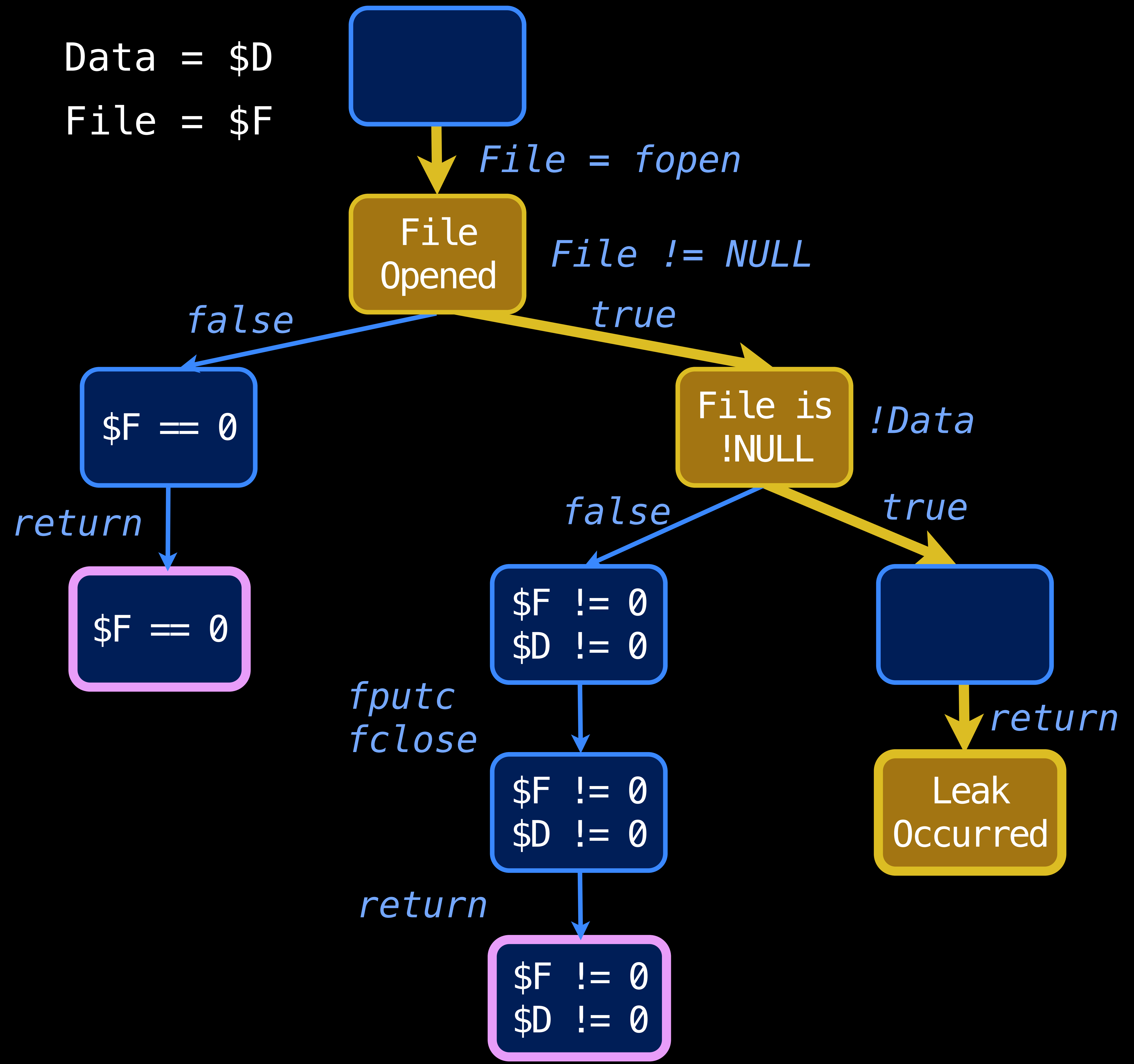


Denotes that the file is open

Denotes the node where File is no longer accessible

Collect Interesting Events for Error Report

```
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```



What's in a Node?

Program Point | Program State

- Execution location
  - pre-statement
  - post-statement
  - entering a call
  - ...
- Stack frame

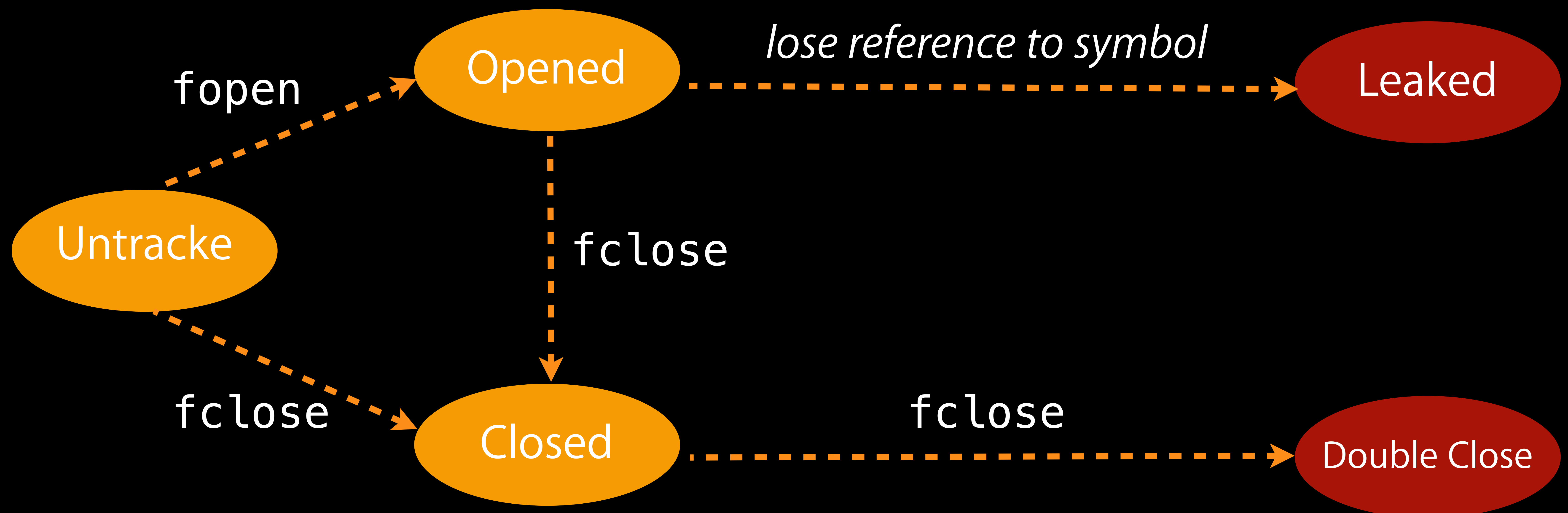
- Environment: `Expr` -> values
- Store: memory location -> values
- Constraints on symbolic values
- Check-specific state

Checkers

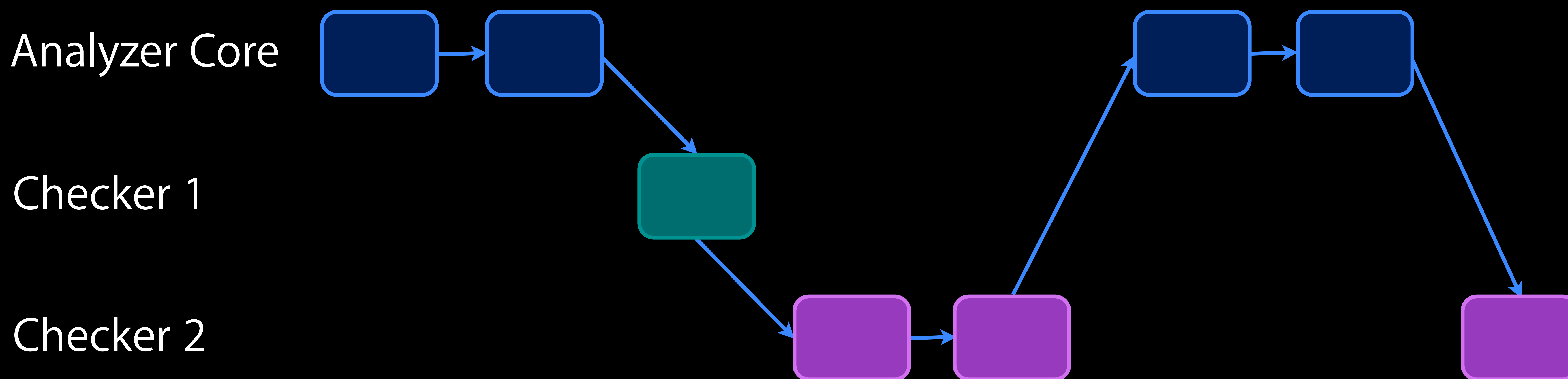
- Related set of correctness rules are checked by a “checker”
- Checkers model the rules (state machines)
- Checkers report errors

File Stream State Transitions

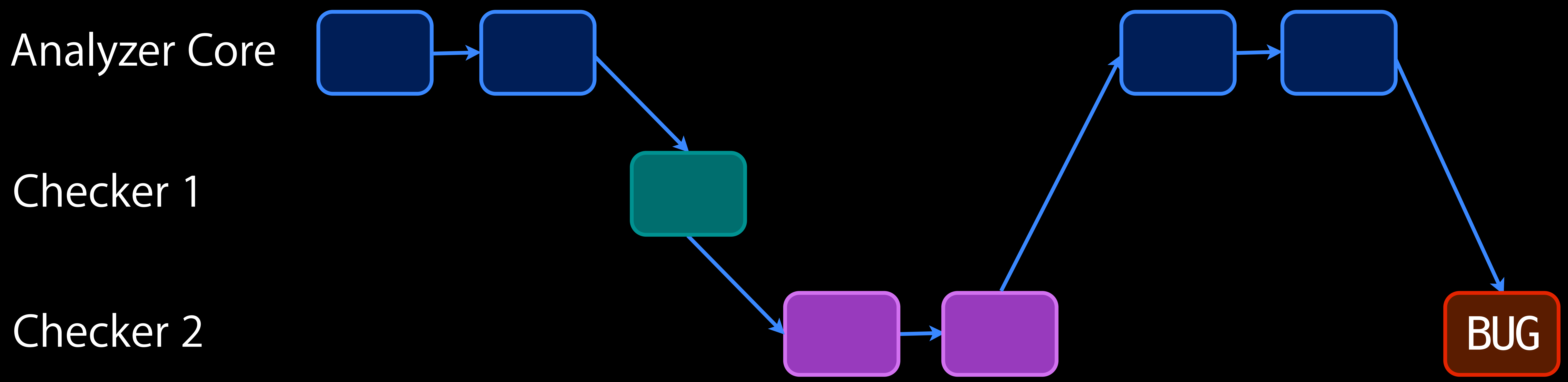
- File handle state changes are driven by the API calls
- Error States:
  - If a file has been closed, it should not be accessed again.
  - If a file was opened with `fopen`, it must be closed with `fclose`



Checkers Participate in the Graph Construction



Exploration Can Stop when a Bug is Reported



Checkers

- Often require check-specific knowledge
- Many checker writers are not static analysis experts
- Solid checker APIs

Checkers are Visitors

```
checkPreStmt(const ReturnStmt *S, CheckerContext &C) const
```

Before return statement is processed

```
checkPostCall(const CallEvent &Call, CheckerContext &C) const
```

After a call has been processed

```
checkBind(SVal L, SVal R, const Stmt *S, CheckerContext &C) const
```

On binding of a value to a location as a result of processing the statement

See the checker writer page for more details:
http://clang-analyzer.llvm.org/checker_dev_manual.html

Current Limitations

- Very simple but super fast range-based constraint solver
  - No bitwise operations (`$F & 0x10`)
  - No constraints involving multiple symbols (`$X > $Y`)
- Analysis are inter-procedural, but not (yet) cross-translation-unit
- No loop invariant inference

- Still very effective used by a huge number of developers!

Optimizing for Bugs Fixed

Driving Factors Behind the Design

Design Goals

The goal is to not find the most bugs
but to make software better!

Design Goals

- Easy access is essential
  - Lightweight enough to run on a laptop
  - Integrated into developer workflows
- Finding a bug is only one part of the problem
  - Explain the bug so the developer knows how to fix it
- Secret sauce

Lightweight Enough to Run on a Laptop

- Memory optimizations
  - Very simple and fast solver
  - Persistent data structures for maximizing sharing
  - Pruning of unused nodes
  - Lazy evaluation of complex expressions
- Many projects turn on analysis during build!

Integrated into Developer Workflows

- IDE: analysis is available in the editor
- Automation: analysis runs as part of continuous integration
- Code reviews: incremental analysis on every commit (opportunity for improvement)

Explaining the Bugs

Static analysis power acts as a double-edged sword when it’s time to explain the results to the user

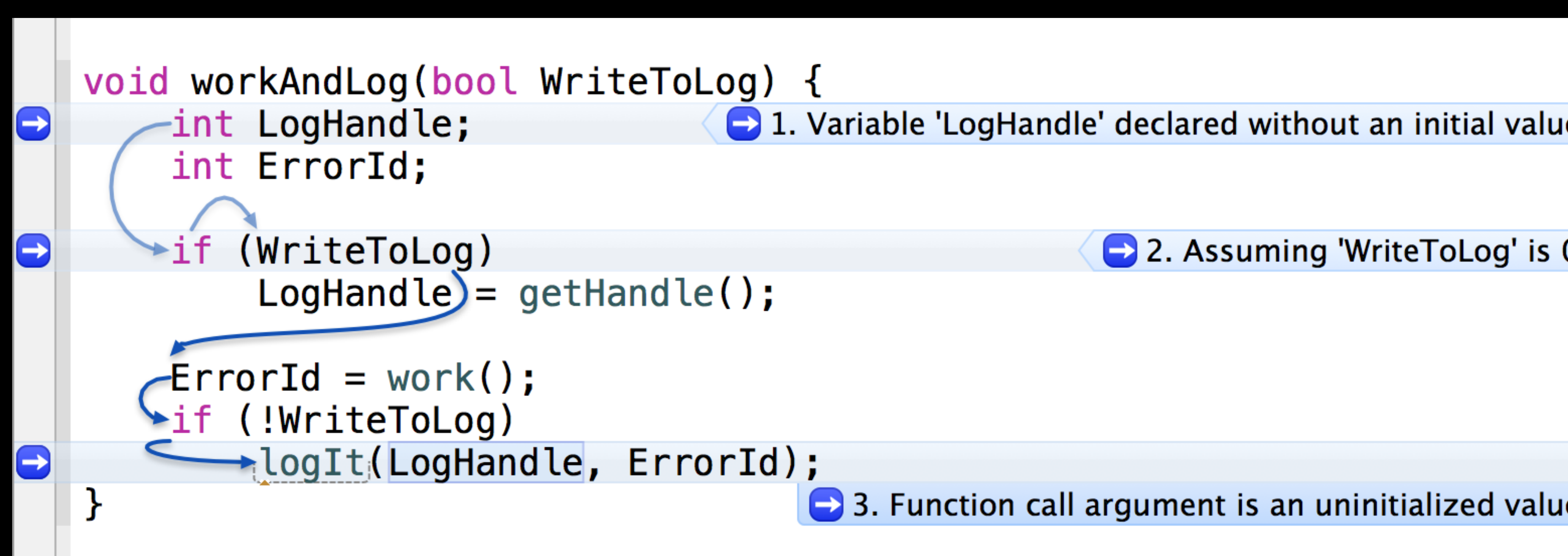
Explaining the Bugs: Visualizing the “Paths”

```
void workAndLog(bool WriteToLog) {
    int LogHandle;
    int ErrorId;

    if (WriteToLog)
        LogHandle = getHandle();

    ErrorId = work();
    if (!WriteToLog)
        logIt(LogHandle, ErrorId);
}
```

1. Variable 'LogHandle' declared without an initial value
2. Assuming 'WriteToLog' is 0
3. Function call argument is an uninitialized value

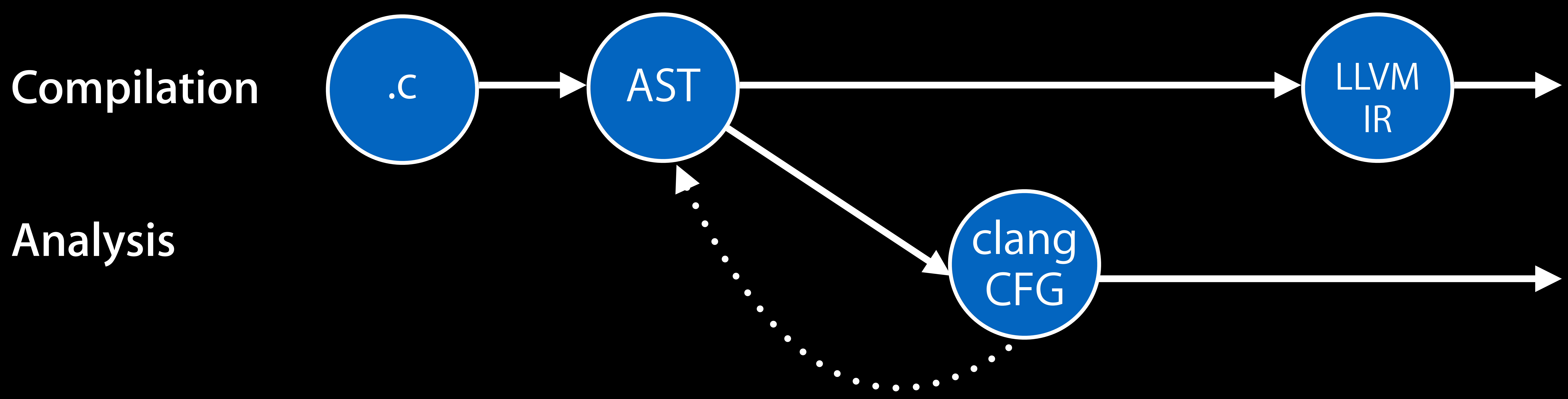
Explaining the Bugs

- Checker APIs for report-specific information along the path
- Highlight only relevant points along the path
- Determine beginning and end of each arrow
- Source-level analysis allows for precise source location information



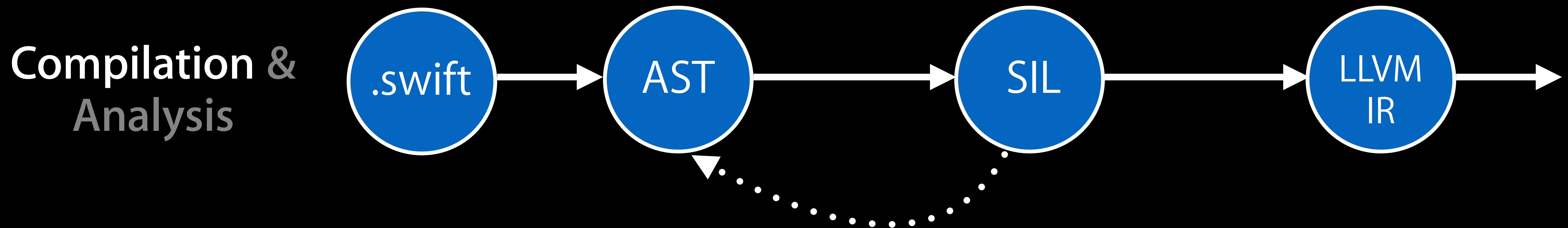
Tradeoff: Clang CFG vs. LLVM IR

- Pros:
  - Allows very precise source locations
- Cons:
  - clang CFG is off the beaten path, harder to maintain
  - Need to model every AST node (C/C++ is much larger than LLVM IR)



Better Tradeoff: Higher-Level IR

- Swift compiler provides a better solution with SIL
  - Intermediate representation with links to AST (source locations)!
  - Used by compilation
- (Currently there is no path-sensitive static analysis for Swift)



Swift's High-Level IR: A Case Study of Complementing LLVM IR with Language-Specific Optimization http://llvm.org/devmtg/2015-10/#talk7

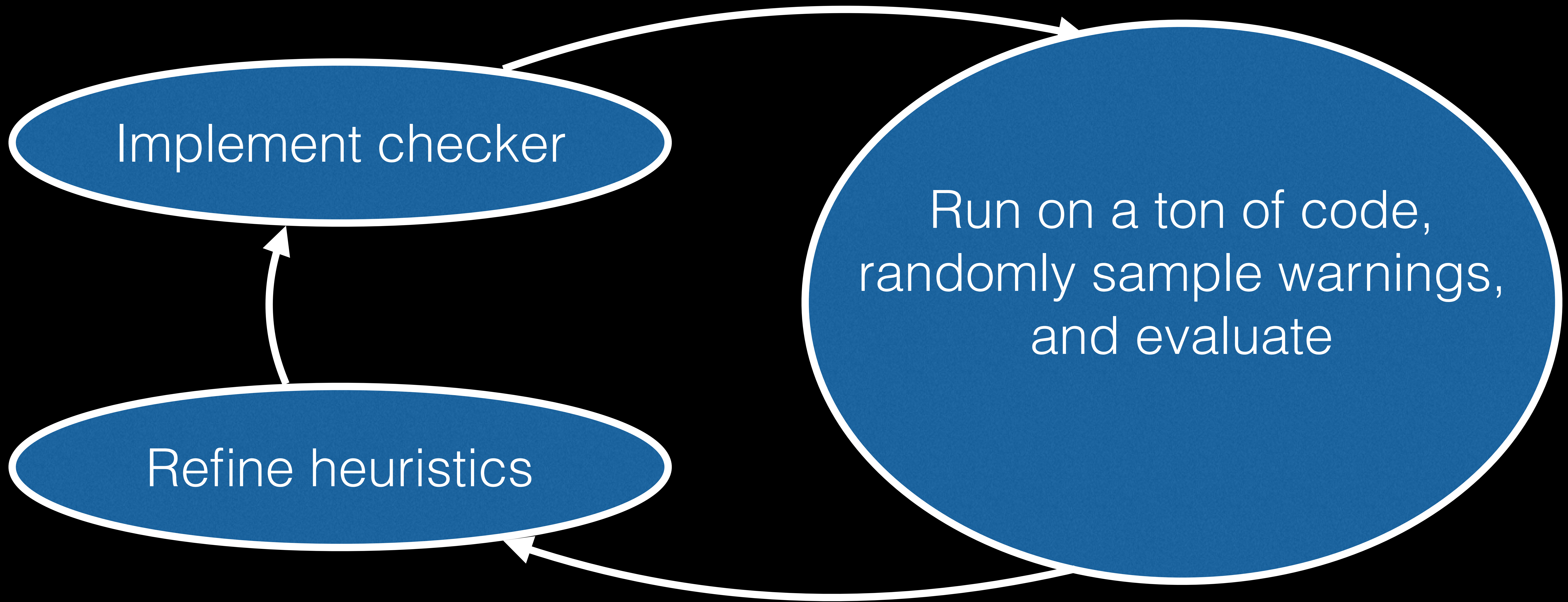
Secret Sauce

Our secret sauce is the art of checker rollout

The Key Ingredient

- Developers rely on code patterns that we do not know about!
- Common checker writer mistakes:
  - Check for rules that are too pedantic
  - Don’t account for all corner cases
  - Do not provide clear diagnostics

Recipe is Simple though Time Consuming

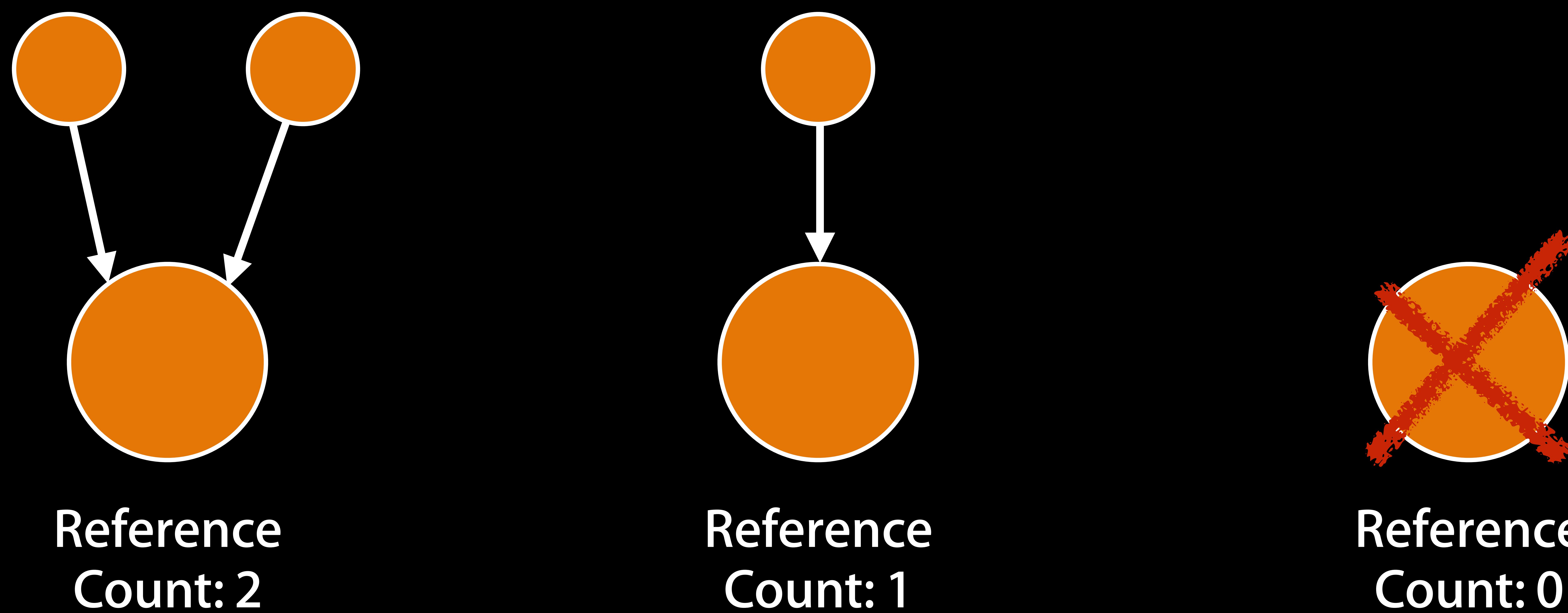


The Story of Automated Reference Counting

Static Analysis Facilitating Language Evolution

Object Ownership in Objective-C

Retain/Release (Reference-Counting)



If you claim ownership for the object, you have to release it when done

Manual Retain/Release in Objective-C

```
- (void)planEvening:(List *)dinnerList {
    Person *aPerson = [[Person alloc] init];
    [dinnerList add:aPerson];
    [aPerson release];
}
```

- Easier to use when following these conventions:
    - Most methods keep the reference count unchanged
    - If method breaks the rule, it indicates that using naming conventions

Manual Retain/Release in Objective-C

Still Error Prone

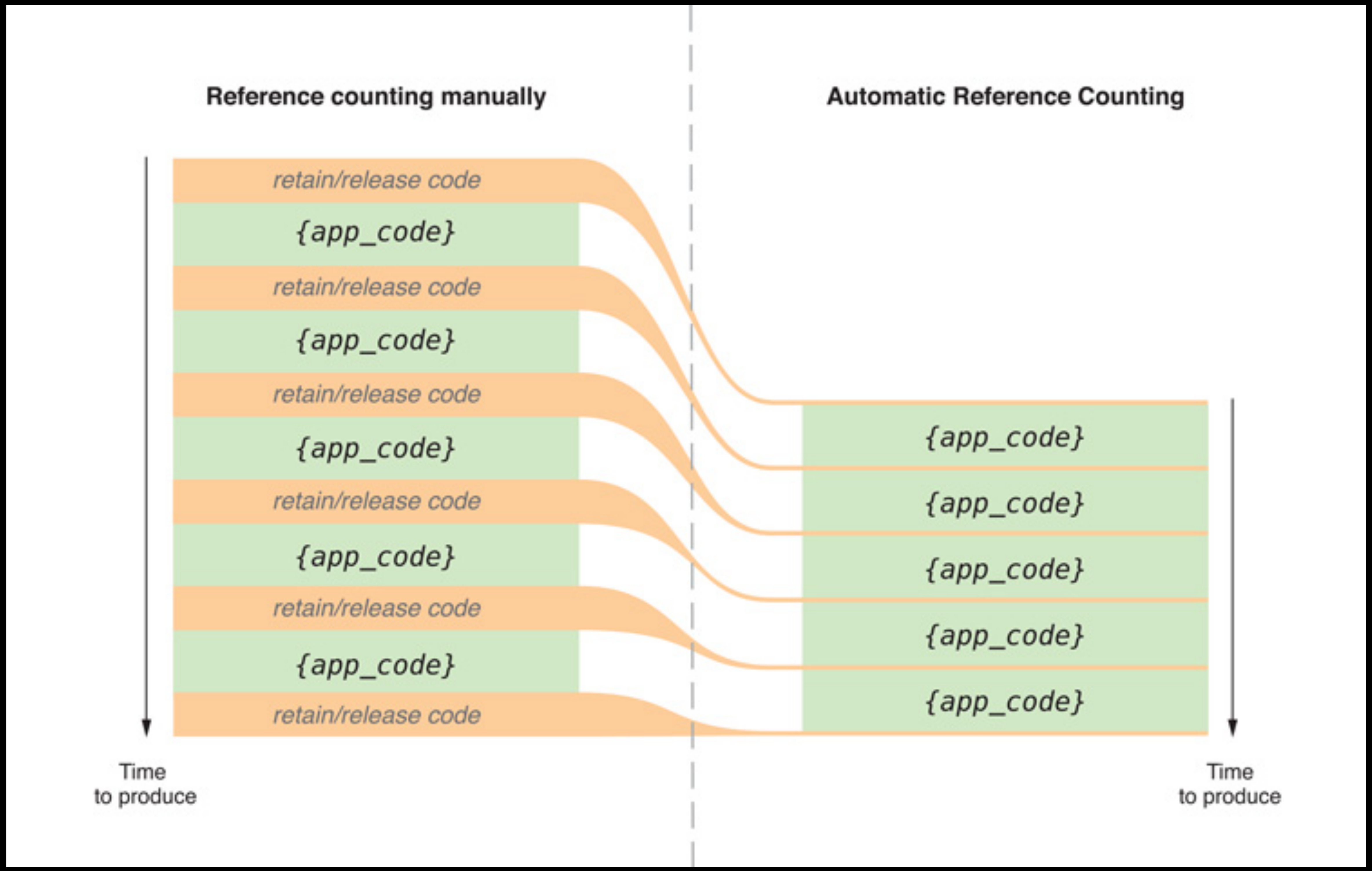
```
- (void)planEvening:(List *)dinnerList with:(List *)stayUpLateList {
    Person *aPerson = [[Person alloc] init];
    [dinnerList add:aPerson];
    if (!aPerson.isChild)
        return;
    [stayUpLateList add:aPerson];
    [aPerson release];
}
```

The object 'aPerson' is leaked

Ideal Problem for Static Analysis

- Finds leaks and use-after-frees in Objective-C code
- Strong API contracts allows for local reasoning
- Can be checked with intra-procedural static analysis
- Checker was very popular
  - Proved that people are willing to change their code
  - Exposed non-conforming APIs that need to change

Inspiring a Language Solution



- For both Objective-C (2012) and Swift
- Automated Reference Counting is now the default

Program Analysis vs Language Evolution

Symbiotic Relationship

Swift Programming Language

Swift Programming Language

- A general-purpose language, expressive, fast and …
- Safe
  - Automatic memory management
  - Definite initialization of variables
  - Optional types

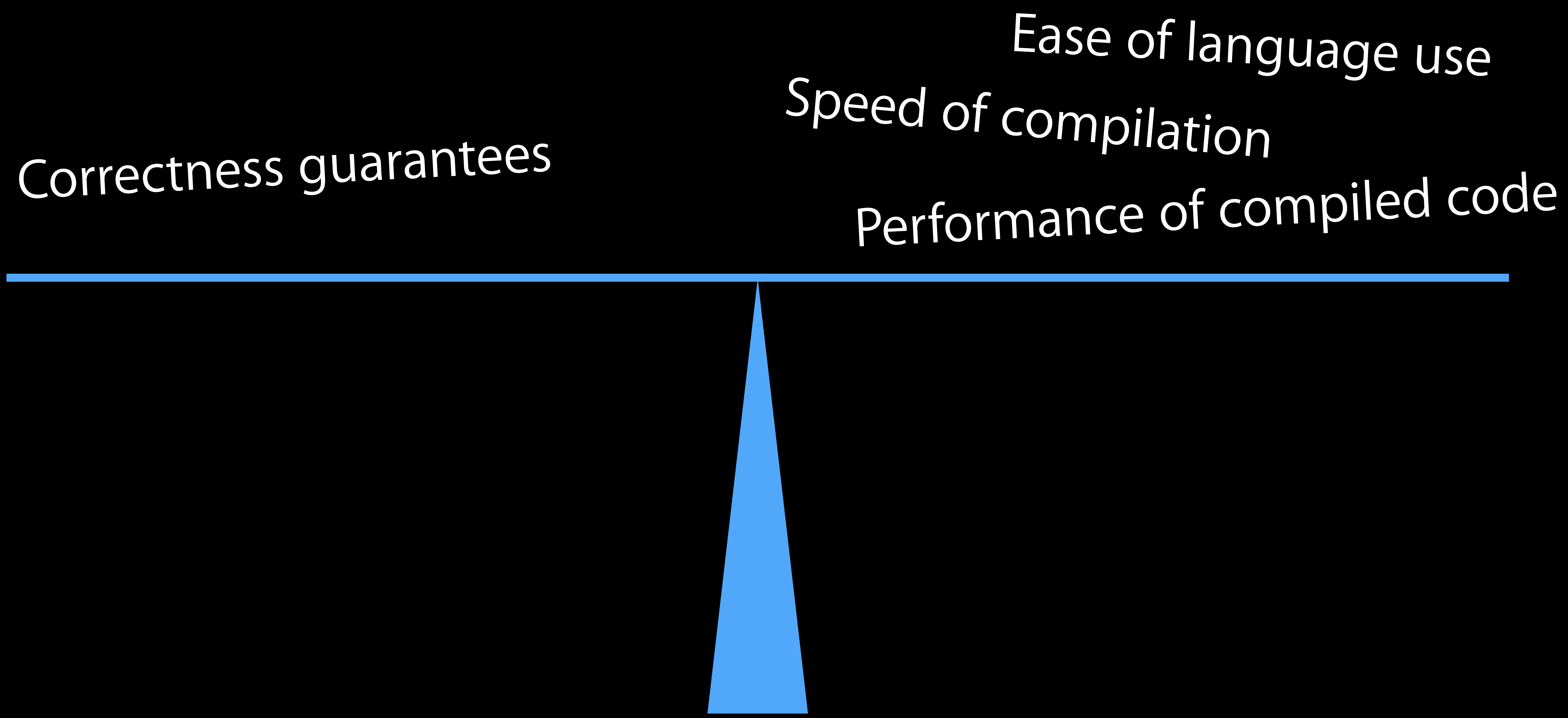
Strengthening the Language is the Best

- Usually stricter rules
- Stronger guarantees
- No need to use an additional tool - everyone runs the compiler!

Program Analysis vs Language Evolution

Do we still need program analysis?

Language Design is about Tradeoffs



Type System Can’t Solve all our Problems

- Check if index is in bounds of an array
- Termination and liveness (LTL/CTL)
- Information flow & taint analysis
- API contracts

Symbiotic Relationship

Language advancements and program analysis work best together!

Conclusion

Focus on the Bigger Picture

Optimize for Bugs Fixed

- More expressive and intuitive way of writing checks
- Developer workflows
- Error reporting
- Interplay with language features

Open Source

- llvm.org
    - Ask questions on mailing lists
    - Read patches/developer lists
- swift.org
    - Language evolution list
    - “Newbie” bugs
- clang-analyzer.llvm.org