SWAN: A Static Analysis Framework for Swift

by

Daniil Tiganov

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Swift is an open-source programming language and Apple's recommended choice for app development. Given the global widespread use of Apple devices, the ability to analyze Swift programs has significant impact on millions of users. Although static analysis frameworks exist for various computing platforms, there is a lack of comparable tools for Swift. Existing Swift static analysis tools are either incomplete, use dynamic analysis, or are otherwise not suitable for deeper analyses of Swift programs such as taint tracking. Moreover, other existing tools for Swift only help enforce code styles and best practices.

In this thesis, we present SWAN, an open-source and configurable framework that allows robust program analyses of Swift programs. SWAN features a suite of call-graph construction algorithms, support for modelling black-box functions using its own internal representation, and can track dataflow through libraries. The framework is also capable of traditional taint analysis, typestate analysis, and detecting security vulnerabilities, such as cryptographic API misuses. We demonstrate the framework's robustness by evaluating its core framework runtime performance, call-graph construction precision and performance, and ability to find cryptographic API misuses in real Swift applications. For most of our benchmarks, SWAN parses, translates, and prepares the program for analysis in under 5 seconds and builds a precise call-graph in less than 13 seconds. SWAN also finds 43 real cryptographic API misuses across 13 applications that we tested.

Preface

This thesis is original work by Daniil Tiganov, with the following exceptions. Chapter 4 and Section 6.2 were joint work with Ifaz Kabir with roughly equal contribution. Shivani Kadel and Guoze Feng assisted with manually writing models for Swift's standard library containers (Section 3.6) and corresponding tests (Section 6.4.1). The work described in this thesis is subject to publication. An old version of the framework described in this work was previously published as D. Tiganov *et al.*, "Swan: A static analysis framework for swift," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1640–1644. Despite sharing the same same title, this thesis describes entirely new work that is only conceptually similar with the published work. With the aforementioned exceptions, I was responsible for the conceptualization and implementation of the framework described in this thesis. I was also responsible for the experimental setup and evaluation of the framework. Karim Ali was the supervisory author of this work and oversaw the development of the framework. Karim also contributed to this work's conceptualization, requirements, empirical evaluation design, and thesis edits.

Acknowledgements

I thank the Lord my God the Most High for His everlasting love and mercy. I thank Him for the blessing of education and the opportunity to pursue graduate studies. I thank the Lord for enabling me to continue my studies, for I can do nothing without Him.

Thank you, Madeleine, my beautiful wife, whom I adore and love, for encouraging me throughout my studies. Your never-ceasing positivity and love ease my burdens. I am very blessed to have you as my wife and as the mother of my beautiful daughter, Amalia, whom I cherish and for whom I am exceedingly grateful to God.

Thank you to my parents for instilling in me an appreciation for education and the pursuit of knowledge. Your encouragement and support helped me and motivated me to continue my studies.

Thank you to my colleagues at the Maple Lab, with whom it was a pleasure to work. It was great being among people with similar interests. I will miss being a part of the lab, and I wish the Maple Lab success and all the best.

Thank you to Dr. Karim Ali, my supervisor, for your patience, enthusiasm, generosity, and guidance. Thank you for your high standards and attention to detail. You have cultivated my passion for static program analysis throughout the years. I especially appreciate that you greatly care about your students. You have given me the knowledge I need to be successful in my career.

Contents

List of Tables

List of Figures

List of Acronyms

API
Application Programming Interface

AST
abstract syntax tree

CFG
control-flow graph

CG
call-graph

CHA
Class Hierarchy Analysis

CLI
Command-Line Interface

DDG
Dynamic Dispatch Graph

DSL
domain-specific language

FSM
finite-state machine

IDE
Interprocedural Distributive Environments [43]

IDE
integrated development environment

IFDS
Interprocedural Finite Distributive Subset [43]

IR
intermediate representation

IV

initialization vector

LOC

lines of code

OOL

object-oriented language

PBE

password-based encryption

PDS

pushdown system

RTA

Rapid Type Analysis [12]

SAST

static application security testing

SIL

Swift Intermediate Language. Apple's intermediate language that it translates Swift into and then later translates to LLVM [5].

SPDS

Synchronized pushdown systems. A novel, on-demand, and highly precise data flow analysis written by Späth et al. [48].

SPM

Swift Package Manager. Apple's alternative project management system to Xcode projects [3].

SSA

Static single assignment form [17].

SWIRL

SWAN InteRmediate Language. $\textsc{Swirl}_{\text{RAW}}$ is SWIRL's raw form, and $\textsc{Swirl}_{\text{CAN}}$ is SWIRL's canonical form.

UI

user interface

VTA

Variable Type Analysis [51]

WPDS

Weighted pushdown systems. An adaptation of synchronized pushdown systems (SPDS) where each rule has additional weights [48].

Glossary of Terms

Boomerang
: A SPDS query engine for making forward and backward queries.

IDE^{al}
: A SPDS query engine for typestate analysis.

regex
: A regular expression that specifies a text search pattern.

UCG
: SWAN's unique call-graph construction algorithm that simultaneously and precisely handles functions pointers and dynamic dispatch.

Chapter 1

Introduction

Static program analysis reasons about the potential runtime behaviour of a program without necessarily executing it. This technique may be used to detect various types of defects [10], from simple bugs to security vulnerabilities, optimize applications [14], and help protect user privacy [8]. Despite the potential benefits of static analysis, there is a lack of available tools for Swift [4], Apple's recommended and most popular [31] choice for development on iOS [1] and macOS [2]. In 2022, the web traffic analysis tool *StatCounter* estimated that iOS devices comprised approximately 27% of mobile devices in the world [50] and macOS devices accounted for 15% of desktop devices [49]. Therefore, the ability to analyze Swift applications has significant impact on millions of users around the world.

Commercial static analysis tools that support Swift (e.g., Coverity [53] and Checkmarx [16]) are impractical for academics and practitioners due to their high cost and closed-source nature. Open-source tools that target the Swift abstract syntax tree (AST) are mostly linters or simple checkers (e.g., SwiftLint [52]) that do not compute call graphs. GitHub's CodeQL [26] is the first commercial[1] open-source framework that has (begun to add) Swift support. However, at the time of writing, GitHub has not yet officially released Swift support for CodeQL. Moreover, in our testing, we were not able to analyze Swift Xcode applications using CodeQL's Swift support.

To bridge the gap between the popularity of Swift and the lack of available

[1]CodeQL is open-source and free for research, but is not free for commercial customers.

analysis tools, we introduce SWAN, an open-source static analysis framework for Swift. We designed SWAN to be a robust, configurable, and extendable framework for various analyses, such as those that may detect security vulnerabilities and API misuses. SWAN features a suite of call-graph construction algorithms for creating call-graphs of varying precision, a configurable taint analysis, a configurable typestate analysis, and a cryptography API misuse analysis. SWAN's call-graph construction algorithms consist of CHA_{FP}, an adapted version of CHA [18], VTA_{FP}, an adapted version of Variable Type Analysis (VTA) [51], and UCG, a novel algorithm that we developed specifically for precisely handling Swift's function call semantics. Application developers and researchers may use the various analyses that SWAN offers out of the box without having to implement their own analysis. Through its suite of analyses, SWAN enables new directions of research for iOS and macOS that have long existed for other platforms such as Android [8], Java [27], and JavaScript [56]. Moreover, SWAN has a modular architecture that enables researchers to build their own analyses on top of it by leveraging its existing analysis infrastructure.

The primary contributions of this thesis' are as follows, along with how we evaluate the contribution and a summary of our results:

- *An open-source static analysis framework for Swift.*

  Evaluation: We assess the runtime overhead of SWAN preparing a Swift application for analysis, thereby evaluating the performance of the "core" framework. The runtime of a static analysis framework partially determines its practicality and ability to give relatively immediate feedback.

  Result: SWAN prepares (i.e., parses and translates) most benchmark applications for analysis in under 5 seconds.

- *A suite of call-graph construction algorithms.*

  Evaluation: We evaluate the performance and relative precision of SWAN' call-graph construction algorithms by running them on 22 open-

source applications and comparing their runtimes, reachable edges and methods, and the conversion of call sites from polymorphic to monomorphic.

Result: We found that CHA_{FP} is fast but is imprecise compared to VTA_{FP} and UCG. VTA_{FP} is much more precise than CHA_{FP} but is less performant. Lastly, UCG is slightly more precise than VTA_{FP} but may be less performant in some cases. For all benchmarks, CHA_{FP} computes its call graph in under 1 second. While UCG is slower than VTA_{FP} for some benchmarks, VTA_{FP} and UCG finish their call graph analysis in under 13 seconds for most benchmarks.

- *An out of the box analysis for detecting crypto API misuses.*

  Evaluation: We evaluate the effectiveness of the crypto analysis by running it on 13 open-source applications with known crypto API misuses and reporting the amount and types of misuses found.

  Result: SWAN detects 44 crypto API miuses in total across all benchmarks, with only one false-positive.

The remainder of this thesis is organized as follows. Chapter 2 discusses background material relating to static analysis concepts, types of analyses, and the Swift language. Chapter 3 describes the core components of the SWAN framework. Chapter 4 describes SWAN's suite of call-graph construction algorithms. Chapter 5 discusses SWAN's various analyses, such as cryptography API misuse detection, and demonstrates how some of SWAN's analyses can be easily configured. In Chapter 6, using open-source Swift applications, we evaluate the performance of SWAN's processing overhead, the precision and performance of SWAN's call-graph construction algorithms, and the effectiveness of SWAN's cryptography API misuse analysis. Lastly, Chapter 7 discuses work related to existing static analysis tools, call-graph construction, and cryptography analyses.

Chapter 2

Background Material

2.1 Static Analysis

A static program analysis determines various properties of interest about a program without running the program, unlike dynamic program analysis which requires running and monitoring a program's properties at runtime. The applications of static analysis vary from simple bug finders to deep analyses that find sophisticated security vulnerabilities.[1] This type of analysis requires using multiple underlying analyses together that each reason about different properties of the program, but are often dependent on each other. These analyses make up one encapsulating analysis that we refer to as a "framework," "static analyzer," or simply "analyzer."

Intermediate Representation

A static analyzer must operate on some kind of representation of the program. It can be designed to analyze the AST of the program's language, which represents the program in a hierarchical structure of nodes specific to the language. However, many analyzers instead abstract away from the program's source language by converting it into a simplified representation called an intermediate representation (IR). There are multiple advantages to this, such as being able to analyze multiple languages as long as the languages can be translated into the analyzer's IR. Another advantage is that if the language AST changes,

[1] For our discussion, we do not consider linters, such as SwiftLint [52] and Tailor [54], to be true static analyzers due to their superficial nature.

the analyzer's maintainer might only need to update the language translator and not the entire analyzer.

The IR's design greatly influences the success of the analyzer. The IR must be designed with the source language in mind to account for various semantics that the source language supports. An IR should ideally soundly capture all of the source language's semantics. However, the IR may sometimes sacrifice precision if the IR is too simple and cannot capture the unique nuances of the source language. In such cases, the translator should likely over-approximate dataflow relating to problematic or difficult semantics to avoid false-negatives downstream during analysis. If the analyzer needs to support multiple languages, then the IR's design is even more challenging because each language likely has its own nuances that the IR must capture. For instance, Java is a reference-based language while C++ supports pointers. If the analysis needs to support both of these languages using a single IR, then the IR should either be reference-based or pointer-based, and either option would require transforming a fundamental aspect of one of the input languages.

Control-Flow Graph

A static analyzer must also be able to construct a control-flow graph (CFG) for a program. A CFG represents the *intraprocedural* flow of a procedure (i.e., a function or method) by using a graph where nodes are usually either statements or basic blocks and edges between the nodes represent execution flow. To be sound (i.e., meaning to never have false negatives), the CFG must model all possible execution paths. If the program has conditional control flow, then a sound CFG will draw edges to all possible paths. Some static analyzers feature sophisticated analyses that reason about conditions and eliminate impossible paths in the CFG (also known as *deadcode elimination*).

Pointer Analysis

One of the most challenging components of a static analysis framework is its pointer analysis, which models variable dependency and dataflow. A pointer analysis can answer questions such as, "Do variables x and y point to the same

```
1 class X {}
2
3 class Foo {
4   var a: Any?
5 }
6
7 let o1 = Foo()
8 let o2 = Foo()
9 let bar = X()
10
11 o2.a = bar
12 let baz = o1.a
13 // baz points to bar
```

Figure 2.1: A Swift code example to demonstrate object-insensitivity.

location in memory?" and, "Does variable x alias variable y?" To answer such questions, the analysis must model variable dataflow. The precision of this modelling greatly affects the precision of most types of analyses that the framework supports.

Pointer analyses may vary in *sensitivity*. An *object*-insensitive analysis does not differentiate between instances of a type. Therefore, if a program creates two objects of the same type, the analysis will treat their dataflow as shared, regardless of whether their dataflow is actually related at runtime. For example, an object-insensitive analysis could not precisely track the dataflow of the program in Figure 2.1. The program creates two objects of type `Foo`, `o1` and `o2`, writes `bar` to the `b` field of `o2`, and then reads the `b` field of `o1` into `baz`. Because the analysis cannot differentiate between multiple instances of the same type (`o1` and `o2`, in this case), it treats them the same. Therefore, according to the analysis, `baz` points to `bar` at the end of the program even though the dataflow of `bar` and `baz` are unrelated at runtime.

A *field*-insensitive analysis does not differentiate between fields. For example, a field-insensitive analysis cannot precisely track the dataflow of the program in Figure 2.2.[2] The program creates an object `o` of type `Foo` with two fields, `a` and `b`, writes `bar` to the `b` field of `o`, and then reads the `a` field of `o` into `baz`. A field-insensitive analysis treats writing to `o.b` the same as writing

[2]The program uses the class `Foo` from Figure 2.1.

```
14 class X {}
15
16 class Foo {
17     var b: Any?
18     var a: Any?
19 }
20
21 let o = Foo()
22 let bar = X()
23 o.b = bar
24 let baz = o.a
25 // baz points to bar
```

Figure 2.2: A Swift code example to demonstrate field-insensitivity.

```
26 class X {}
27
28 func foo(_ x: Any) -> Any {
29   return x
30 }
31
32 var a: Any = X(), b: Any = X(), c: Any, d: Any
33 c = foo(a)
34 d = foo(b)
35 // d points to {a, b, c}
```

Figure 2.3: A Swift code example to demonstrate context-insensitivity.

to `o.a`. The same is true for field reads. Therefore, according to the analysis, `baz` points to `bar` at the end of the program even though the dataflow of `bar` and `baz` are unrelated at runtime.

A *context*-insensitive analysis does not keep a call stack for function/method calls, and therefore the analysis merges all input dataflow to a function, thereby greatly reducing analysis precision. For example, a context-insensitive analysis cannot precisely track the dataflow of the program in Figure 2.3. The program calls `foo` with `a` as an argument and writes the result to `c`. Then, the program calls `foo` with `b` as an argument and writes the result to `d`. Even though `a` and `b` are entirely disjoint, the analysis merges the two values (i.e., `x` points to both `a` and `b`). Consequently, and after the two calls to `foo`, the return value of `foo` will always include at least the values `a` and `b`. Therefore, according to the analysis, `d` points to `a` and `c` at the end of the program even though the dataflow of `d` and `a`/`c` are unrelated at runtime.

```
36 class X {}
37
38 var bar = X(), baz = X(), a: Any, b: Any
39 a = bar
40 a = baz
41 b = a
42 // b points to {a, bar, baz}
```

Figure 2.4: A Swift code example to demonstrate flow-insensitivity.

Lastly, a *flow*-insensitive analysis does not take into account the order of statements inside a function or a block. Ignoring statement order may reduce analysis precision because a variable may temporarily hold some property that is later overwritten, but the analysis would still assume that, at the variable's deallocation/destruction, it still holds the property. For example, a flow-insensitive analysis cannot precisely track the dataflow of the program in Figure 2.4. The program writes `bar` to `a`, then writes `baz` to `a`, and finally writes `a` to `b`. Because the analysis ignores statement order, the two writes to `a` take place at effectively the same time, thereby making `a` point to both `bar` and `baz`. Therefore, according to the analysis, `b` points to `bar` at the end of the program even though the value of `a` is overwritten with `baz` at runtime.

In general, the more "sensitivities" a pointer analysis has, the greater its precision. The same is generally true for any type of static analysis. Furthermore, because other analyses utilize and rely on a pointer analysis, such as those that construct call-graphs, those analyses' precision will also be greatly affected by the precision of the pointer analysis.

Call-Graph Construction

A call-graph (CG) represents procedure calling semantics from call sites to methods or functions. It serves as the backbone for any static analysis framework—the more precise the CG, the more precise the framework's results. A call-graph construction algorithm must consider all language features and semantics that may influence the soundness or precision of the CG, such as polymorphism and recursion.

Call-graph construction algorithms typically take one of two approaches.

```
43 public static void main(String[] args) {
44   Collection c = makeCollection(args[0]);
45   c.add("elem");
46 }
47
48 static Collection makeCollection(String s) {
49   if (s.equals("list")) {
50     return new ArrayList();
51   } else {
52     return new HashSet();
53   }
54 }
```

Figure 2.5: A Java code example to demonstrate CHA analysis.

The first approach, often called *type-based*, only looks at the types in the program to resolve call sites. This approach may also be referred to as *propagation-based* if the algorithm tracks which types the program uses and propagates instantiated types to call sites for more precise type information. CHA [18] is a popular type-based approach that resolves a dynamic dispatch call site to any possible matching method using the type hierarchy. CHA first finds the the type of the variable using declared variable type information. Then, CHA finds the type and its subtypes in type hierarchy and resolves the method to the type's method as well as any sub-type methods with the same name. CHA is fast but also quite imprecise, and does not do any type propagation.

For example, Figure 2.5 shows a Java[3] program for which CHA would create a highly imprecise CG. The program's `main()` function calls `makeCollection()` on line 44 and then calls the `add()` method on the resulting value on line 45. The `add()` method in Java is implemented by all types that implement the `Collection` interface. The `makeCollection()` function either returns a new object of type `ArrayList` or of type `HashSet` (lines 50 and 52).

Figure 2.6 shows the CHA CG for the program in Figure 2.5. Because CHA knows the type hierarchy of the program, it knows which types implement `Collection`'s `add()` method, such as `ArrayList`, `HashSet`, `LinkedList`,

[3]We use Java because it is commonly used to demonstrate CHA and Java analyses widely use CHA.

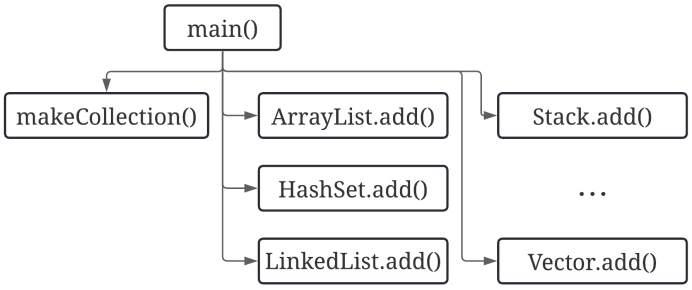

Figure 2.6: The CHA CG of the Java program in Figure 2.5.

`Stack`, and `Vector` (among many others). CHA does not know and does not consider which types the program actually instantiates and uses. Therefore, CHA resolves the call on line 45 in Figure 2.5 to *every* `Collection` type with an `add()` method, thereby creating primarily spurious edges. The CG shown in Figure 2.6 should only have edges from `main()` to `makeCollection()`, `ArrayList.add()`, and `HashSet().add()` to be maximally precise. Instead, CHA creates a highly imprecise CG.

The second approach, often called *flow-based*, uses dataflow analysis to determine the possible types that a value may hold to determine potential call targets. VTA [51] is one such flow-based approach. VTA begins with an initial CG, often generated using CHA, and prunes dynamic dispatch call sites by determining possible runtime types of class objects. That is, given any call-site, VTA knows the type of the variable on which a method is being called, and can greatly reduce CHA's resolutions by filtering based on the type if the variable. To achieve this, VTA uses a pointer analysis to track variables through the program back to their allocation sites, where their type can be determined or inferred.

Analysis Interdependence

The various analyses that make up a static analysis cannot always be executed without requiring information from another analysis. For instance, to construct a whole-program pointer analysis, the analysis needs a CG for interprocedural dataflow information. However, the call-graph construction process

may also require points-to information from the pointer-analysis to determine the type of an object upon which a method is being called. Therefore, many static analyses go back and forth between these analyses until a *fix-point* [55] is reached, meaning that the analyses have exhausted all available information and cannot find any more properties (e.g., points-to information or CG edges).

2.2 Taint Analysis

A taint analysis determines whether there is a possibility that *tainted* data from a *source* will flow to a *sink*. Sources are values (typically the return values of specific functions) that contain user data or otherwise data that may be manipulated by the user (or an attacker). Sinks are sensitive code that are prone to attacks if their input data is not properly sanitized using a *sanitizer*. A sanitizer verifies that the data is not harmful and alters the data if necessary to remove potentially harmful components. A taint analysis is comprised of pointer analysis queries, and therefore if a static analysis framework has a pointer analysis, it is generally able to implement a simple taint analysis.

Preventing a "SQL Injection" [30] attack is a classic application of taint analysis. In this type of attack, the program passes user data from a source that is considered to be tainted, such as a username or password from a login form, to a SQL query (a sink) to verify the user's credentials without first sanitizing user data. The user may craft a query extension such that when the program constructs the SQL query using the user input, the query contains an additional malicious query that may have detrimental affects, such as deleting the entire database. For example, if the program constructs the query "`SELECT * FROM Users WHERE UserId =`" to verify the username and the user inputs "`78; DROP ALL TABLES;`" as their username, then the query will be "`SELECT * FROM Users WHERE UserId = 78; DROP ALL TABLES;`" and will delete all tables in the database.

A taint analysis will alert the programmer if they have any code that could be susceptible to a SQL Injection attack by checking whether any user input can flow to SQL query constructors. The taint analysis will not alert if the

Table 2.1: SPDS' supported instruction semantics.

Instruction	**Notation**
Allocation site	$x \leftarrow \textbf{new}$
Local assignment	$x \leftarrow y$
Call site	$y \leftarrow m(p)$
Return statement	**return** x
Static field store	$A.f \leftarrow y$
Static field load	$x \leftarrow A.f$
Non-static field store	$x.f \leftarrow y$
Non-static field load	$x \leftarrow y.f$

dataflow passes through a SQL query sanitizer. Taint analyses are useful for detecting such simple vulnerabilities where it is sufficient to check for specific dataflow paths. These are one of the most common types of analyses that static analysis frameworks support because of their robustness in applicability.

2.3 Synchronized Pushdown Systems

Synchronized pushdown systems (SPDS) [48] is a context-, field-, and flow-sensitive dataflow and pointer analysis. It utilizes two pushdown systems (PDSs): one for field-sensitivity and another for context-sensitivity. Using the results of both PDSs, SPDS answers dataflow queries with high precision and efficiency. SPDS also provides weighted pushdown systems (WPDS), which is an extended version of SPDS that adds edge weights to the PDSs.[4] In practice, SPDS is similar to Interprocedural Finite Distributive Subset (IFDS) in terms of the queries it can solve, and WPDS is similar to Interprocedural Distributive Environments (IDE) [43].[5]

SPDS operates on a simple program representation consisting of eight different types of three-address code (i.e., instructions), which are listed in

[4]In this work, we generally use the term "SPDS" to encapsulate both SPDS and WPDS.

[5]We encourage our readers to read Johannes Späth's [48] work if they are interested in learning more about these concepts.

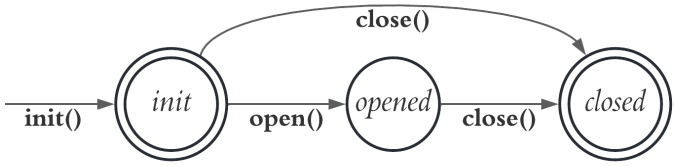

Figure 2.7: A state machine for a typestate analysis that tracks the state of an abstract `File` type.

Table 2.1. These types of instructions are similar to the rules that IFDS consumes, and therefore any program that may be converted into an IFDS-compatible representation is also compatible with SPDS. However, converting any non-trivial input language into SPDS rules requires dramatically simplifying the input.

The authors of SPDS provide two query engines: Boomerang, which uses SPDS, and IDE^{al}, which uses WPDS. Boomerang can either make *forward* or *backward* queries. Forward queries ask Boomerang to solve simple dataflow questions such as, "does variable `x` flow to line 10?" Backward queries ask Boomerang from where values flow, such as, "does variable `x` come from the return value of function `foo`?" IDE^{al} utilizes Boomerang to answer dataflow queries, but also enables tracking value states. Therefore, IDE^{al} can answer questions such as, given a variable `f` that holds a file resource, "what is the state of `f` at line 11—is the file resource open or closed?" Boomerang is good for answering taint analysis queries, whereas IDE^{al} enables a more sophisticated kind of analysis called typestate analysis.

2.4 Typestate Analysis

A *typestate analysis* tracks the state of an object of a specific type to determine various properties about the object, usually at the time of its destruction. SPDS provides support for typestate analysis using IDE^{al}/WPDS. To define such an analysis, we must first define a finite-state machine (FSM) and the identify the type that the analysis is interesting in tracking. The FSM tracks

the state of any instantiations of the selected type, and the state may change, depending on the FSM, if a specified method is called on the object.

For instance, we may want to track the state of file resources in a program. We can define a typestate analysis such that if a user opens a file resource for instance and assigns it to a variable, the analysis will begin to track the state of this variable. Figure 2.7 shows what a FSM for such an analysis could look like for a generic file resource type `File` that has `init()`, `open()` and `close()` methods. When a program creates an object of type `File`, it will call the `init()` method and the FSM will put the object into the `init` state. This state is shown as a double-circle, which means that it is an *accepting* state. If the program destroys/deallocates the variable and it is in an accepting state, then the analysis will not report an error. If the program calls the `open()` method on the variable, then its state transitions to the `opened` state and is no longer in an accepting state. If the program never calls `close()` on the variable to put it into the `closed` state (an accepting state) after putting it into the `opened` state, then the analysis will report an error.

IDE^{al}'s typestate analysis is very useful for constructing analyses that go beyond the capabilities of taint analysis. For instance, constant propagation is a classic non-distributive problem that cannot be solved using taint analysis because it requires tracking the value of variables in addition to their dataflow, but IDE^{al} is able to solve it [43]. Furthermore, we can use typestate analysis in conjunction with taint analysis to craft more sophisticated analyses. For instance, perhaps we want to add an extra transition condition to the FSM by checking if the method called on an object has a tainted argument or even has a specific state itself. Various APIs often construct and configure objects using multiple method calls. We might want to make sure that the object is built correctly by checking the method arguments. If a program calls a method on the object with a tainted argument, we can put the object into an error state. Such an analysis would not be possible with a simple taint analysis unless we use a sequence of queries to effectively transfer taintedness from the method argument to the object, but that is in fact similar to what IDE^{al} does internally.

2.5 Swift

Swift [4] first appeared in 2014 and quickly became Apple's premier language for macOS, watchOS, and iOS development. The language is multi-paradigm, meaning that it has both functional and object-oriented elements. Swift features *protocols*, which are similar to interfaces and traits, that allow for robust class extensibility. Swift also has interoperability with Objective-C to support Objective-C libraries. The Swift compiler compiles Swift to the Swift Intermediate Language (SIL), and then finally to LLVM [39].

SIL is a linear, pointer-based, and lower-level language that features over 180 instructions. In SIL, a *module* contains all information pertaining to a *compilation unit*. A compilation unit typically contains an entire program, such as a library or the user's project code. The information inside a module includes global variables, functions, and lookup tables for dynamic dispatch. Functions and methods are both located at the top-level in a module—there are no explicit classes in SIL. Rather, SIL uses lookup tables, called *value* (for classes) and *witness* tables (for protocols), to resolve dynamic dispatch calls. Every function in SIL consists of *basic blocks*. A block consists of *operator* instructions necessarily followed by a single *terminator* instruction. An operator instruction operates on data, whereas a terminator is responsible for transferring execution flow, either interprocedurally or intraprocedurally.

Because Swift ultimately compiles to LLVM, SIL is a lower-level language closer to LLVM than it is to Swift. It manually handles memory semantics, such as allocation, deallocation, and reference counting. SIL's types consist of Swift, SIL, and Objective-C types, making its type system complex and requiring a lot of type conversions. SIL also makes heavy use of pointers, and uses function pointers for every function call.

Despite its low-level complexity, SIL's dataflow semantics are fairly straightforward and much of the low-level semantics can be ignored for static analysis, making SIL an excellent analysis target. SIL is a better candidate for static analysis than Swift because, while Swift is a rapidly evolving language and its AST is always changing, SIL generally receives less drastic changes and there-

fore an analysis that targets SIL is more maintainable. SIL is also already in an expressionless (linear) form, and therefore the dataflow of instructions can be assessed individually. Lastly, the Swift compiler outputs SIL for entire modules, removing the need to resolve types and imports across multiple files.

Chapter 3

The SWAN Framework

In this section, we discuss the SWAN framework and most[1] of its primary components in detail. First, we provide an overview of the framework and introduce its components and how they are related. Then, we present a detailed discussion of each component in subsequent sections.

3.1 Overview

Figure 3.1 shows the general workflow of the SWAN framework. First ①, a Command-Line Interface (CLI) application, `swan-xcodebuild` or `swan-swiftc`, builds an Xcode [7] project (or single Swift file) and dumps the SIL representation of the built program to a directory, called `swan-dir` Ⓐ by default. Then, SWAN processes the SIL files Ⓑ inside of `swan-dir` using the following steps. SWAN parses the SIL using its SIL parser ② and saves the SIL into data structures that capture the SIL format. Next, SWAN converts the SIL into *raw* SWAN InteRmediate Language (SWIRL) (also called $\text{SWIRL}_{\text{RAW}}$) ③. This representation reduces the 163 SIL instructions that SWAN supports[2] into 26 $\text{SWIRL}_{\text{RAW}}$ instructions. SWAN then converts $\text{SWIRL}_{\text{RAW}}$ into $\text{SWIRL}_{\text{CAN}}$ (SWIRL's *canonical* form) ④, which has 17 instructions. This step runs multiple passes on the $\text{SWIRL}_{\text{RAW}}$ to achieve its canonical form. Internally, SWAN uses the SilParser for step ②, SwirlGen for step ③, and SwirlPass for step ④. SWAN finally converts the $\text{SWIRL}_{\text{CAN}}$ into SPDS rules ⑤. This is a

[1] We discuss SWAN's call-graph construction in Chapter 4 and analyses in Chapter 5.

[2] SWAN does not support all SIL instructions because many of them are not in use or never appeared in our testing.

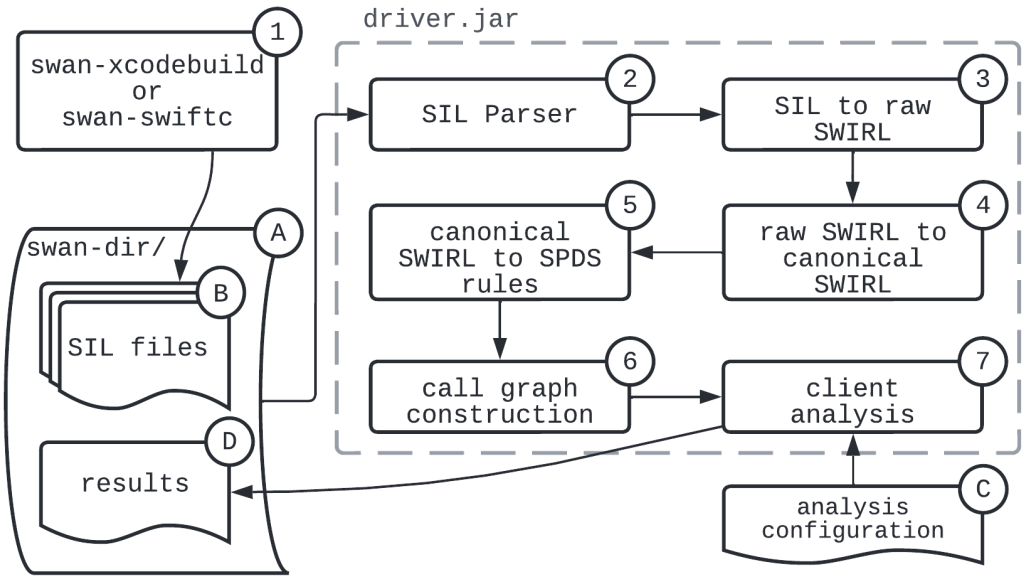

Figure 3.1: The general workflow of SWAN.

one-to-one conversion because every $\text{SWIRL}_{\text{CAN}}$ instruction corresponds to an SPDS rule.

From this point, the user can determine what SWAN will do. If the user wants to run a client analysis, SWAN will first need to generate a call graph. SWAN provides multiple call graph construction algorithms from which the user can choose. Once SWAN builds a call graph (6), the user can run a client analysis by specifying the type of analysis and providing an analysis configuration file (C). Alternatively, the user can use a built-in analysis that SWAN provides, such as the Cryptographic API misuse analysis. SWAN writes analysis results back to the `swan-dir`, where the user can view them (D).

A SWAN distribution comes with the `swan-xcodebuild` and `swan-swiftc` CLI executables. The distribution also comes with a `driver.jar` file that executes steps (2) to (7), which the user must run using a JVM. Therefore, using the command line, analyzing a Swift program with SWAN is a two-step process. We leave it up to the user to add a third step that consumes the results (D), such as a program that can display the results to the user within their integrated development environment (IDE). We have explored what a user interface (UI) for SWAN could look like in other work [60].

3.2 Building Xcode Projects and Dumping SIL

An Xcode project is the most common Swift project format. Apple provides two methods for building Xcode projects. The first method is to open the project using the Xcode IDE and build it via Xcode's UI. This approach is not favourable for external tool integration, such as SWAN, because Xcode does not feature plugin support or any way to extract build information automatically.

Apple also provides a CLI application called `xcodebuild` that can build Xcode projects without having to use the Xcode IDE. The user typically only needs to provide the application with two options to build their project: a `-project` argument pointing to the `.xcodeproj` file (or `-workspace` pointing to the `.xcworkspace` file) and a `-scheme` (or `-target`) argument specifying which build scheme or target to build. For SWAN, this approach is preferable because it does not require having to launch the IDE to configure the build settings and build the project. Therefore, we wrote a wrapper for the `xcodebuild` application called `swan-xcodebuild`, which is itself written in Swift. The user runs `swan-xcodebuild` almost[3] identically to how they would normally run `xcodebuild`. The wrapper gives `xcodebuild` additional options that allow `swan-xcodebuild` to extract SIL code from the Swift program. Specifically, giving `xcodebuild` the following options will print SIL code for every program module and with source information to the build output log:

```
OTHER_SWIFT_FLAGS='-Xfrontend -gsil
    -Xllvm -sil-print-debuginfo
-Xllvm -sil-print-before=SerializeSILPass'
```

Apple developed SIL to be an internal IR, and Apple does not make a program's SIL easily accessible to the user. However, by using these options we are able to obtain the SIL code for an Xcode project. During compilation, `xcodebuild` will write the SIL code to the build output and `swan-xcodebuild` will extract the SIL from the output. The wrapper does this by identifying

[3]The user must precede their `xcodebuild` arguments with '`--`' because any arguments before '`--`' are specific to `swan-xcodebuild`, such as `--swan-dir` which specifies the name of the output directory (`swan-dir` by default).

when a module (or "compilation unit") is being compiled by the Swift compiler, and then it extracts any SIL code that follows, thereby dividing the build output into multiple SIL programs. The wrapper will then write the SIL programs to the `swan-dir` directory as individual files.

3.2.1 Building Single Swift Files

SWAN also provides a wrapper for `swiftc` called `swan-swiftc` that can dump SIL for a single Swift file with no external dependencies. The user runs `swan-swiftc` similarly to how they would normally run `swiftc` to compile a single Swift file. The wrapper calls `swiftc` with the same additional options that `swan-xcodebuild` gives `xcodebuild`, but also gives the compiler frontend the `-emit-sil` option, and then writes the captured SIL to the `swan-dir`.

While typically not very useful to users of SWAN, `swan-swiftc` proves very useful when building a test suite with many Swift files, each testing a specific language feature. Therefore, SWAN uses this wrapper extensively for regression testing, which we discuss in Section 6.4.

3.2.2 Swift Package Manager

Many Swift apps use the Swift Package Manager (SPM) [3] instead of the traditional Xcode project or workspace format. It is possible to also dump SIL for SPM applications by editing the SPM project's `Package.swift` configuration file. Setting the `swiftSettings.unsafeFlags` field, for each build target, to the same additional options that `swan-xcodebuild` gives to `xcodebuild` will dump SIL to the build output during compilation. However, we have found that doing so is unstable because its success varies depending on the operating system, installed Swift/Xcode version, and the `swift-tools-version` specified at the top of the `Package.swift` file. Furthermore, modifying a project's configuration file is not ideal. Instead, the user can generate an Xcode project for an SPM application by running the command, `swift package generate-xcodeproj`. Then, the user can proceed with using SWAN normally. Alternatively, XcodeGen [64] can generate an Xcode project for multiple types of folder structures and project specifications.

3.3 Parsing SIL

SWAN's SIL parser (internally called SilParser) was originally a replica of Swift for TensorFlow's[4] [58] SIL parser. TensorFlow implemented their SIL parser in Swift and we replicated the parser in Scala, the primary language in which SWAN is written.[5] However, their parser only supports 49 instructions [57] (SilParser supports 163), is largely incomplete, and has numerous parsing issues, such as not being able to robustly parse SIL types. Therefore, Swift for TensorFlow's parser was only a starting point for SilParser.

SIL is a linear, assembly-like language, with no expressions, whose *general* structure is easy to understand. SIL has functions, basic blocks, instructions, and some additional information, such as witness and value tables. However, certain SIL components are challenging to parse, such as SIL types, which make up a significant part of SIL code, and the format of instructions. The biggest hurdle in writing a parser for SIL is the lack of consistent documentation. While SIL's documentation is extensive, it is not comprehensive and is often not up-to-date. Therefore, much of SilParser's implementation is the result of reverse-engineering, ad hoc fixes, trial and error testing, and investigating Apple's SIL parser, which was often not congruent with their own documentation and seemingly ad hoc itself.

SilParser roughly follows the form of a recursive descent parser. However, we have optimized SilParser to not require backtracking except in one case where SilParser checks for a type definition optionally defined in a comment before a function definition. Consequently, SilParser backtracks at most once per function definition and is largely a predictive parser. SIL has a simple structure and its instructions do not use expressions, and therefore it is usually very clear what tokens to expect next when parsing, for instance, an instruction. We designed SilParser to immediately capture information into structures that represent the SIL code. Therefore, SilParser does tok-

[4]Swift for TensorFlow is now defunct.

[5]In fact, the primary motivation for using Scala for SWAN was that Swift's enums, which Swift for TensorFlow's SIL parser uses extensively, could be elegantly converted to Scala case classes.

enization and syntactic analysis in one step.

SilParser is easy to maintain and update to support Apple's changes to SIL. Unlike Swift, which constantly receives new significant updates, SIL does not experience dramatic changes in its language. Typically, Apple rarely changes an instruction but instead removes or adds instructions. If Apple removes an instruction, we do not remove support for it to keep backwards compatibility with older Swift versions. If Apple adds an instruction, we look at its documentation and add parsing support for it.

3.4 Translating SIL to SWIRL

SPDS expects a simple, reference-based language, which we refer to as "SPDS rules." The rules include assignment, field read/write, return, and function/method call rules (or instructions). On the other hand, SIL is a pointer-based language and features closures, coroutines, basic block arguments, and complex instructions, none of which SPDS supports. SWAN supports 163 of SIL's instructions, and therefore SWAN needs to dramatically simplify its input SIL programs for them to be analyzable by SPDS. For this reason, we developed the SWAN InteRmediate Language (SWIRL) to serve as an intermediate language between SIL and SPDS rules. Once SWAN has translated SIL to SWIRL, it can convert SWIRL to SPDS rules to enable analysis on the input program.

SWAN translates SIL into SWIRL in two stages, SwirlGen and SwirlPass, which generate raw SWIRL (hereafter $\text{Swirl}_{\text{RAW}}$) and canonical SWIRL (hereafter $\text{Swirl}_{\text{CAN}}$), respectively. SwirlGen generates 26 instructions, and SwirlPass simplifies those instructions into 17 instructions, resulting in SWIRL's canonical and final form. Table 3.1 presents all SWIRL instructions. The "Type" column of indicates whether an instruction is an operator, which is a non-control-flow instruction that typically operates on some data, or a terminator, which is a control-flow instruction that is necessarily the last instruction of a basic block. The table also indicates whether the instruction exists in SWIRL's raw and canonical forms. Many raw instructions remain unchanged

Table 3.1: SWIRL's instructions.

Instruction	**Type**	**Raw**	**Canonical**
`new`	operator	✓	✓
`assign`	operator	✓	✓
`literal`	operator	✓	✓
`dynamic_ref`	operator	✓	✓
`builtin_ref`	operator	✓	✓
`function_ref`	operator	✓	✓
`apply`	operator	✓	✓
`singleton_read`	operator	✓	✓
`singleton_write`	operator	✓	✓
`field_read`	operator	✓	✓
`field_write`	operator	✓	✓
`unary_op`	operator	✓	×
`binary_op`	operator	✓	×
`cond_fail`	operator	✓	✓
`switch_enum_assign`	operator	✓	×
`switch_value_assign`	operator	✓	×
`pointer_read`	operator	✓	×
`pointer_write`	operator	✓	×
`br`	operator	✓	×
`br` (no arguments)	terminator	×	✓
`br_if`	terminator	✓	×
`br_if` (no arguments)	terminator	×	✓
`cond_br`	terminator	✓	×
`switch_enum`	terminator	✓	×
`return`	terminator	✓	✓
`unreachable`	terminator	✓	✓
`yield`	terminator	✓	✓

in SWIRL$_{\text{CAN}}$. However, not all raw instructions are present in the canonical form because SWIRLPASS simplifies them into other instructions, and SWIRLPASS converts some raw instructions into their canonical form, such as `br` and `br_if`.

The `new` instruction allocates a new value (either on the stack or heap), `assign` provides regular assignment semantics, and `literal` creates a value containing a literal, which can either be an integer, float, or string. The instructions `dynamic_ref`, `builtin_ref`, and `function_ref` are all used to reference a function. Just like in SIL, a SWIRL program must first reference a function and then apply that function reference using the `apply` instruction to call referenced function. `Dynamic_ref` references a method of an object, `builtin_ref` references a SIL builtin function but behaves exactly the same as `function_ref`, which references a function normally. The `singleton_read` and `singleton_write` instructions are for reading and writing to global data, and `field_read` and `field_write` provide access to an object's fields. `Unary_op` and `binary_op` provide operation semantics, where `unary_op` takes a single operand and an operation, and `binary_op` takes two operands and an operation. However, the operation given is irrelevant for SWAN's dataflow because all operands will flow to the result. In SIL, the `cond_fail` produces a runtime failure if the given operand (an integer) is equal to one. We retain this instruction in SWIRL for completeness, but it has no affect on dataflow because SWAN does not yet have any special handling for runtime failures. SWIRLGEN translates `switch_enum_assign` and `switch_value_assign` verbatim from SIL's `select_enum` and `select_value` instructions, respectively, because they are too complex to simplify into other instructions at the SWIRLGEN stage. The `pointer_read` and `pointer_write` instructions provide pointer semantics and are similar to SIL's `load` and `store` instructions, respectively.

The `br`, `br_if`, `cond_br`, and `switch_enum` instructions all affect intra-procedural control-flow by branching to other basic blocks. SWIRLGEN translates the `cond_br` instruction verbatim from SIL's `cond_br` instruction, but is later broken down into `br` and `br_if` instructions. The `return` instruction

provides regular return semantics. In SIL and SWIRL, a `return` instruction always requires an operand. Similarly to `cond_fail`, `unreachable` is included from SIL for completeness, but SWAN ignores this instruction and it has no effect. Lastly, SIL uses `yield` for coroutine semantics and this instruction temporarily transfers the program's execution to the calling function. SwirlGen retains these coroutine semantics.

3.4.1 SWIRLGen

$\text{Swirl}_{\text{raw}}$ can be conceptualized as being a simplified form of SIL, but without any instructions that we have determined do not mutate or move data that is relevant to dataflow analysis, such as low-level memory management instructions. For instance, SIL has various instructions that destroy or deallocate data in memory, such as `destroy_addr` and `dealloc_stack`, and SwirlGen will ignore those. While some analyses may be interested in such memory semantics, SWAN does not support these analyses. Ignoring these semantics does not compromise the soundness of dataflow propagation in SWIRL. However, SWAN's pointer analysis does not take advantage of these memory semantics, which may negatively impact its precision. For example, if a variable's data is destroyed and then written to again, the pointer analysis will not know that the old data is gone, and will therefore still say that the variable points to two values.

SwirlGen preserves all of SIL's structural and control-flow elements, such as functions and basic blocks, including their arguments. Figure 3.2 shows a SIL program and its corresponding $\text{Swirl}_{\text{raw}}$ representation. The SIL program has function called `foo` (line 56) containing multiple basic blocks (lines 57, 61 and 65), with arguments (variables `%0`, `%2`, and `%4`), and conditional control-flow (line 60). The $\text{Swirl}_{\text{raw}}$ translation of the SIL program, beginning on line 71, has an identical structure with only slight differences in the instructions. For instance, SwirlGen translates the SIL `alloc_stack` instruction to a SWIRL `new` instruction (line 58 corresponds to line 73).

In SIL, the result type of an operator instruction is not always clear from the plain-text representation of the instruction. For instance, `%1` on line 58 is

```
// ---- SIL ----
sil @foo : ($Builtin.Int1) -> () {
bb0(%0 : $Builtin.Int1):
  %1 = alloc_stack $B
  // write something to %1
  cond_br %0 : $Builtin.Int1, bb1(%1), bb2(%1)
bb1(%2 : $*B):
  // do something with %2
  %3 = tuple ()
  return %3 : $()
bb2(%4 : $*B):
  // do something with %4
  %5 = tuple ()
  return %5 : $()
}
// ---- Raw SWIRL of the above SIL ----
func @`foo` : $`($Builtin.Int1) -> ()` {
bb0(%0 : $`Builtin.Int1`):
  %1 = new $`B`, $`*B`
  // write something to %1
  cond_br %0, true bb1(%1), false bb2(%1)
bb1(%2 : $`*B`):
  // do something with %2
  %3 = new $`()`, $`()`
  return %3
bb2(%4 : $`*B`):
  // do something with %4
  %5 = new $`()`, $`()`
  return %5
}
```

Figure 3.2: A code example showing how SwirlGen preserves the structure of a SIL program in $\text{Swirl}_{\text{raw}}$.

```
SIL: %1 = init_enum_data_addr %0 : $*U, #U.DataCase!enumelt

SWIRL: %1 = new $`Any`, $`*Any`
       %new = field_read [alias %1] %0, data, $`Any`
       pointer_write %new to %1
```

Figure 3.3: How SwirlGen translates SIL's `init_enum_data_addr` instruction to $\text{Swirl}_{\text{raw}}$.

of type `$*B` (a pointer), not `$B`. In SWIRL, all result types of operators are explicit, as can be seen on line 73, where both the allocation type, `$B`, and the result value type, `$*B`, are specified. When the result type is unknown from the SIL code (not all type information is present in plain-text SIL), SwirlGen will use the type `Any` (or `*Any` for pointers). For instance, SIL's `init_enum_data_addr` instruction returns the pointer of an enum's underlying data, but SwirlGen does not know the type of the enum's data, so the result will be of type `*Any`. Figure 3.3 shows how SwirlGen translates this SIL instruction to SWIRL. This figure also shows a case where SwirlGen uses SWIRL's simpler instructions to preserve the semantics of a complex SIL instruction. SWIRL does not have an instruction that generates a pointer of an object's underlying data, mostly because pointer semantics have to be converted to SPDS' reference-based semantics later anyway. Therefore, in this case, SWIRL creates a new pointer (line 87), reads the enum's data (line 88), and finally writes the data to the pointer (line 89).

3.4.2 SWIRLPass

SwirlPass produces $\text{Swirl}_{\text{CAN}}$, which can easily be converted to SPDS rules for analysis. SwirlPass uses multiple passes to transform $\text{Swirl}_{\text{RAW}}$ so that it conforms to SPDS rule form (see Section 3.5). These passes include converting pointers to objects/references, canonicalizing $\text{Swirl}_{\text{RAW}}$ instructions, and removing basic block arguments. SwirlPass does not do all transformations (passes) at once because some passes depend on others. For instance, SwirlPass first simplifies complex instructions by breaking them down into simpler raw instructions. Later, in a separate pass, SwirlPass will convert these instructions into canonical form, along with all the other instructions.

Pointer to References Conversion

SwirlPass converts pointers to references (or objects) using multiple strategies. The simplest strategy that SwirlPass uses, if the other strategies do not apply, is to treat pointer reads and writes as field reads and writes. SWIRL's `pointer_read` and `pointer_write` instructions are converted to `field_read`

and `field_write` instructions, respectively, by treating every pointer as an object with a single field, which is arbitrarily called `value`. For example, `pointer_write %0 to %1` would be converted to `field_write [pointer] %0 to %1, value`, where `[pointer]` simply signifies that the `field_write` instruction was originally a `pointer_write` instruction.

The Swift compiler automatically generates SIL code, and SIL is a low-level language. Therefore, much of SIL's code is motivated by lower-level semantics, such as memory management. SwirlGen translates SIL to $\text{Swirl}_{\text{RAW}}$ instructions, which are simpler than SIL's, and ignores some lower-level semantics. As a result, $\text{Swirl}_{\text{RAW}}$ pointer usage may seem verbose because the motivation for the usage of pointers in many instances is gone. To reduce the number of unnecessary pointer reads and writes (which become field reads and writes in SwirlPass), and to improve both SWIRL's readability, which is helpful for debugging, and SWAN's pointer analysis performance, SwirlPass will do a pointer escape analysis and convert pointer reads and writes to assignment statements when possible. If a pointer never leaves the function and is never written to a field (i.e., has trivial dataflow), then SwirlPass will conclude that the pointer can be converted to a regular (non-pointer) value.

SwirlPass' last strategy for resolving pointers is resolving field aliases. A field alias maps a pointer to an object and a field name for a `field_read` instruction. SwirlGen marks some `field_read` instructions as aliasing a value if the `field_read` result is a pointer which the original SIL program uses to mutate the data of the object being read. On line 88 in Figure 3.3, SwirlGen marks the `field_read` instruction as aliasing `%1` because it comes from the `init_enum_data_addr` instruction, which generates a pointer typically used for mutating an enum's data.

Figure 3.4 presents a comprehensive example that demonstrates why field aliasing is important for preserving dataflow. Lines 91–98 show a program represented in $\text{Swirl}_{\text{RAW}}$. Lines 92–94 could be translated from SIL instructions that return a pointer to an object's field, such as the aforementioned `init_enum_data_addr` instruction (Figure 3.3). In this program, the value `%2` is a pointer that initially contains the data of `%0`'s `mydata` field (line 93). Then,

```
90  // ---- Raw SWIRL ----
91  bb0(%0 : $Data):
92    %1 = new $`String`, $`*String`
93    %2 = field_read [alias %1] %0, mydata, $`String`
94    pointer_write %2 to %1
95    //...
96    %3 = literal [string] "something", $`String`
97    pointer_write %3 to %1
98    %4 = field_read %0, mydata, $`String`
99
100 // ---- Canonical SWIRL Without Field Aliasing ----
101 bb0:
102   %1 = new $`String`, $`*String`
103   %2 = field_read [alias %1] %0, mydata, $`String`
104   field_write %2 to %1, value
105   //...
106   %3 = literal [string] "something", $`String`
107   field_write %3 to %1, value
108   %4 = field_read %0, mydata, $`String`
109   // %4 != %3, even though they should be equal
110
111 // ---- Canonical SWIRL With Field Aliasing ----
112 bb0:
113   %1 = new $`String`, $`*String`
114   %2 = field_read [alias %1] %0, mydata, $`String`
115   field_write %2 to %1, value
116   //...
117   %3 = literal [string] "something", $`String`
118   field_write %3 to %0, mydata
119   %4 = field_read %0, mydata, $`String`
120   // %4 == %3
```

Figure 3.4: An example showing why field alias resolution is necessary to preserve data flow in the translation from $\text{SWIRL}_{\text{RAW}}$ to $\text{SWIRL}_{\text{CAN}}$ during SwirlPass.

the program overwrites `%2` with the string "something" on line 97. Because the `field_read` on line 93 aliases `%1`, we know that any write to `%1` will in fact write to the `mydata` field of `%0`. Therefore, in the original SIL program, we know that the `pointer_write` on line 97 would overwrite the `mydata` field of `%0`.

Lines 101–108 show the corresponding program in $\text{SWIRL}_{\text{CAN}}$, but without any field alias resolution. $\text{SWIRL}_{\text{CAN}}$ does not have basic block arguments, which we explain later in this section, but `%0` is still present in the program. In this case, SwirlGen converts the `pointer_write` on line 94 to

a `field_write` on line 115 with the same operands. Semantically, this means that the `field_write` on line 107 does not overwrite `%0`'s data. Therefore, `%4` (`%0`'s `mydata`) on line 108 would not equal the written string `%3`. SwirlPass creates a dataflow gap by not correctly translating pointer writes to field writes.

Lines 112–119 show the result of SwirlPass *with* field alias resolution. In this case, SwirlPass translates the `pointer_write` on line 94 to a `field_write` to `%0`'s `mydata` field on line 118 using the field alias information encoded into the `field_read` instruction on line 114. Therefore, when SwirlPass uses field alias resolution, there is no gap in dataflow for this simple example.

SwirlPass' field alias resolution is intra-procedurally limited, meaning it does not resolve pointer writes to pointers that escape the function or have otherwise non-trivial dataflow (e.g., writing the pointer to an object and then later retrieving the pointer from the object). This can create dataflow gaps. For instance, if the program would pass `%2` to another function and that function would modify the value with a `pointer_write`, that would not affect `%0`. However, `%2` will still contain `%0`'s original value, which is written on line 115, in case the function reads the pointer's data.

Instruction Canonicalization

When SwirlPass canonicalizes a $\text{Swirl}_{\text{RAW}}$ instruction, it will do one of several types of conversions, depending on the instruction. SwirlPass may simply keep the instruction as is, which is the case for all instructions that are listed as both raw and canonical in Table 3.1. It will also translate some instructions into their canonical counterpart, such as `br` and `br_if`. Lastly, SwirlPass will decompose complex instructions into simpler instructions, while preserving their semantics, which it does for all instructions listed in Table 3.1 that have a raw representation but no canonical counterpart. For most complex instructions, this canonicalization requires modifying the structure of the program.

For example, Figure 3.5 demonstrates how SwirlPass translates the `cond_br` instruction. SwirlPass decomposes `cond_br` (line 123) by replacing it with

```
121 // ---- Raw SWIRL ----
122 bb0:
123   cond_br %0, true bb1, false bb2
124 bb1:
125   ...
126 bb2:
127   ...
128
129 // ---- Canonical SWIRL ----
130 bb0:
131   br_if %0, bb1
132 bb0i0:
133   br bb2
134 bb1:
135   ...
136 bb2:
137   ...
```

Figure 3.5: An example showing how SwirlPass translates the `cond_br` instruction from raw to canonical form.

a `br_if` instruction (line 131), which handles the 'true' case, and by adding a new block (`bb0i0` on line 132) to the program that contains a `br` instruction, which handles the 'false' case. The `br_if` instruction will make the program's control-flow fall through to the subsequent block (`bb0i0`) if its operand is false.

Omitting `cond_br` from $\text{Swirl}_{\text{CAN}}$ is a design decision motivated by keeping $\text{Swirl}_{\text{CAN}}$ simple. We would like to minimize the number of instructions in $\text{Swirl}_{\text{CAN}}$, especially if we can represent a $\text{Swirl}_{\text{RAW}}$ instruction with simpler instructions. This type of translation is more dramatic and necessary for complex instructions, especially those with implicit dataflow, such as `switch_enum_assign` and `switch_value_assign`.

Basic Block Argument Removal

SIL has basic block arguments to conform to static single assignment (SSA) form [17], which requires that the program assigns any value exactly once. However, SWAN's pointer analysis, SPDS, has no explicit support for basic blocks with arguments. To adapt $\text{Swirl}_{\text{RAW}}$ to be compatible with SPDS, SwirlPass must remove basic block arguments (but not the blocks themselves). There are at least two approaches to tackle this problem. The first

```
138 // ---- Raw SWIRL ----
139 bb0(%0 : $`Builtin.Int1`, %1 : $`Any`, %2 : $`Any`):
140   cond_br %0, true bb1(%1), false bb2(%2)
141 bb1(%a : $`Any`):
142   return %a
143 bb2(%b : $`Any`):
144   return %b
145
146 // ---- Canonical SWIRL ----
147 bb0:
148   %a = assign %1
149   br_if %0, bb1
150 bb0i0:
151   %b = assign %2
152   br bb2
153 bb1:
154   return %a
155 bb2:
156   return %b
```

Figure 3.6: An example showing how SwirlPass removes basic block arguments from a $\text{SWIRL}_{\text{RAW}}$ program.

is to convert basic blocks to functions that take arguments. A SIL program often has hundreds of basic blocks for a function, and therefore this approach would bloat the IR with unnecessary functions, which furthermore do not correspond with functions in the original SIL. The second approach is to assign the value(s) given to the basic block as arguments to the argument values, and this is what SwirlPass does. While this violates the conditions for SSA, there is no reason that $\text{SWIRL}_{\text{CAN}}$ must conform to SSA form.

Figure 3.6 shows a program similar to the program in Figure 3.5 that uses basic block arguments when branching using a `cond_br` instruction. Lines 139–144 show a partial $\text{SWIRL}_{\text{RAW}}$ program whose basic blocks have arguments. The first block (`bb0` on line 139) takes three arguments, and the first block's arguments are always identical to the function's arguments (the encapsulating function is not shown in this example). The `cond_br` instruction on line 140 branches to `bb1` or `bb2`, depending on the value of `%0`. The instruction will give `%1` to the true block `%1` as an argument and `%2` to the false block. Lines 147–156 shows the program's corresponding canonical form. As described earlier in this section, SwirlPass breaks down the `cond_br` instruction. To remove

the need for basic block arguments, SwirlPass adds `assign` instructions (lines 148 and 151) before the branch instructions, which assign the original argument values `%1` and `%2` to `%a` and `%b`, respectively. Consequently, SwirlPass removes the implicit dataflow from `cond_br` operand arguments to basic blocks receiving arguments present in the original $\text{Swirl}_{\text{RAW}}$ program.

3.5 SPDS Integration

SWAN uses SPDS for its pointer analysis, and therefore SPDS is the backbone of SWAN's analyses. Most of SWAN's architecture serves to convert a SIL program into a form that is analyzable by SPDS. Once the program is in $\text{Swirl}_{\text{CAN}}$ form, it is ready to be converted into its final representation: SPDS form.

It was necessary for us to build our own data types that wrap the SPDS API's types because SPDS was originally designed for Java and was not immediately compatible with $\text{Swirl}_{\text{CAN}}$. We designed these types to be able to accept $\text{Swirl}_{\text{CAN}}$, and therefore converting $\text{Swirl}_{\text{CAN}}$ to SPDS form is only a matter of constructing our SPDS types with the information present in the $\text{Swirl}_{\text{CAN}}$ representation of the program. Every $\text{Swirl}_{\text{CAN}}$ instruction has a direct SPDS instruction (or rule) counterpart. SWAN converts $\text{Swirl}_{\text{CAN}}$ functions to SPDS methods, which each have their own CFG. In SPDS, the program's CG is responsible for containing the entire program, and therefore SWAN will also create an SPDS CG, but the CG will not have any edges until call-graph construction, which we discuss in Chapter 4.

Control-Flow Graph SwirlPass computes a CFG for every function. A CFG in SWAN, at the $\text{Swirl}_{\text{CAN}}$ level, encodes intra-procedural[6] control-flow information into a directed graph from basic blocks to basic blocks. We will refer to this CFG as $\text{Swirl}_{\text{CFG}}$. If a block returns a value, $\text{Swirl}_{\text{CFG}}$ considers it to be an *exit block*. One motivation for making $\text{Swirl}_{\text{CAN}}$ simple, especially its control-flow instructions, is to make $\text{Swirl}_{\text{CFG}}$ easy to convert to SPDS'

[6]SWAN computes inter-procedural control-flow information using a CG, which we discuss in Chapter 4.

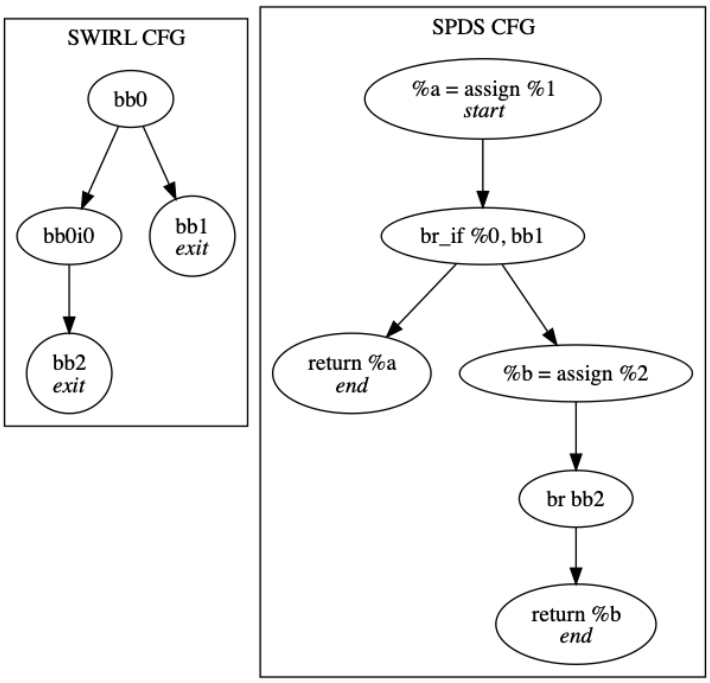

Figure 3.7: $\text{SWIRL}_{\text{CFG}}$ and SPDS_{CFG} for the $\text{SWIRL}_{\text{CAN}}$ program in Figure 3.6 (lines 147–156).

CFG, SPDS_{CFG}. Unlike $\text{SWIRL}_{\text{CFG}}$, SPDS_{CFG} maps statements to statements and has no notion of basic blocks. Using SPDS_{CFG}, SWAN knows what statement(s) execute after a given statement. A `br` instruction, for instance, will simply have an edge from it to the first instruction in the block it branches to. Figure 3.7 shows the $\text{SWIRL}_{\text{CFG}}$ and SPDS_{CFG} for the $\text{SWIRL}_{\text{CAN}}$ program in Figure 3.6 (lines 147–156). The $\text{SWIRL}_{\text{CFG}}$ only has edges between blocks, whereas the SPDS_{CFG} has edges between statements.

3.6 Models

Apple does not provide the source code for its APIs, and therefore Swift's standard library [6] consists of *black-box functions* (i.e., functions with unknown implementations). In SIL, when the program needs to use one of Swift's built-in API functions, it will declare the function signature but provide no implementation of the function. These API functions provide, for instance, container operation (for Array, Set, and Dictionary) and string manipulation

```
157 func [model] @`Swift.Array.subscript.getter : (Swift.Int)
      -> A` : $`@out τ_0_0` {
158 bb0(%0 : $`*τ_0_0`, %1 : $`Int`, %2 : $`Array<τ_0_0>`):
159   val = pointer_read %2, $`τ_0_0`
160   pointer_write val to %0
161   return val
162 }
```

Figure 3.8: An example of SWIRL model for a Swift Array getter function.

functionality. Without the implementation of these functions, SWAN does not know how the functions affect the program's dataflow. To remedy this problem, we had to build hand-crafted models that approximate the dataflow of Swift's black-box functions. Currently, these models only cover the most common functions, the three Swift containers, and many builtin type functions (e.g., `Swift.String` and `Swift.Array` related operations). Most of the Swift standard library remains not modelled because each model needs to be manually written and tested, thereby requiring significant time. SWAN's models are written in $\text{SWIRL}_{\text{RAW}}$, and SWAN parses the models using its $\text{SWIRL}_{\text{RAW}}$ parser.[7] SWAN automatically includes the models into the program as a separate module. If the user wants to use additional models, they simply need to include their own model (`.swift`) file in the `swan-dir` when analyzing their application.

Figure 3.8 shows SWAN's model for an Array container's getter function. By analyzing how programs use this function, we determined that the array's (`%2`) value at the given index (`%1`) must be written to the first argument (`%0`). Because the return type is τ_0_0 and the array's type is `Array<`τ_0_0`>`, we also determined that the function must return the array's value at the given index. Therefore, we wrote a model to provide the required dataflow between the argument and return values. SWAN treats all containers, including arrays, as an object with a single field.[8] First, the model reads from the array on

[7] SWAN does not feature a $\text{SWIRL}_{\text{CAN}}$ parser.

[8] Tracking values inside of arrays requires a sophisticated analysis which we do not support, but by using a single value for the array that is the union of all values written to the array, we can still over-approximate the values as long as we do not kill the data inside the array when writing another value to it.

line 159 to the value `val`, and then the model writes the read value to the first argument, `%0`, on line 160, and lastly the model also returns `val` on line 161.

3.7 Cross-Module Analysis

SWAN allows analyzing multiple modules (i.e., separate programs) simultaneously by stitching the modules together before analysis using the ModuleGrouper. SWAN uses multi-threading to parse and translate each module individually on a separate thread, and then SWAN groups the resulting modules on the main thread once all threads finish.

Typically, the user's program will be contained inside one module, and the other modules will be any libraries that the user's program requires. If a module requires a function from another module, it will declare that function's signature in the program but will provide no implementation. For such functions with no implementation, SwirlGen will generate a stub for that function. The stub allocates a dummy value of the function's return type and returns that value. If another module has the implementation for a stub function, ModuleGrouper will replace the stub with the implementation.

SWIRL has multiple function attributes that annotate a function with extra information, primarily for debugging purposes. Functions with the `[coroutine]` attribute are coroutines with a `yield` terminating instruction. The `[stub]` attribute is for the aforementioned automatically generated stubs. Functions annotated with the `[model]` attribute are hand-written models. The `[model_override]` attribute indicates that a `[model]` function replaced the original implementation of the function. Lastly, the `[linked]` attribute indicates that a stub function was replaced by its implementation (from another module).

Chapter 4

Call-Graph Construction

In this chapter, we describe SWAN's call-graph construction. We first introduce the challenges that SIL presents for call-graph construction, and then describe SWAN's suite of call-graph construction algorithms, namely CHA_{FP}, VTA_{FP}, and UCG. We discuss their differences in terms of precision with anecdotal examples (see Chapter 6.2 for empirical evaluation). Lastly, we discuss some nuances of our implementation.

4.1 Background

Swift functions and methods are a subset of functions in SIL. A Swift function or method corresponds to at least one SIL function (except in special cases such as inlining). Therefore, a call-graph for SIL includes a call-graph for its Swift source code. SIL has two major features that present a challenge for call-graph construction, especially when used together: function pointers and dynamic dispatch. For precise call-graph construction, function pointers require dataflow analysis, for tracking function pointers through that program, and dynamic dispatch requires determining possible resolution types at a polymorphic call-site. Furthermore, in SIL, dynamic dispatch call-sites also use function pointers. We will now discuss function pointers and dynamic dispatch, how they are related in SIL, and how they affect call-graph construction. We will also discuss why existing call-graph construction strategies, without modification, are insufficient to fully handle these features.

```
163 %0 = func_ref @foo, $T1
164 %1 = apply %0(), $T2
```

Figure 4.1: A simplified SIL code example demonstrating function pointer usage.

4.1.1 Function Pointers

Function pointers hold a reference to a function, and they may flow throughout the program to be called later. To precisely resolve a call-site, a call-graph construction algorithm must determine which functions a function pointer may reference. In SIL, this resolution requires determining the allocation sites of the function pointer for regular (non-dynamic dispatch) call-sites. SIL instructions that create regular function pointers directly reference the function by its name, and therefore the algorithm must find where the function pointers were created. In cases where the function pointer has non-trivial dataflow (e.g., passed as an argument to a call), the algorithm requires a pointer analysis. The precision of the pointer analysis thereby also determines the precision of the call-graph. For instance, SIL may write function pointers to fields. If the algorithm's pointer analysis is field-insensitive, the call-graph will be less precise.

SIL uses function pointers by first referencing a function, writing the reference to a value, and then later applying that value at a call-site. Figure 4.1 demonstrates that case, where the program references a function called `foo` (line 163) and then applies the function pointer (line 164). `$T1` represents the type of variable `%0`, and `$T2` represents the type of the returned value `%1`. Similar to this example, most SIL function reference applications typically occur immediately after the function reference instruction. SIL also allows references to be passed as an argument to a function or a basic block just like any other value.

Table 6.2 lists our benchmark apps and their characteristics. In our benchmark apps, on average, 0.40% of non-dynamic function pointers have non-trivial dataflow. This value may seem insignificant, but SIL uses function

```
protocol Parent { func foo() }
class ChildA : Parent { func foo() { ... }
class ChildB : Parent { func foo() { ... }

func bar(p: Parent) {
  if (...) { p = ChildA() }
  else { p = ChildB() }
  p.foo() // which foo is called?
}
```

Figure 4.2: A generic Swift code example demonstrating polymorphic method calling.

```
protocol Parent { func foo() }
class ChildA : Parent { func foo() {} }
ChildA().foo()
```

Figure 4.3: A Swift code example with a class, protocal, and method call usage.

pointers for nearly every function call, averaging 8,265 function pointers per benchmark (for both trivial and non-trivial dataflow). SIL also uses function pointers for low-level procedures that it frequently uses and which would not be present in the source Swift code (e.g., auto-generated deallocators, setters, and getters), thereby increasing the amount of function pointers. Furthermore, 30.78% of all dynamic function pointers in our benchmark apps have non-trivial dataflow. Therefore, resolving function pointers for SIL programs is necessary to construct a sound call-graph.

4.1.2 Dynamic Dispatch

Swift is an object-oriented language (OOL) and features polymorphic call-sites. This type of semantic is often called dynamic dispatch. Figure 4.2 demonstrates how a variable may be of two different sub-types of the same parent type at runtime. In such cases, a sound call-graph must model the call-site `p.foo()` (line 178) to target potentially both `ChildA.foo()` and `ChildB.foo()`.

Lookup Tables To resolve dynamic dispatch, SIL uses witness tables for protocols and value tables for classes. These tables contain useful class in-

```
177 %5 = class_method %4 : $ChildA, #ChildA.foo : <type
      x>, $<type y>
178 %6 = apply %5(%4) : $<type y>
179 [...]
180 sil_vtable ChildA {
181   #ChildA.foo: <type x> : @$<mangled> // ChildA.foo()
182   [...]
183 }
```

Figure 4.4: A simplified SIL code example corresponding to the Swift program in Figure 4.3.

formation for resolving dynamic calls, such as which methods belong to a class and any inherited methods from its super-classes. SIL dynamic dispatch instructions, such as `class_method` and `witness_method`, use virtual lookup tables to resolve method calls at runtime. Figure 4.3 shows a program where class `ChildA` extends protocol `Parent` and implements method `foo`. The program creates a new `ChildA` object and calls `foo` on it (line 176). Figure 4.4 shows the program's corresponding partial SIL code. Similar to a regular function call, SIL first references a function or a method and then applies it. The `class_method` instruction (line 177) finds the corresponding value table (line 180) based on the dynamic type of its operand `%4`, which is `ChildA`, and then looks up the given index, `#ChildA.foo`, in the table. Line 181 shows the corresponding value table entry using the mangled name of the function. Thus, the call-site on line 178 resolves to `ChildA.foo()` at runtime.

SIL uses witness tables for generic type method dispatch (e.g., protocols). Method calls on protocol types require witness tables to determine which method to resolve to and use the `witness_method` instruction. The SIL semantics for such cases are similar to how a function pointer is referenced and called in Figure 4.4 on line 177 and line 178.

Function Pointers Dynamic dispatch in SIL utilizes function pointers. So far, we have only given intra-procedural (trivial) examples of function pointer usage. However, SIL often uses function pointers with non-trivial dataflow that requires a dataflow analysis to resolve. Therefore, dynamic dispatch resolution

```
184 sil_vtable ChildA {{
185   #ChildA.foo: (A) -> () -> () : @$<mangled> //
      ChildA.foo()
186 }
187 sil_vtable ChildB {{
188   #ChildB.foo: (B) -> () -> () : @$<mangled> //
      ChildB.foo()
189 }
190 sil_witness_table hidden ChildA: Parent module test {
191   // protocol witness for Parent.foo() in conformance
      ChildA
192   method #Parent.foo: <type> : @$<mangled>
193 }
194 sil_witness_table hidden ChildB: Parent module test {
195   // protocol witness for Parent.foo() in conformance
      ChildB
196   method #Parent.foo: <type> : @$<mangled>
197 }
```

Figure 4.5: A simplified SIL value and witness tables that SWAN generates for the program of Figure 4.2.

for SIL call-graph construction is a two step process: (1) find the allocation sites of the function pointer at a call-site, and (2), for each allocation site (i.e., dynamic dispatch instruction), determine the types of the operand (or, for a less precise solution, the instantiated types at that program point).

Dynamic Dispatch Graph SWAN generates a Dynamic Dispatch Graph (DDG) for resolving a dynamic call-site given the call-site index. Whenever a program makes a dynamic call, it looks up the statement's call-site index along with the statement's operand type to resolve the call. A DDG is essentially a traditional type hierarchy tree that includes methods and call-site indices, and this statically emulates SIL's runtime lookups. To build a DDG, SWAN reverse-engineers SIL value and witness tables. As an example, Figure 4.5 shows the corresponding value and witness tables (without `init` and `deinit` entries) for the classes `ChildA` and `ChildB` from Figure 4.2, and Figure 4.6 shows the DDG for that program. SWAN's call-graph construction algorithms use the DDG to find which methods a dynamic reference may resolve to, and VTA_{FP} and UCG can further filter these methods with type information.

To handle value tables, SWAN creates nodes for all indices (`#ChildA.foo`

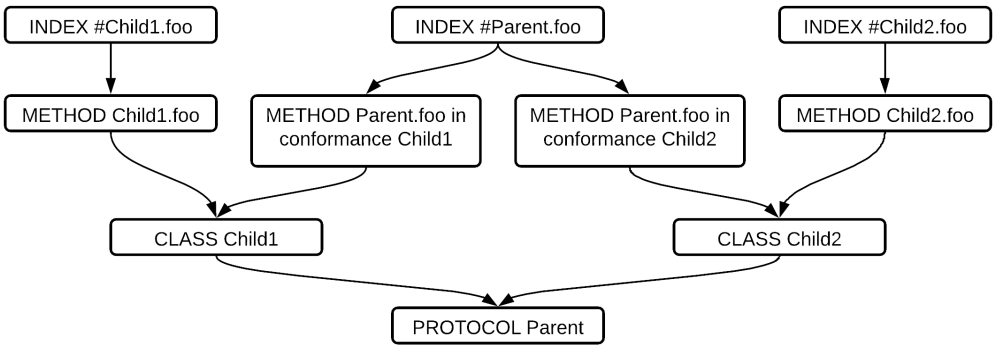

Figure 4.6: A DDG for the Swift program of Figure 4.2.

and `#ChildB.foo`), methods (`ChildA.foo()` and `ChildB.foo()`), and classes (`ChildA` on line 184 and `ChildB` on line 187). SWAN then adds edges from indices to methods and from methods to classes. While not shown in the example, in the case of inheritance, a value table may have methods from other types. In such cases, SWAN adds nodes for the other types. SWAN adds an edge from these types to the type indicated in the value table, thereby inferring type inheritance.

Handling witness tables is more complicated because they may have multiple types of entries. For this example, we infer type inheritance from the witness table definition (e.g., `sil_witness_table hidden ChildA: Parent module test`). Therefore, we create nodes for the types involved and add edges between them (`Child1`→`Parent` from line 190 and `Child2`→`Parent` from line 194). Lastly, SWAN creates nodes for the `method` table entry functions and indices and draw edges between them and the class from the table definition.

4.2 Algorithms

In this section, we describe SWAN's suite of call-graph construction algorithms. The algorithms are: CHA_{FP} and VTA_{FP}, variants of CHA and VTA, respectively, that handle function pointers, as well as UCG, which simultaneously handles function pointers and dynamic dispatch more precisely. We first

describe these algorithms and then demonstrate the differences between them using several code examples. We later evaluate these algorithms in Chapter 6.2.

4.2.1 $\mathbf{CHA_{FP}}$

Because traditional CHA does not resolve function pointers, we present CHA_{FP}, our adaptation of CHA, which uses signature matching to resolve call-sites with non-trivial function pointers. To resolve a call-site, CHA_{FP} looks at the type signature of a call-site and then resolves that call-site to all functions with the same type signature. This resolution results in a large number of call-graph edges, especially for common type signatures such as `(Bool)→()`. Nevertheless, this strategy soundly over-approximates function pointers and aligns with the sound but imprecise results of traditional CHA.

4.2.2 $\mathbf{VTA_{FP}}$

Traditional VTA can be described as a pointer analysis because VTA resolves the types of values by tracking their dataflow back to their allocation site, where VTA can determine their type. Because VTA does not resolve function pointers, we present VTA_{FP}, an adaptation to VTA [51] that, in addition to the original algorithm, resolves function pointers by utilizing VTA's pointer analysis. Similar to the original VTA algorithm, VTA_{FP} starts with a conservative call-graph produced by CHA_{FP} and then prunes that graph. In addition to resolving calls based on variable type, VTA_{FP} also tracks function pointer usages back to their allocation sites to resolve them. Because the initial graph is highly imprecise and VTA_{FP} is a pruning algorithm (that is, a non-optimistic algorithm that removes edges rather than adds edges), VTA_{FP} may significantly over-approximate the call-graph.

VTA_{FP} is field-based (i.e., object-insensitive) and, therefore, does not distinguish between different objects and treats all fields as belonging to one instantiation of a type. VTA_{FP}'s object-insensitivity may lead to imprecision if the program uses multiple objects of the same type in the same place because VTA_{FP} would treat these multiple objects as the same object. Storing

function pointers to fields, particularly to pointers (i.e., an object with a single field), is common in SIL due to its low-level nature and heavy usage of pointers. VTA_{FP} is also flow-insensitive, meaning that statements in a block after a call may affect the types VTA_{FP} resolves for that call. While flow-insensitivity may lead to imprecision in the context of a taint analysis, it is less likely to lead to imprecision for VTA_{FP}. Flow-insensitivity may cause imprecision if the program overwrites values, but SIL is SSA-compliant (i.e., SIL programs do not overwrite values). Any value overwrites are only introduced by SwirlPass to resolve basic block arguments. SWAN's taint tracking uses a different, more precise pointer analysis, and therefore any extra edges that VTA_{FP} produces are unlikely to lead to imprecision in SWAN's taint analysis.

4.2.3 UCG

UCG is a Kleene-style worklist algorithm that collects instantiated types and iteratively and optimistically adds edges to the call-graph. UCG uses two types of pointer analyses: an on-demand pointer analysis for resolving function pointers, and an optional flow-based pointer analysis for pruning instantiated types. By default, UCG uses SPDS [48] queries for the on-demand pointer analysis and VTA_{FP} as the flow-based pointer analysis for pruning instantiated types. However, these pointer analyses can be swapped for other on-demand or flow-based pointer analyses if desired.

UCG consists of the following components:

- The instantiated types component goes through all reachable parts of the program and collects types created by the program.
- The function pointer resolution component queries the on-demand pointer analysis to resolve function pointer edges.
- The dynamic dispatch component optionally queries the flow-based pointer analysis to resolve dynamic dispatch edges.

Algorithm 1 Compute UCG Call Graph

1: **Input**
2: – g is the initial Call Graph without any edges
3: – d is the Dynamic Dispatch Graph
4: **procedure** MAIN(g, d)
5: Let q be a map of queried call-sites to their function ref alloc sites
6: **for** $e \in g$.entryPoints() **do**
7: Let w be a worklist of all the blocks of e
8: Let *out* be a map from blocks to sets of instantiated types (bit vectors)
9: Let $proc_{in}$ be a map from functions to sets of instantiated types (bit vectors)
10: Let *ret* be a map from functions to sets of blocks
11: Let *seen* be a set of blocks
12: Add e to w
13: PROCESSWORKLIST(w, *out*, $proc_{in}$, *ret*, *seen*, q, g, d)

Type Propagation

This component of our algorithm is a fix-point style worklist algorithm that collects and stores an out-set of instantiated type for each block and an in-set of instantiated types for each method. UCG starts by adding the blocks of an entry point into its worklist. UCG then processes all blocks in the worklist, adding blocks to it by following intraprocedural and interprocedural edges. UCG repeats the process until it processes all entry points.

Algorithm 1 shows the main procedure of UCG, which takes in a call-graph g, to which it adds edges, and a DDG d. A DDG enables UCG to find potential resolutions of a dynamic dispatch function reference. For each entry point, UCG calls `processWorklist()` with a set of arguments: the worklist containing the basic blocks of the current entry point (w), a map of block out-sets of instantiated types (*out*), a map of method in-sets of instantiated types ($proc_{in}$), a map of methods to their return call-sites (*ret*), a map of blocks to successor blocks (*ibs*), a set of blocks (*seen*), and a map of call-sites to pointers that flow to their function references (q).

The map *out* stores a conservative set of instantiated types at the end of the block in any instance of the program. The map $proc_{in}$ stores a conservative set of instantiated types at the beginning of a method in any instance of the pro-

gram. If the algorithm adds a new value to the block's outset in *out*, then the successors of the block, both intraprocedural and interprocedural, were previously processed with an under-approximated input set of instantiated types. That is, there is now new type information available to the successors. Therefore, UCG needs to recompute their call-graph edges and out-sets. Similarly, if UCG adds types to a method's in-set in $proc_{in}$, UCG needs to revisit the blocks of the method. The map *ret* tracks interprocedural control-flow edges. The parameters *seen* and q are relevant in other parts, which we discuss later in this section.

Algorithm 2 illustrates how UCG processes an item from its worklist. UCG first pops a block c from the worklist and collects an initial set b of instantiated types from c's intraprocedural predecessors (Lines 14–18). This set b represents all instantiated types that UCG knows of thus far and that the current block c begins with. The set b does not necessarily contain all instantiated types (unless a fix-point has been reached) because UCG may not have traversed all of the block's predecessors yet. UCG gathers these types by looking at the out-sets of all of c's predecessors and aggregates them into b. Furthermore, if c is the entry point of a method (i.e., the first basic block), UCG adds the in-set of the method to b (Line 20). UCG does not track interprocedural predecessors, but instead uses an in-set in $proc_{in}$ to track the instantiated types that would appear from interprocedural control-flow.

UCG then goes through all operator statements in the block c, in the order of their execution, adding any new instantiated types to b (Line 24). If UCG finds a call-site, it gathers the targets for that call-site using `getTargets` (Line 26), which uses either the DDG or the on-demand pointer analysis to find possible targets. For each target, UCG updates the return sites (i.e., the $proc_{in}$ in-sets) and the set of instantiated types after the return from possible calls (line 29). The call to `invalidateCacheandRevisitSuccessors` (Line 31) is part of the function pointer resolution component of UCG, which we describe later in this section.

Algorithm 2 Process Worklist of Blocks

```
1: Input
2: – w is a worklist of blocks with unique elements
3: – seen is a set of seen blocks
4: – out is a mapping from blocks to bit vectors
5: – proc_in is a mapping from functions to bit vectors
6: – ret is a mapping from functions to blocks (return sites as successors)
7: – q is a map of seen call sites, that require queries, to their alloc sites
   (query cache)
8: – g is the call-graph
9: – d is the DDG
10: procedure processWorklist(w, in, out, proc_in, ret, q, g, d)
11:   while |w| > 0 do
12:     c ← w.pop()
13:     Let b be a set of instantiated types
14:     for pred ∈ c.preds() do
15:       if pred ∈ out then
16:         b ← b ∪ out[pred]
17:       else
18:         out[pred] ← ∅
19:     if c == m.firstBlock() and m ∈ proc_in then
20:       b ← b ∪ proc_in[m]
21:     seen ← seen ∪ b
22:     for o ∈ c.operators() do
23:       if o is an allocation statement then
24:         b ← b ∪ o.type()
25:       else if o is a call site then
26:         targets ← getTargets(o, b, g, d, q)
27:         for t ∈ targets do
28:           Let added = addEdge(o, t, g)
29:           processTarget(t, c, b, w, proc_in, out)
30:           if added and t.entryBlock() ∈ seen then
31:             invalidateCacheAndRevisitSuccessors(t, w, q)
32:     changed ← c ∉ out or out[c] ≠ b
33:     out[c] ← b
34:     if changed then
35:       Add all successors of c to w
36:       if c is an exit block of function m then
37:         Add ret[m] to w
```

When UCG finishes going through all statements, it checks if the out-set of the block c has changed compared to previous iterations. If the out-set has changed, the successors now have new information that may affect their call-graph edges. Therefore, UCG adds all successors of the current block to the worklist, including interprocedural ones from ret. Using this approach, UCG loops through the worklist until no new blocks are added to it, indicating that the call-graph has reached a fix point and all information has been processed.

Function Pointer Resolution

UCG requires an on-demand pointer analysis that is monotonic with respect to call-graph edges. Thus, for any variable, the more call-graph edges we have, the larger the points-to sets will be for this pointer analysis. The larger the points-to sets, the more values that pointers may have through interprocedural data and control-flow, leading to more call-graph edges through function calls. This process continues until UCG reaches a fix-point.

Algorithm 3 shows how UCG finds targets of call-sites. UCG easily resolves intraprocedural function references (Line 10) if their target is readily available, and handles dynamic dispatch indices by making queries to the DDG (Lines 12 and 21). If the function reference is an interprocedural function pointer value, UCG queries the on-demand pointer analysis (Line 15). The query returns a collection of allocation sites, which are either dynamic dispatch indices or function references. UCG then uses this result to resolve b to a set of targets.

Dynamic Dispatch Resolution

UCG collects instantiated types in a control-flow sensitive manner and uses them to make DDG queries. The instantiated types are the set of live types at that program point and thus a dynamic dispatch must resolve to a method of one of these types at runtime. Without the flow-based pointer analysis, the DDG query returns methods from the entire set of instantiated types. However, if a type is instantiated, it does not mean that it is a possible receiver of a call. The type may never flow to the receiver argument of the call and may only be useful elsewhere.

Algorithm 3 Get Targets of a Call-Site

```
1: Input
2: – o is a map of seen call sites, that require queries, to their alloc sites
3: – b is a set of instantiated types
4: – g is the call-graph
5: – d is the DDG
6: – q is a map of seen call sites, that require queries, to their alloc sites
   (query cache)
7: function GETTARGETS(o, b, g, d, q)
8:    r ← lookup o.functionRef() in o.function.symbolTable
9:    if r is a static function reference then
10:      return {r.target()}
11:   else if r is a dynamic dispatch index then
12:      targets ← query r.index() in d with instantiated types b
13:      return targets
14:   else if r is inter-procedural then
15:      allocSites ← QUERY(g, o.functionRef(), q)
16:      targets ← {}
17:      for a ∈ allocSites do
18:         if a is a static function reference then
19:            targets ← targets ∪ {a.target()}
20:         else if a is a dynamic dispatch index then
21:            targets ← targets ∪ query a.index() in d
22:      return targets
```

UCG mitigates the imprecision arising from propagating instantiated types that are not possible dynamic dispatch operands. Instead of querying the DDG using the set of all instantiated types, UCG queries the optional flow-based pointer analysis for the types that may flow to the receiver of the call. UCG then intersects the instantiated types with the types that flow to the receiver, and makes the DDG query with the resulting set of types. This type pruning is included in the queries to d on lines 12 and 21 in Algorithm 3.

Querying

Algorithm 4 shows the query to the on-demand pointer analysis. UCG uses the query cache q to see if it has queried the same call-site before (Line 6). If the function reference is in q, then the previous results are still valid. That is, the results have not been invalidated by the algorithm's revisiting logic. If

Algorithm 4 Get pointers for a Function Reference.

```
1: Input
2: – q is a map of seen call sites, that require queries, to their alloc sites
3: – r is an interprocedural function ref
4: – g is the Call Graph
5: function QUERY(q, r, g)
6:    if r ∈ q then
7:       return q[r]
8:    else
9:       allocSites ← POINTERANALYSISQUERY(r, g)
10:      q[r] ← allocSites
11:      return allocSites
```

Algorithm 5 Invalidate Caches for On-demand Pointer Analysis

```
1: Description
2: This function adds all the transitive successor blocks of a
3: target t to the worklist w and removes all call-sites in those
4: blocks from q.
5: Input
6: – t is the function to invalidate along with its successors
7: – w is a queue of blocks
8: – q is the query cache, a map of seen queried call sites, to their alloc sites
9: procedure INVALIDATECACHEANDREVISITSUCCESSORS(t, w, q)
10:   Let s be all the transitive successor blocks of t, including t's blocks in
      depth first order
11:   for b ∈ s do
12:      for c in call-sites of b do
13:         Remove c from q
14:      Add b to w
```

there is a cached set of query results, UCG uses the cached results. Otherwise, UCG makes a query to the pointer analysis, cache the results, and then use them (Lines 9–11).

Invalidation

Algorithm 5 shows UCG's invalidation logic, which simply involves removing all cached successors of method t and adding the successors back into the worklist w. To motivate invalidation, consider a function reference r at a call-site c. We assume that the pointer analysis may produce a larger points-to set with new information if there are new call-graph edges that precede c.

However, adding call-graph edges to call-sites that are strict successors of c should not change the set of pointers that r may have because no new pointers will flow to r. Alternatively, if UCG visits c but later adds call-graph edges that precede c in control-flow, then the points-to sets for r may change, and query cache q will contain an invalid set of types for r. Therefore, UCG will invalidate the entry for c in q and revisit c so that it computes call-graph edges with a larger points-to set that contains updated information. This new information could result in new edges when UCG revisits c. UCG detects if it is adding an edge to a previously visited block using the *seen* variable. If such an edge is added (Line 30 in Algorithm 2), UCG invalidates entries for all intraprocedural and interprocedural successors of that block (Line 31 in Algorithm 2).

4.3 Algorithm Comparison

To anecdotally compare our call-graph construction algorithms and demonstrate differences in precision between CHA_{FP}, VTA_{FP}, and UCG, we present the following simple examples.

CHA_{FP} Imprecision

Figure 4.7 shows three classes `A`, `B`, and `C` that implement the protocol `Parent`. In the function `cha_imprecision()` on line 204, using CHA_{FP}, the call `x.foo()` on line 208 resolves to `A.foo()`, `B.foo()`, and `C.foo()` because the class hierarchy for `Parent` includes `A`, `B`, and `C`. However, only `A` and `B` could be instantiated at runtime.

Because `x` is assigned a read from the field `f`, and `f` is only assigned to an `A` in the constructor for `F`, VTA_{FP} reasons that `x` must point to a value of type `A`. Therefore, VTA_{FP} determines that `x.foo()` must dispatch to only `A.foo()`. UCG propagates both `A` and `B` to the program point at the call to `x.foo()`. However, VTA_{FP} says `x` may only be of type `A`. Since UCG uses VTA_{FP} for pruning, it also resolves `x.foo()` to only `A.foo()`.

```
198 protocol Parent { func foo() }
199 class A : Parent { func foo() { ... } }
200 class B : Parent { func foo() { ... } }
201 class C : Parent { func foo() { ... } }
202 class F : { var f: Parent = A() }
203
204 func cha_imprecision() {
205   var aF: F = F()
206   var b: Parent = B()
207   var x: Parent = aF.f
208   x.foo()
209 }
210
211 func vtafp_imprecision() {
212   var aF: F = F()
213   var x = aF.f
214   x.foo()
215   var b: Parent = B()
216   var bF: F = F()
217   bF.f = b
218 }
```

Figure 4.7: A Swift code example of non-trivial data flow using classes.

VTA_{FP} Imprecision

In Figure 4.7 on line 211, the function `vtafp_imprecision()` demonstrates how VTA_{FP}'s field-based and flow-insensitive nature makes it less precise than UCG. If we only consider lines 212–214, then `x.foo()` on line 214 resolves to `A.foo()` using VTA_{FP}. However, due to VTA_{FP}'s flow-insensitivity, the statements on lines 215–217 also affect what `x.foo()` resolves to because VTA_{FP} is also field-based (object-insensitive), and the field write on line 217 effectively writes `b` to `aF.f`. Therefore, the field read on line 213 writes both `aF.f` (a value of type `A`) and `bF.f` (a value of type `B`) to `x`. As a result, VTA_{FP} not only resolves `x.foo()` to `A.foo()` but also (incorrectly) to `B.foo()`.

On the other hand, due to its flow-sensitivity and type propagation, UCG resolves `x.foo()` to only `A.foo()`, even without VTA_{FP} type pruning, because the statements after the call have no affect. Therefore, for this example, UCG is more precise than VTA_{FP}. Lastly, CHA_{FP} resolves `x.foo()` to `A.foo()`, `B.foo()`, and `C.foo()`, giving the least precision.

4.4 Implementation Details

In this section, we discuss some technical details of our call-graph construction algorithms and give further insight into how they work.

4.4.1 Entry Points

Our implementation shares some logic across all call-graph construction algorithms, including the detection of call-graph entry points. Only using `main()` as the single entry point is not sufficient due to lifecycle execution flow (e.g., callbacks from the program's UI), which creates discontinuity in execution flow, and because we want to analyze parts of the program that are not necessarily called. We initially consider all functions that meaningfully contribute to the program's dataflow and, for UCG, non-library functions to be entry points. That is, we ignore uninteresting functions such as deinitialization and deallocation functions because they have no meaningful effect on the call-graph, especially in relation to the original Swift source code. UCG ignores most library functions because we are interested in user code and only parts of the libraries that are reachable from the user code. However, the analysis user may enable library analysis, which initially sets all meaningful library functions as entry points.

4.4.2 Closures

Because SWAN does not have complete closure support and has no way of modelling partially applied function state, our pointer analyses either make no extra effort to resolve them or entirely ignore them. CHA_{FP} resolves calls to and from closures using type signature matching because type matching is fast and resolving closures this way has minimal additional overhead. VTA_{FP} has no special handling for closures. For UCG, we suspect that handling closures leads to performance degradation because closures often produce complex dataflow, which may cause SPDS queries to time out. Therefore, UCG does not attempt to find the allocation sites of function pointers, using its on-demand pointer analysis, whose dataflow goes through or from closures.

4.4.3 Libraries

One advantage of analyzing Swift, as opposed to other compiled languages such as C/C++, is that we have access to the source code of all non-proprietary libraries due to Swift's build system. Consequently, when SWAN analyzes an application, it has complete access to the libraries and can track dataflow through them. This has the advantage of having all relevant dataflow available, but may degrade performance if an optimistic algorithm starts traversing every library, especially parts of libraries that the user code never even uses.

In our implementation, CHA_{FP} creates a call-graph for the entire application, including its libraries, because the analysis is fast and can analyze entire libraries with very little additional overhead. VTA_{FP} prunes the graph that CHA_{FP} creates, and therefore also produces a graph for the application and libraries. On the other hand, UCG is optimistic and starts analyzing the application from entry points within the user code. Therefore, UCG only explores the libraries in so far as the user code uses them (unless the user enables full library analysis).

4.4.4 Improving the Precision of Instantiated Types

If a program passes a dynamic function reference interprocedurally, then the methods that the reference may refer to must be among the methods of objects that were instantiated when the reference was taken/allocated. Therefore, as a precision improvement and optimization when making a DDG query to draw an edge for an application of a dynamic reference, UCG does not use the instantiated type at the call-site. It instead uses the instantiated types of the block where the dynamic reference was created. We have implemented the optimization in UCG, and it required the following changes to our algorithm.

1. The procedure `processWorklist()` of Algorithm 1 receives an additional argument, *ibs*, which is a mapping from blocks where dynamic references are created to sets of blocks where the dynamic reference is used.
2. In line 35 of Algorithm 2, we additionally add the blocks in *ibs* to the worklist.

3. In line 21 of Algorithm 3, we add the current block to the *ibs* set of the block to which the dynamic reference belongs.

4.5 Summary

SWAN provides three call-graph construction algorithms: CHA_{FP}, VTA_{FP}, and UCG. CHA_{FP} is an extension of CHA that resolves function pointers using simple type signature matching, which can lead to a high amount of spurious edges if the program has many functions that share the same type signature. VTA_{FP} is an adaption of VTA that uses utilizes VTA's pointer analysis to resolve function pointers to their allocation sites, determining the function that the pointer references. UCG is a novel algorithm that we developed specifically to resolve function pointers and dynamic dispatch simultaneously. UCG propagates instantiated types through the program to resolve dynamic dispatch, utilizes SPDS for its highly precise and on-demand pointer analysis to resolve function pointers, and only revisits parts of the program when necessary. Because UCG is an optimistic algorithm, it also (optionally) utilizes VTA_{FP}, a pessimistic algorithm, to prune instantiated types. In terms of theoretical precision, we expect CHA_{FP} to be the least precise due to its fast but highly imprecise type signature matching. VTA_{FP} should be significantly more precise than CHA_{FP} due to its pointer-analysis based type tracking. Lastly, we expect UCG to be the most precise, and at least as precise as VTA_{FP}, because of its precise propagation of instantiated types, use of SPDS' pointer analysis, and type pruning using VTA_{FP}.

Chapter 5

SWAN Analyses

SWAN serves as a general analysis framework for Swift and provides various types of analyses, some of which are configurable and enable the user to write their own analyses. SWAN offers a configurable taint analysis and typestate analysis. SWAN also has two domain-specific and non-configurable analyses: an analysis that detects energy inefficient configurations of Swift's *Core Location* API, and an analysis that detects cryptography misuses in the *CryptoSwift* API [34]. The latter two analyses serve as examples of SWAN's practicality and applicability to specific problems in Swift programs. In this section, we explain each of these analyses.

5.1 Taint Analysis

SWAN's taint analysis utilizes Boomerang, which is SPDS' query engine for making simple forward and backward pointer analysis queries. To determine whether a source flows to a sink without passing through a sanitizer, SWAN first finds *seeds* representing all values that come from specified sources. A seed requires the value of interest and a control flow edge to indicate the point in the program from where the query begins. SWAN uses Boomerang's forward queries, and therefore it generates seeds for any values returned from any source functions. Then, SWAN builds forward queries with these seeds and asks Boomerang to solve the queries. The Boomerang solver will populate its PDSs with reachability information representing the seed's dataflow. Boomerang returns the results in a table format that represents the final

PDS states. SWAN then checks the results to see if any seed reached a sink function without first passing through a sanitizer function. If SWAN detects such dataflow, it will report a defect to the user.

Using the Analysis

SWAN's taint analysis may be configured entirely in a JSON file (hereafter referred to as a "spec"). A taint analysis spec allows the user to configure sources, sinks, and sanitizers.[1] Every source, sink, and sanitizer must either be the full name of a function (based on its demangled SIL function name) or a regular expression matching at least one function (the user must set `"regex": true` in this case).[2] The user may also optionally specify which argument of the sink is sensitive (e.g., one argument of the sink is a SQL query and the other arguments are not sensitive because they are not manipulatable). The user must provide taxonomy information, such as a description of the taint analysis indicating what the analysis detects, a description for each source, sink, and sanitizer, and advice for fixing the issue, if detected. Inside the spec, the user may define multiple taint analyses, and therefore SWAN takes exactly one spec as an argument for its taint analysis mode. To use the taint analysis mode, the user must run SWAN with the flag `-t` followed by the path to the spec.

Figure 5.1 shows an example taint analysis spec that utilizes all features that SWAN offers for taint analysis. Lines 220–222 define the general taxonomy of the spec. This information is displayed to the user to assist them with resolving defects. This spec defines one source on lines 224–227 using the source's full name (as opposed to using a regex). Next, the spec defines a sink on lines 230–235 using a regex to match the full function name of the sink. On line 232, the spec sets only the first argument of the sink function to be considered sensitive by the analysis. Lastly, the spec defines a sanitizer on lines 238–241. For the purposes of this example, this sanitizing function

[1] To use sanitizers, the user must enable experimental path tracking with `-p`. Path tracking is experimental because the SPDS source code [47] has some bugs related to path tracking that make it unstable.

[2] Regex does not currently work for sanitizers.

```
[{
    "name": "testing",
    "description": "what this spec detects",
    "advice": "how to solve the issue",
    "sources": [
      {
        "name": "test.source() -> Swift.String",
        "description": "generic source"
      }
    ],
    "sinks": [
      {
        "name": ".*sink.*",
        "args": [0],
        "regex": true,
        "description": "generic sink"
      }
    ],
    "sanitizers": [
      {
        "name": "test.sanitizer(tainted: Swift.String) ->
    Swift.String",
        "description": "generic sanitizer"
      }
    ]
}]
```

Figure 5.1: An example illustrating taint analysis JSON specification in SWAN.

removes any possibly malicious elements from the argument `tainted`.

Figure 5.2 shows a program that utilizes (toy) sink, source, and sanitizer functions for which the spec in Figure 5.1 was written. The source function on line 244 returns a string that we will consider to be tainted. The sink function on line 248 takes two arguments: an argument that should not be tainted called `sensitive` and another argument whose taintedness is irrelevant called `someOtherParam`. The sanitizer function on line 252 takes a variable called `tainted`, sanitizes it, and returns it.

The functions `prog1` and `prog2` on lines 257 and 263 each utilize the source, sink, and sanitizer in different ways to demonstrate SWAN's taint analysis capabilities. The function `prog1` writes a source to the variable `sourced` on line 258. Because the the spec defines the `source` function as a source, any value returned by the function will be tainted, and therefore `sourced` is tainted.

```
244 func source() -> String {
245     return "e.g., manipulatable input";
246 }
247
248 func sink(_ sensitive: String, _ someOtherParam: String) {
249     // ...
250 }
251
252 func sanitizer(tainted: String) -> String {
253     // sanitize "tainted" ...
254     return tainted;
255 }
256
257 func prog1() {
258   let sourced = source();
259   let sanitized = sanitizer(tainted: sourced);
260   sink(sanitized, sourced);
261 }
262
263 func prog2() {
264   let sourced = source();
265   let otherSourced = source();
266   let sanitized = sanitizer(tainted: otherSourced);
267   sink(sourced, sanitized); // vulnerability
268 }
```

Figure 5.2: A Swift program utilizing sources, sinks, and sanitizers. Figure 5.1 contains the program's corresponding taint analysis specification.

Then, the function sanitizes `sourced` by passing it to the `sanitizer` function on line 259. At this point in the program, `sanitized` contains similar information to `sourced` but is no longer tainted because the spec defines the `sanitizer` function as a sanitizer. Lastly, the function gives `sanitized` and `sourced` to the `sink` function on line 260. According to the spec, if the first argument of the `sink` function is tainted, then the analysis will report a defect. In this case, the program sanitizes the first argument to `sink`, and while the second argument to `sink` is tainted, the analysis correctly does not report a defect.

The `prog2` function demonstrates a case where the analysis reports a genuine defect. The function creates two tainted values, `sourced` and `otherSourced`, on lines 264 and 265, respectively. Then, the function sanitizes `otherSourced` using the `sanitizer` function on line 266, thereby leaving `sourced` tainted. Lastly, the function sinks both values and passes the unsanitized value `sourced`

```
[{
    "name": "testing",
    "description": "what this spec detects",
    "advice": "how to solve the issue",
    "paths": [
      {
        "source": {
          "name": "test.source() -> Swift.String",
          "description": "generic source"
        },
        "sink": {
          "name": "test.sink(Swift.String, Swift.String)
    -> ()",
          "description": "generic sink"
        },
        "path": [
          "test.swift:213:3"
        ]
      }
    ]
}]
```

Figure 5.3: Taint analysis results for the program in Figure 5.2 based on the specification in Figure 5.1.

as the first argument. Because the spec defined the first argument as sensitive and the function never sanitized `sourced`, the taint analysis reports a defect on line 267.

SWAN's taint analysis notifies the user of any defects in the command-line output and writes defect information to `taint-results.json` in the `swan-dir` directory. Figure 5.3 shows the `taint-results.json` file for the program in Figure 5.2.[3] Analysis results contain sufficient information for the user to resolve the vulnerability, including why the defect was detected, how to resolve it, where the vulnerability is, and where the possibly malicious information comes from and flows to.

[3]The path on line 284 only contains a single node due to a limitation with path tracking. Normally, the path contains two elements: the location of where the tainted variable was sourced (line 264) and sunk (line 267).

5.2 Typestate Analysis

SWAN's typestate analysis utilizes IDE^{al}, which is SPDS' query engine for tracking object states based on a given finite-state machine (FSM) and seeds. SWAN provides two methods for defining an FSM: either programmatically (with corresponding JSON for taxonomy information) or solely using a JSON configuration file. To determine seeds (i.e., values of interest), SWAN looks for allocations of the type of interest (e.g., the type `File`, which represents a file resource) within the program. SWAN invokes IDE^{al} with the FSM and seeds. After the analysis is complete, SWAN finds the destructing statements for each seed, which are determined by SPDS, and reports a defect to the user if the value is in an error state at any of those statements.

5.2.1 JSON Configuration

SWAN's typestate analysis can be configured entirely in a JSON file (hereafter referred to as a "spec"). The spec allows the user to configure the allocation type of interest, a description of the analysis, advice for fixing detected defects, and an FSM, by defining states and transitions between those states. A state must be defined as an error, initial, and/or accepting state. A transition must be defined from a state to another state based on a method call, which is either the full name of the method or a matching regular expression. Inside the spec, the user may define multiple typestate analyses. To use the typestate analysis mode, the user must run SWAN with the `-e` flag followed by the path to the spec.

Figure 5.4 shows an example spec that is based on the FSM in Figure 2.7 described in Chapter 2.4. The `style` field is set to `0` on line 290 to indicate that this spec is using the JSON-only format (as opposed to `1`, which indicates that a corresponding programmatic typestate analysis exists). The type `File` on line 291 indicates the allocation type of interest. The field `class` is set to `true` on line 292 so that SWAN's analysis will look for `File.__allocating_init()` constructor calls, as opposed to simply looking for `new` instructions that allocate a value of type `File`. Lines 293–295 define the general taxonomy of the

```
[{
  "style": 0,
  "type": "File",
  "class": true,
  "name": "FileOpenClose",
  "description": "Not closing file resources can cause
   resource leaks.",
  "advice": "Close file resources.",
  "states": [
    {
      "name": "INIT",
      "error ": false,
      "initial": true,
      "accepting": true
    },{
      "name": "OPENED",
      "error": true,
      "message": "file left open",
      "initial": false,
      "accepting": false
    },{
      "name": "CLOSED",
      "error": false,
      "initial": false,
      "accepting": true
    }],
  "transitions": [
    {
      "from": "INIT",
      "method": ".*File.open.*",
      "param": "Param1",
      "to": "OPENED",
      "type": "OnCall"
    },{
      "from": "INIT",
      "method": ".*File.close.*",
      "param": "Param1",
      "to": "CLOSED",
      "type": "OnCall"
    },{
      "from": "OPENED",
      "method": ".*File.close.*",
      "param": "Param1",
      "to": "CLOSED",
      "type": "OnCall"
    }]
}]
```

Figure 5.4: An example illustrating typestate analysis JSON specification in SWAN.

```
335 func open_file(_ f: File) {
336   f.open(); // defect - file left open
337 }
338
339 let f1 = File("path/to/f1.txt");
340 open_file(f1);
341
342 let f2 = File("path/to/f2.txt");
343 open_file(f2);
344 f2.close();
```

Figure 5.5: A Swift program that allocates a file resource. Figure 5.4 contains the program's corresponding typestate analysis specification.

spec. This information is displayed to the user to assist them with resolving defects. Lines 296–313 define the FSM states. Lines 314–333 define the FSM transitions and use regular expressions to match the corresponding method calls of the `File` type. The transition fields `param` and `type` are invariables for SWAN's analysis and are included for completeness.

Figure 5.5 shows a program that uses file resources. The program allocates two file resources on lines 339 and 342, opens the resources using a call to `open_file` on lines 340 and 343, and closes only the second resource on line 344. The program never closes the first resource, `f1`, and therefore, using the spec in Figure 5.4, SWAN's typestate analysis reports a defect on line 336.[4] The analysis does not report a defect for `f2` because the program correctly closes `f2` on line 344.

SWAN's typestate analysis notifies the user of any defects in the command-line output and writes defect information to `typestate-results.json` in the `swan-dir` directory. Figure 5.6 shows the `typestate-results.json` file for the program in Figure 5.5.

5.2.2 Energy Inefficient API Misuse Analysis

SWAN's typestate analysis can also be written programmatically and supplemented with a JSON file for taxonomy information. To demonstrate and motivate this, we use an application of SWAN's typestate analysis from our

[4]SWAN reports a defect on line 336 as opposed to line 340 due to a limitation with using SPDS' destructing statements logic.

```
[
  {
    "name": "FileOpenClose",
    "description": "Not closing file resources can cause
    resource leaks.",
    "advice": "Close file resources.",
    "errors": [
      {
        "pos": "test.swift:290:7",
        "message": "file left open",
        "state": "OPENED"
      }
    ]
  }
]
```

Figure 5.6: Typestate analysis results for the program in Figure 5.5 based on the specification in Figure 5.4.

```
import CoreLocation

let locationManager = CLLocationManager()
locationManager.startUpdatingLocation()
locationManager.desiredAccuracy =
    kCLLocationAccuracyHundredMeters
locationManager.distanceFilter = 4096 // inefficient
```

Figure 5.7: A Swift program that uses the Core Location API. Figure 5.8 contains the program's corresponding typestate analysis specification.

previous work: an analysis that detects energy inefficient usages of Apple's iOS Core Location API [13].

The Location API provides various methods of accessing and monitoring the phone user's physical location and activity type, such as the user being airborne or inside a car. An app may access all location and activity information using the `CLLocationManager`. The app may also configure various parameters, such as the location accuracy, minimum distance change to generate a new event, and activity type to monitor. In our previous work, we determined that some of these parameters (or combinations of parameters) use a significant amount of energy and are therefore inefficient and should largely be avoided.

Figure 5.7 shows an example program that use the Location API in various

ways. The program allocates a `CLLocationManager` and then calls `startUpdatingLocation()`, which starts the standard location service and location monitoring. Then, the program sets the accuracy to 100 meters and the distance filter to 4,096 meters. This particular parameter combination is energy inefficient, but individually these parameters are not necessarily energy inefficient. For instance, an accuracy of one kilometre and a distance filter of 4,096 is not considered inefficient, and an accuracy of 100 meters and a distance filter of 16 is not considered inefficient.

This step-by-step process is an excellent application candidate for typestate analysis because the state of the `CLLocationManager` changes at every statement, and there are many possible states for all the different parameter combinations. After line 361, the manager is inactive and is not monitoring the user's location, and only begins to monitor the user's location after line 362. Line 363 sets the manager to a state that represents an accuracy of 100 meters, and finally line 364 sets the manager to a state that represents both an accuracy of 100 meters and a 4,096 meter filter distance.

To define an FSM for an analysis that can track such state, we must consider which states and transitions we need to model. There are 20 different possible parameter combinations (four different accuracies, and five different distance filters), but there are in fact 40 states because until the program calls `startUpdatingLocation()`, the parameters have no effect and the manager is in an "inactive" state. Therefore, there are 20 "active" states and 20 "inactive" states. Furthermore, there are 820 transitions between the states (400 between active states, 400 between inactive states, and 20 transitions between active and inactive states). Writing these states and transitions manually in a JSON typestate specification would be tedious and error-prone. Therefore, we instead wrote a programmatic typestate analysis that generates these states and transitions.

The programmatic aspect not only aids in eliminating tedious configuration, but also in allowing the analysis to have conditional transitions which would otherwise not be possible using a regular JSON typestate spec. The program sets the location manager parameters through setter method calls that

```
[
  {
    "style": 1,
    "name": "StandardLocationService",
    "description": "Certain configurations of the Standard
   Location Service are not optimal.",
    "advice": "Use `kCLDistanceFilterNone` (default),
   `kCLLocationAccuracyKilometer`, or (filter: 16-256,
   accuracy: `kCLLocationAccuracyHundredMeters`).",
    "states": [
      {
        "name": "Hundred_4096",
        "message": "Using
   `kCLLocationAccuracyHundredMeters` and a distance
   filter of 4096 is nonoptimal.",
        "severity": 2
      },
      [...]
    ]
  }
]
```

Figure 5.8: A (partial) typestate analysis taxonomy JSON specification that complements SWAN's programmatic typestate configuration for the Core Location API.

take either the accuracy or filter distance as arguments. The analysis needs to determine to which state to transition based on these argument values, and therefore the method name is not sufficient. The analysis uses a BOOMERANG backwards query on the setter arguments to determine their value and transitions accordingly to the appropriate state. That is, if the query finds a constant value of 16 for the distance filter argument, the typestate analysis will transition the manager to a state that represents the distance filter value of 16.

Figure 5.8 shows the corresponding JSON spec that includes taxonomy information for both the analysis and error states. The spec includes a "severity" rating, which indicates how energy inefficient the configuration is. This analysis reports errors similarly to regular typestate analysis and writes the results to `typestate-results.json` in the `swan-dir` directory.

5.3 Crypto API Misuse Detection

A cryptography API provides developers with the cryptography operations they need to secure user data and provide authentication, such as hashing, encryption, decryption, and digital signature verification. However, these APIs are prone to misuse because, while they provide the needed operations, using them correctly is not trivial. Many studies have shown that developers widely misuse these APIs. Lazar et al. [36] found that the overwhelming majority of the cryptography-related vulnerabilities are due to developer misuse rather than incorrect implementations of the APIs. Many other studies have shown that crypto-API misuse is prevalent in almost all applications or is otherwise a significant security concern [15], [22], [42].

Therefore, to provide developers with crypto API misuse detection for Swift (iOS) apps and to demonstrate SWAN's practicality, we developed a custom analysis that detects crypto API misuses. The analysis supports detecting six different types of misuses for the popular and open-source CryptoSwift API [34].

5.3.1 Crypto Misuse Rules

Egele et al. present six rules for crypto API misuse detection [22]. These rules have become a standard for crypto analysis tools, and therefore we based our analysis on these rules. We list them here and provide a CryptoSwift code example violation for each.

Rule 1: Do not use ECB mode for encryption.

```
try AES(key: _, blockMode: ECB(), padding: _)
```

Rule 2: Do not use a non-random initialization vector (IV) for CBC encryption.

```
CBC(iv: "constant".bytes)
```

Rule 3: Do not use constant encryption keys.

```
HMAC(key: "constant".bytes)
```

Rule 4: Do not use constant salts for password-based encryption (PBE).

```
let salt = "constant".bytes
try HKDF(password: _, salt: salt, ...)
```

Rule 5: Do not use fewer than 1,000 iterations for PBE.

```
try PKCS5.PBKDF1(..., iterations: 500, ...)
```

Rule 6: Do not use static seeds to seed `SecureRandom` (i.e., an Android function that is a psuedo-random number generator). This rule is not applicable to CryptoSwift because there is no API call in CryptoSwift that is analogous to the Android API call for which the rule was specifically written.

Rule 7: Do not use a constant password for encryption. This rule is not included in the original six rules, but we added it because Rule 6 is not applicable.

```
let pwd = "constant".bytes
try HKDF(password: pwd, ...)
```

5.3.2 Misuse Detection

SWAN's crypto analysis consists of custom dataflow queries (or sets of queries) for the various crypto rules. The analysis queries require using forward and/or backward queries to determine if a particular argument adheres to certain constraints. If an argument does not meet a rule's constraints, then the analysis will report a violation of that rule. We outline these constraints and logic for each rule here.

Rule 1: For all initializers that take a block mode, the block mode argument should not come from the ECB initializer. Algorithm 6 shows the highly-simplified logic of the analysis for evaluating this rule. Essentially, the logic is executing a taint analysis from the initialization method of `CryptoSwift.ECB` to any CryptoSwift function calls that take a block mode as an argument.

Algorithm 6 Evaluate Rule 1

```
1: Input
2: – p is the input program
3: procedure EVALUATERULE1(p)
4:     s ← p.getCallSitesWithBlockMode()
5:     c ← all calls to 'CryptoSwift.ECB.init() → CryptoSwift.ECB'
6:     r ← ∀i ∈ c ( forward query i in p )
7:     for x ∈ s do
8:         a ← x.arguments[0]
9:         if ∃z ∈ r ( z flows to a ) then
10:            report violation
```

Rule 2: For all initializers that take an IV, the IV argument should only come from known random functions, such as `Cryptors.randomIV()`. Algorithm 7 shows the simplified logic for this rule. Unlike Rule 1, this logic will report a violation if the analysis does *not* detect dataflow from pre-defined random generator functions to call sites that use an IV.

Algorithm 7 Evaluate Rule 2

```
1: Input
2: – p is the input program
3: procedure EVALUATERULE2(p)
4:     s ← p.getCallSitesWithIVs()
5:     c ← all calls to known random functions
6:     r ← ∀i ∈ c ( forward query i in p )
7:     for x ∈ s do
8:         a ← x.ivArgument()
9:         if ∄z ∈ r ( z flows to a ) then
10:            report violation
```

Rule 3: For all initializers that take a key, the key argument should not be constant. We only report a violation if we find a constant value. Algorithm 8 shows the simplified logic for this rule. The analysis queries any key arguments to determine whether they are constant. Constant values can either be the result of a `literal` instruction or the return value of a static initializer function.

Algorithm 8 Evaluate Rule 3

```
1: Input
2: – p is the input program
3: procedure evaluateRule3(p)
4:     s ← p.getCallSitesWithKeys()
5:     c ← all static initializers
6:     r ← ∀i ∈ c ( forward query i in p)
7:     for x ∈ s do
8:         a ← x.keyArgument()
9:         r ← backward query a in p
10:        if r is constant or r is a static initializer then
11:            report violation
```

Rule 4: For all initializers that take a salt, the salt argument should not be constant. We only report a violation if we find a constant value. The logic for this rule is very similar to Algorithm 8.

Rule 5: For all initializers that take an iteration count, the iteration count argument should be at least 100,000. The rule given by Egele et al. states 1,000 iterations, which is the bare minimum according to the original recommendation given in the standard from the year 2000 [29]. However, given the increased computing power of modern machines, we decided to increase the minimum to 100,000. If the analysis cannot detect a constant integer value, it does not report a violation. The logic for this rule is very similar to Algorithm 8, except that it includes the additional step of checking the constant value, if found.

Rule 7: For all initializers that take a password, the password argument should not be constant. We only report a violation if we find a constant value. The logic for this rule is very similar to Algorithm 8.

SWAN's typestate and taint analysis JSON configuration is not always expressive enough to craft certain types of analyses, such as those that require checking specific values. SWAN's crypto analysis is an example of how the framework may be utilized to write custom, domain-specific analyses. Using SPDS forward and backward queries allows analysis developers to write powerful analyses with specific constraints.

5.4 Summary

SWAN serves as a platform for practitioners and researchers to write their own custom analyses for Swift. SWAN also provides built-in, configurable analyses capable of detecting illegal dataflow from sources to sinks (i.e., taint analysis) and incorrect usage of APIs (i.e., typestate analysis). Our framework's taint analysis can be configured entirely in a JSON file and can be populated with useful taxonomy information to assist developers in resolving any detected issues. SWAN's typestate analysis can similarly be configured entirely inside of a JSON file. However, the JSON configuration may not be expressive enough or practical for certain types of analyses. One such analysis is detecting inefficient configurations of the Core Location API because the analysis requires a large number of states and transitions and requires adding argument value constraints to transitions. Therefore, we wrote a custom programmatic analysis for the Core Location API to serve as an example of what SWAN is capable of as a platform. Lastly, SWAN provides a custom crypto API misuse analysis that detects six different types of rule violations (misuses) of the CryptoSwift API. The crypto analysis utilizes multiple queries together to detect whether arguments given to certain API functions conform to the analysis' constraints, which are based on the crypto rules.

Chapter 6

Evaluation

In this chapter, we evaluate various components of SWAN by running SWAN on open-source Swift applications. We evaluate the runtime of SWAN's processing overhead–the time it takes to process an application and prepare it for analysis. Then, we compare the performance and relative precision of SWAN's three call-graph construction algorithms. We also evaluate the effectiveness of SWAN's crypto analysis. We run the crypto analysis on applications with known crypto violations and report the number and types of detected issues. Lastly, we describe SWAN's testing system to demonstrate how we test and evaluate SWAN's analysis capabilities and language feature support on an ongoing basis.

6.1 Processing Overhead

In this section, we assess the processing overhead in terms of runtime of the "core" framework to evaluate SWAN's general efficiency on a collection of open-source Swift applications. That is, the amount of time from the start of invoking SWAN until the framework is ready for analysis, but not including any analyses, such as call-graph construction. We demonstrate that SWAN's overhead is small and that ultimately SWAN's performance is determined mostly by its analyses. We report the median time taken across the benchmarks. SWAN uses multi-threading to parse and translate each module individually, and therefore calculating the amount of time the parser takes, for instance, would not be useful because each thread finishes at different times. Therefore,

Table 6.1: SWAN processing (pre-analysis) overhead for each benchmark (seconds).

Benchmark	SWAN Overhead	SIL LOC	SIL Functions
ZenTuner	3.04	23,747	966
github-contributions-ios	9.23	344,078	7,939
Kotoba	4.11	49,543	1,566
authenticator	4.53	88,848	2,704
FrameGrabber	8.64	235,805	7,621
Swiftagram	9.41	265,332	5,356
PGPro	5.83	135,547	3,331
daylight-ios	6.95	126,580	3,346
compositional-layouts-kit	3.20	37,123	1,246
flappy-fly-bird	4.32	67,390	2,206
swiftui-2048	3.85	56,570	1,712
FlappySwift	2.67	24,862	635
Tofu	3.81	57,674	1,723
CalendarKit	4.70	69,332	2,384
CompositionalDiffablePlayground.ios	4.34	73,891	2,624
edhita	2.79	23,185	915
watchOS-2-Sampler	3.48	33,013	1,296
iOS-Depth-Sampler	4.88	67,701	2,197
wireguard-apple	7.32	226,708	4,824
Swift-Radio-Pro	5.79	124,747	4,209
swift-2048	2.88	25,526	863
trailer	11.40	341,233	7,747
Median	**4.44**	**68,517**	**2,295**

we only report the total time taken until the final grouped module is ready.

6.1.1 Benchmark Applications

We evaluated SWAN on 22 open-source Swift apps, which are the non-library apps that we were able to build from the largest collaborative collection of Swift apps on GitHub [20]. We chose apps that have at least 100 stars, have been recently updated within the last two years, and that built on our machine. Table 6.1 lists the benchmarks.

6.1.2 Experimental Setup

We ran our experiments on an Apple M1 processor with 4 performance cores, 4 efficiency cores, and 16 GB of unified memory. We used the Hotspot JVM to run SWAN. At the time of writing, binaries for the Hotspot JVM are only available compiled for Intel CPUs, so there is some emulation overhead in our runtimes. Our reported runtimes are likely greater than they would be without emulation. However, we do not know exactly what the emulation overhead is because we did not test without emulation on our machine. We ran SWAN on each benchmark 11 times but we discard the first runtime to eliminate any memory caching bias (i.e., the SIL files may be cached in memory after the first run, which may make subsequent runs faster). To further minimize caching bias, we ran the benchmarks in round-robin fashion until every benchmark was run 11 times in total, as opposed to immediately running a benchmark all 11 times and then moving on to the next benchmark. Therefore, we report the median runtime across 10 runs[1] for each benchmark.

6.1.3 Results

Table 6.1 shows the runtimes in seconds, SIL lines of code (LOC), and number of SIL functions for all benchmarks. SWAN parsed, translated, and prepared for analysis most benchmarks in under 5 seconds. The longest runtime took over 11 seconds for the TRAILER benchmark, which is the second largest benchmark in terms of SIL LOC. Figure 6.1 shows a plot of the runtimes versus benchmark LOC, along with a trend line, demonstrating that SWAN's runtime increases relatively linearly with LOC.

SWAN prepares most benchmark apps for analysis in under 5 seconds.

6.1.4 Discussion

Although most of our apps are not large (under 80,000 SIL LOC), SWAN managed to achieve high performance, even with emulation overhead. In practice, large industry Swift applications are split up into many smaller Xcode

[1] 10 runs is common practice when evaluating benchmark performance.

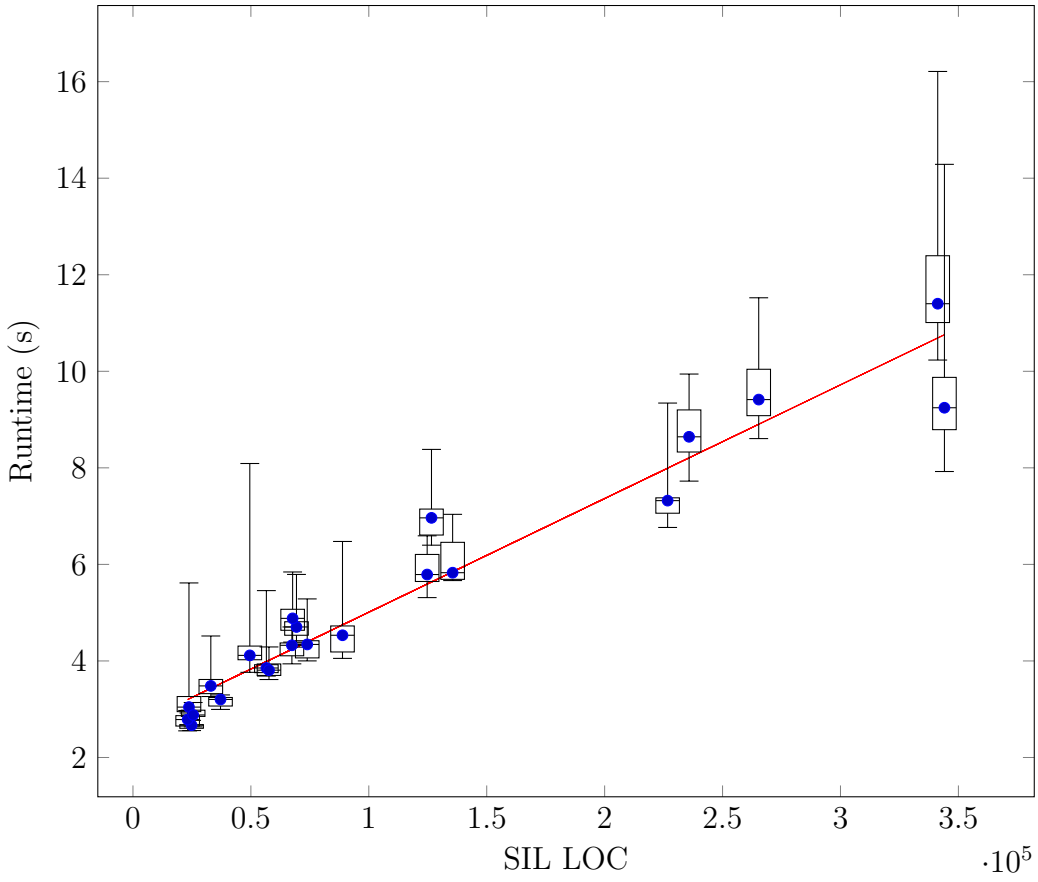

Figure 6.1: Plotted runtimes from Table 6.1.

projects, such as Uber's apps [62], thereby limiting the LOC that SWAN needs to process. Furthermore, with caching optimizations[2], the runtime of SWAN's processing overhead may be reduced even further. SWAN's performance may make it a good fit for continuous integration environments, such as merge request pipelines [21]. However, to reanalyze an application using SWAN, it must first be rebuilt using `(swan-)xcodebuild` so that SWAN can process its updated SIL, and this will add an additional delay to the total time it takes to reanalyze the application[3]. Therefore, SWAN may not be well suited for

[2]We have experimented with caching the canonical SWIRL representation of libraries, requiring SWAN to re-parse only the user code during subsequent SWAN invocations. Many applications have (sometimes significantly) more library code than user code and, with caching, SWAN only needs to completely reprocess the user code. Such caching is currently only experimental.

[3]We do not evaluate the performance of `swan-xcodebuild` because it is a wrapper of `xcodebuild`, which is an Apple tool.

integration into development environments (e.g., IDEs), which require near-immediate feedback.

6.2 Call-Graph Construction

To understand the performance and relative precision of SWAN's suite of call-graph construction algorithms, we evaluated them on a collection of open-source Swift applications. We compared CHA_{FP}, VTA_{FP}, and UCG through the following research questions:

RQ1. Can we increase call-graph precision by using non-propagation, type-based call resolution for dynamic dispatch and type signature matching for function pointers in SIL?

RQ2. Can we increase call-graph precision by using a propagation-based and type-based call resolution approach combined with flow-based pruning?

RQ3. Can we increase call-graph precision by using a flow-based call resolution approach for resolving both dynamic dispatch and function pointers?

RQ4. How do these approaches compare in terms of running time?

6.2.1 Benchmark Applications

We evaluated our suite of algorithms on the same benchmarks for which we evaluated SWAN's processing overhead. Table 6.2 lists these apps and some of their characteristics[4]. We treat any non-library code as being a potential entry point to the program to over-approximate code that may be called by the iOS operating system.

6.2.2 Experimental Setup

Our setup is the same as for evaluating the processing overhead. SWAN's call-graph construction is deterministic. Therefore, the only varying results

[4]We exclude zero values in our geometric mean calculation.

Table 6.2: Various characteristics of our Benchmarks Programs

Benchmark	Call Sites	Functions	Allocations	Function Refs	Non-trivial Function Refs (%)	Dynamic Refs	Non-trivial Dynamic Refs (%)
ZenTuner	1,633	1,079	4,123	1,600	0.94	49	40.82
github-contributions-ios	25,972	8,220	52,967	22,359	1.26	3,614	43.69
Kotoba	3,742	1,576	7,267	3,542	0.56	231	35.93
authenticator	6,956	2,706	13,283	6,823	0.56	103	32.04
FrameGrabber	17,036	7,868	37,549	15,896	0.95	1,224	32.46
Swiftagram	18,816	5,344	38,046	18,307	1.04	570	8.20
PGPro	12,626	3,334	22,487	12,150	0.27	421	27.65
daylight-ios	9,455	3,335	18,162	8,513	0.46	866	39.03
compositional-layouts-kit	3,183	1,246	6,631	2,971	0.84	243	49.79
flappy-fly-bird	5,850	2,206	11,373	5,184	0.31	594	32.71
swiftui-2048	4,361	1,713	9,953	4,334	0.30	38	10.53
FlappySwift	2,531	635	4,576	2,188	0.00	259	24.84
Tofu	5,076	1,723	10,017	4,507	0.27	486	23.72
CalendarKit	5,395	2,416	12,376	5,134	0.04	271	42.44
CompositionalDiffablePlayground.ios	6,631	2,823	13,157	6,214	0.24	413	37.53
edhita	1,920	920	4,407	1,720	0.29	188	69.68
watchOS-2-Sampler	2,447	1,296	5,244	2,258	0.31	199	52.26
iOS-Depth-Sampler	5,481	2,197	10,891	5,242	0.06	201	44.28
wireguard-apple	17,238	4,825	35,260	15,574	0.24	1,483	23.46
Swift-Radio-Pro	9,361	4,210	20,338	8,487	0.33	803	37.64
swift-2048	2,237	863	4,472	1,974	0.00	243	58.44
trailer	26,600	7,741	55,360	26,859	2.72	358	7.00
Geometric Mean	-	-	-	-	**0.40**	-	**30.78**
Mean	**8,843**	**3,103**	**18,088**	**8,265**	-	**584**	-
Median	**5,666**	**2,311**	**11,875**	**5,213**	**0.31**	**315**	**36.73**

are the runtimes. For the runtimes of the call-graph construction algorithms, we ran each benchmark 10 times and we report the median runtime. We do not discard the first runtime in this case because the call-graph construction algorithms do not cache anything in memory that could affect the runtime of subsequent iterations.

6.2.3 Results

RQ1. To answer this question, we compare the call graphs generated by CHA_{FP} with those generated by VTA_{FP}, which uses a more precise strategy.

Table 6.3: Number of reachable nodes and edges in the call graphs computed using CHA_{FP}, VTA_{FP}, and UCG, as well as the precision improvement across the call graphs.

Benchmark	CHA_{FP} Nodes	VTA_{FP} Nodes	CHA_{FP}–VTA_{FP} Improvement (%)	UCG Nodes	VTA_{FP}–UCG Improvement (%)	CHA_{FP} Edges	VTA_{FP} Edges	CHA_{FP}–VTA_{FP} Improvement (%)	UCG Edges	VTA_{FP}–UCG Improvement (%)
ZenTuner	893	828	7.28	792	4.35	1,577	1,374	12.87	1,291	6.04
github-contributions-ios	4,919	1,908	61.21	1,047	45.13	60,119	8,656	85.60	1,752	79.76
Kotoba	1,574	1,543	1.97	1,543	0.00	4,957	3,465	30.10	3,463	0.06
authenticator	2,568	2,484	3.27	2,484	0.00	12,926	6,411	50.40	6,394	0.27
FrameGrabber	7,382	6,697	9.28	6,583	1.70	96,604	13,471	86.06	12,354	8.29
Swiftagram	5,344	5,254	1.68	5,254	0.00	33,225	17,635	46.92	17,635	0.00
PGPro	3,310	3,253	1.72	3,253	0.00	23,410	12,183	47.96	12,038	1.19
daylight-ios	2,569	2,074	19.29	2,074	0.00	11,561	5,532	52.15	5,468	1.16
compositional-layouts-kit	1,241	1,217	1.93	1,217	0.00	3,488	2,924	16.17	2,914	0.34
flappy-fly-bird	2,204	2,167	1.68	2,167	0.00	7,794	5,263	32.47	5,162	1.92
swiftui-2048	1,708	1,701	0.41	1,701	0.00	4,990	4,279	14.25	4,272	0.16
FlappySwift	625	622	0.48	622	0.00	2,957	2,216	25.06	2,216	0.00
Tofu	1,716	1,695	1.22	1,695	0.00	6,520	4,605	29.37	4,494	2.41
CalendarKit	1,102	703	36.21	617	12.23	3,625	1,982	45.32	1,638	17.36
CompositionalDiffablePlayground.ios	2,817	2,749	2.41	2,749	0.00	9,698	6,123	36.86	6,112	0.18
edhita	916	901	1.64	901	0.00	2,825	1,705	39.65	1,703	0.12
watchOS-2-Sampler	611	182	70.21	182	0.00	1,704	188	88.97	188	0.00
iOS-Depth-Sampler	2,035	1,818	10.66	1,818	0.00	7,815	4,037	48.34	4,008	0.72
wireguard-apple	4,553	4,131	9.27	4,110	0.51	65,701	12,868	80.41	12,246	4.83
Swift-Radio-Pro	4,204	4,141	1.50	4,141	0.00	31,909	8,495	73.38	8,368	1.49
swift-2048	857	842	1.75	842	0.00	2,607	2,115	18.87	1,968	6.95
trailer	7,721	7,644	1.00	7,644	0.00	84,024	26,411	68.57	25,898	1.94
Geometric Mean	-	-	**4.56**	-	**4.61**	-	-	**40.25**	-	**1.40**

For each algorithm, Table 6.3 shows the number of reachable nodes from all entry points, as well as the reachable edge counts in the generated call graphs. We calculated geometric means for the improvements between algorithms and ignored zero values in our calculation.

With respect to reachable nodes, VTA_{FP} has small improvements compared to CHA_{FP}, with a geometric mean of 4.56% fewer reachable nodes. However, there are some outliers that have significantly fewer reachable nodes such as the benchmarks GITHUB-CONTRIBUTIONS-IOS (61.21%) and WATCHOS-2-SAMPLER (70.21%). These applications have a high number of spurious edges due to many functions sharing the same type signature. For instance, GITHUB-CONTRIBUTIONS-IOS and its libraries have many methods that override the `==` operator. Whenever the application compares two values using the operator, which the program does 79 times, the applied function pointer uses the generic

type signature for the operator method. Using its type signature matching, CHA_{FP} resolves the function pointer to every == operator method in the program, and there are 54 such methods. Therefore, there are over 4,000 spurious edges[5] from this method usage alone. VTA_{FP} prunes these spurious edges and substantially reduces the number of CG edges.

Reachable edges provide a better indication of the precision of a call graph than reachable nodes. Most functions in an app will likely be in the call graph because SWAN sets all non-library functions as entry points, and a call to a library function will likely transitively call (possibly many) other library functions. VTA_{FP} has a geometric mean of 40.25% fewer edges than CHA_{FP}, with several benchmarks having over 80% fewer edges, such as WIREGUARD-APPLE (80.51%), GITHUB-CONTRIBUTIONS-IOS (85.60%), FRAMEGRABBER (86.06%), and WATCHOS-2-SAMPLER (88.97%).

> VTA_{FP} generally provides better precision than CHA_{FP}, with a geometric mean of 40.25% fewer reachable edges. This significant decrease in reachable edges is due to the heavy use of non-trivial function pointers with similar type signatures in SIL.

Table 6.4 provides detailed data of how many CHA_{FP} polymorphic edges that VTA_{FP} resolves to be monomoporphic (indicating an increase in precision). The more precise VTA_{FP} analysis identifies a mean of 2.00% polymorphic call sites that CHA_{FP} detects to be monomorphic. Notably, for SWIFT-2048 and IOS-DEPTH-SAMPLER, 16.13% and 12.25% of polymorphic call sites, respectively, become monomorphic in VTA_{FP} compared to CHA_{FP}. However, for 16/22 benchmarks, less than 1% of polymorphic call sites become monomorphic.

> Overall, VTA_{FP} identifies a mean of 2.00% of the polymorphic call sites identified by CHA_{FP} to be monomorphic, but most benchmarks see a difference of less than 1%.

Most of the imprecision that CHA_{FP} exhibits is due to its imprecise handling of non-trivial function pointers by using type-signature matching. CHA_{FP}

[5] $79 * (54 - 1)$ because every call site has one real edge and the rest are spurious.

often resolves function pointers to completely unrelated functions. In these benchmarks, the type hierarchies are not very complex. Therefore, these spurious edges do not arise from handling dynamic dispatch in CHA_{FP}, but rather from handling function pointers imprecisely.

RQ2. To answer this question, we compare the call graphs generated by VTA_{FP} with those generated by UCG.

With respect to reachable methods, Table 6.3 shows that UCG has little to no improvements in reachable methods compared to VTA_{FP}, with a geometric mean of 4.61% fewer reachable methods and 17/22 benchmarks seeing no improvement. The benchmark GITHUB-CONTRIBUTIONS-IOS is an exception for which UCG has 45.13% fewer methods than VTA_{FP}.

Across all apps, UCG has a geometric mean of 1.4% fewer edges than VTA_{FP}, with 16/22 benchmark apps having less than a 3% improvement. The two outliers are GITHUB-CONTRIBUTIONS-IOS and CALENDARKIT, which have a 79.76% and 17.36% improvement, respectively. These outliers can be attributed primarily to VTA_{FP} treating library functions as entry points because VTA_{FP} starts with a CG produced by CHA_{FP}, which builds a CG for the entire program. VTA_{FP} pessimistically prunes CHA_{FP}'s edges, whereas UCG is an optimistic algorithm that does not treat library functions as entry points (see Chapter 4.4.1 and Chapter 4.4.3). For instance, GITHUB-CONTRIBUTIONS-IOS has many `toString()` functions on polymorphic types within a library that call each other. CHA_{FP} grossly over-approximates their edges, and VTA_{FP} prunes CHA_{FP}'s edges using its type information. However, those library functions are never called by the user program. Since UCG does not treat them as entry points, it does not have any edges to or from those functions. CALENDARKIT has a similar situation where call sites within library functions have many resolutions due to polymorphism and are pruned by VTA_{FP}, but the user code never calls them and therefore UCG has no edges to them.

UCG generally provides slightly better precision than VTA_{FP}, with a geometric mean of 1.4% fewer edges.

Table 6.4: Number of monomorphic and polymorphic reachable call sites in the call-graphs generated by CHA_{FP} and how many of them become unreachable, monomorphic, and polymorphic in VTA_{FP}.

			VTA_{FP}		
Benchmark	CHA_{FP}		**Unreachable**	**Mono**	**Poly**
github-contributions-ios	**Mono**	14,059	6,429	7,397	0
	Poly	1,084	403	0 (0.00%)	483
wireguard-apple	**Mono**	10,017	1,022	8,486	0
	Poly	2,874	22	14 (0.49%)	2,505
Swift-Radio-Pro	**Mono**	7,210	0	6,768	0
	Poly	721	0	5 (0.69%)	529
FrameGrabber	**Mono**	13,478	1,203	11,599	0
	Poly	1,176	88	60 (5.10%)	717
swift-2048	**Mono**	1,956	0	1,869	0
	Poly	124	0	20 (16.13%)	92
daylight-ios	**Mono**	3,664	820	2,614	0
	Poly	1,729	117	0 (0.00%)	1,387
Tofu	**Mono**	3,985	0	3,800	0
	Poly	474	0	1 (0.21%)	335
FlappySwift	**Mono**	1,550	0	1,435	0
	Poly	471	0	0 (0.00%)	362
compositional-layouts-kit	**Mono**	2,699	0	2,481	0
	Poly	187	0	0 (0.00%)	180
flappy-fly-bird	**Mono**	4,331	0	4,018	0
	Poly	613	0	1 (0.16%)	461
swiftui-2048	**Mono**	3,592	0	3,566	0
	Poly	231	0	8 (3.46%)	201
ZenTuner	**Mono**	1,289	78	1,209	0
	Poly	87	3	0 (0.00%)	75
PGPro	**Mono**	11,357	0	11,186	0
	Poly	538	0	9 (1.67%)	388
authenticator	**Mono**	5,556	29	5,470	0
	Poly	387	0	0 (0.00%)	318
CompositionalDiffablePlayground.ios	**Mono**	5,761	0	5,435	0
	Poly	318	0	1 (0.31%)	273
trailer	**Mono**	23,702	0	23,682	0
	Poly	973	0	3 (0.31%)	645
CalendarKit	**Mono**	3,119	1,085	1,924	0
	Poly	53	4	0 (0.00%)	23
Swiftagram	**Mono**	15,468	54	15,338	0
	Poly	1,125	0	1 (0.09%)	753
iOS-Depth-Sampler	**Mono**	4,494	821	3,528	0
	Poly	253	1	31 (12.25%)	186
watchOS-2-Sampler	**Mono**	1,382	1,122	171	0
	Poly	16	5	0 (0.00%)	8
edhita	**Mono**	1,754	0	1,577	0
	Poly	68	0	2 (2.94%)	47
Kotoba	**Mono**	2,063	0	1,941	0
	Poly	811	0	2 (0.25%)	726
Mean	**Mono**	-	-	-	-
	Poly	-	-	- (**2.00%**)	-

Table 6.5 provides detailed data of how many VTA_{FP} polymorphic edges UCG resolves to be monomoporphic. Unlike VTA_{FP}, UCG's pointer analysis (i.e., SPDS) is context-sensitive, flow-sensitive, and field-sensitive and therefore precisely finds the allocation sites of function pointers. This sensitivity enables UCG to build a precise CG through which UCG propagates instantiated types in a flow-sensitive manner. UCG analysis identifies a mean of 8.63% polymorphic call sites that VTA_{FP} detects to be monomorphic. Notably, for WIREGUARD-APPLE and CALENDARKIT, 71.86% and 47.83% of polymorphic call sites, respectively, become monomorphic in UCG compared to VTA_{FP}. However, for 14/22 benchmarks, less than 1% of polymorphic call sites become monomorphic.

Overall, UCG identifies a mean of 8.63% of the polymorphic call sites identified by VTA_{FP} to be monomorphic, but most benchmarks see a difference of less than 1%.

RQ3. To answer this question, we compare the call graphs generated by VTA_{FP} with those generated by both CHA_{FP} and UCG. As we previously showed using Table 6.3, VTA_{FP} has 40.25% fewer reachable edges than CHA_{FP}, and UCG has 1.4% fewer reachable edges than VTA_{FP}. Therefore, VTA_{FP} is between CHA_{FP} and UCG in terms of precision, but is far closer to UCG than CHA_{FP}. The improvement from CHA_{FP} to VTA_{FP} removes several spurious edges that CHA_{FP} computes due to its type signature function pointer resolution strategy. However, VTA_{FP} still has some imprecision due to its field-based and flow-insensitive nature.

VTA_{FP} offers a significant increase in precision compared to CHA_{FP}, with a geometric mean of 40.25% fewer edges, but is slightly less precise than UCG with a geometric mean of 1.4% more edges.

Table 6.5: Number of monomorphic and polymorphic reachable call sites in the call-graphs generated by VTA_{FP} and how many of them become monomorphic and polymorphic in UCG.

Benchmark	**VTA_{FP}**		**UCG**	
			Mono	**Poly**
github-contributions-ios	**Mono**	7,397	1,545	0
	Poly	483	22 (4.55%)	82
wireguard-apple	**Mono**	8,500	8,126	0
	Poly	2,505	1,800 (71.86%)	675
Swift-Radio-Pro	**Mono**	6,773	6,681	0
	Poly	529	0 (0.00%)	523
FrameGrabber	**Mono**	11,659	10,752	0
	Poly	717	64 (8.93%)	580
swift-2048	**Mono**	1,889	1,742	0
	Poly	92	0 (0.00%)	92
daylight-ios	**Mono**	2,614	2,614	0
	Poly	1,387	30 (2.16%)	1,338
Tofu	**Mono**	3,801	3,690	0
	Poly	335	0 (0.00%)	335
FlappySwift	**Mono**	1,435	1,435	0
	Poly	362	0 (0.00%)	362
compositional-layouts-kit	**Mono**	2,481	2,481	0
	Poly	180	0 (0.00%)	178
flappy-fly-bird	**Mono**	4,019	3,918	0
	Poly	461	0 (0.00%)	461
swiftui-2048	**Mono**	3,574	3,567	0
	Poly	201	0 (0.00%)	201
ZenTuner	**Mono**	1,209	1,126	0
	Poly	75	13 (17.33%)	62
PGPro	**Mono**	11,195	11,070	0
	Poly	388	0 (0.00%)	382
authenticator	**Mono**	5,470	5,453	0
	Poly	318	12 (3.77%)	306
CompositionalDiffablePlayground.ios	**Mono**	5,436	5,425	0
	Poly	273	0 (0.00%)	273
trailer	**Mono**	23,685	23,684	0
	Poly	645	3 (0.47%)	639
CalendarKit	**Mono**	1,924	1,618	0
	Poly	23	11 (47.83%)	4
Swiftagram	**Mono**	15,339	15,339	0
	Poly	753	0 (0.00%)	753
iOS-Depth-Sampler	**Mono**	3,559	3,530	0
	Poly	186	0 (0.00%)	186
watchOS-2-Sampler	**Mono**	171	171	0
	Poly	8	2 (25.00%)	6
edhita	**Mono**	1,579	1,577	0
	Poly	47	0 (0.00%)	47
Kotoba	**Mono**	1,943	1,941	0
	Poly	726	0 (0.00%)	726
Mean	**Mono**	-	-	-
	Poly	-	- (**8.63%**)	-

Table 6.6: Median runtimes for each algorithm (milliseconds)

Benchmark	**CHA$_{FP}$**	**VTA$_{FP}$**	**UCG**	**VTA$_{FP}$/ CHA$_{FP}$**	**UCG/ VTA$_{FP}$**
ZenTuner	109	805	824	7.4	1.0
github-contributions-ios	799	22,860	2,645	28.6	0.1
Kotoba	162	1,201	925	7.4	0.8
authenticator	285	1,351	3,835	4.7	2.8
FrameGrabber	841	9,173	12,061	10.9	1.3
Swiftagram	546	6,657	54,179	12.2	8.1
PGPro	320	2,253	5,538	7.0	2.5
daylight-ios	284	1,585	1,576	5.6	1.0
compositional-layouts-kit	120	1,039	882	8.7	0.8
flappy-fly-bird	213	1,587	1,988	7.5	1.3
swiftui-2048	183	1,303	1,560	7.1	1.2
FlappySwift	114	906	526	7.9	0.6
Tofu	212	940	1,733	4.4	1.8
CalendarKit	188	1,528	953	8.1	0.6
CompositionalDiffablePlayground.ios	260	1,170	1,799	4.5	1.5
edhita	125	883	600	7.0	0.7
watchOS-2-Sampler	167	994	385	6.0	0.4
iOS-Depth-Sampler	219	1,516	1,659	6.9	1.1
wireguard-apple	521	7,692	9,178	14.7	1.2
Swift-Radio-Pro	370	2,552	3,243	6.9	1.3
swift-2048	100	872	1,729	8.7	2.0
trailer	712	12,562	87,701	17.6	7.0
Geometric Mean	-	-	-	**8.1**	**1.2**
Mean	**311**	**3,701**	**8,887**	-	-
Median	**216**	**1,433**	**1,731**	-	-

RQ4. Table 6.6 shows the median running times for our suite of algorithms for all benchmark apps. Figure 6.2 shows a plot of the running times in Table 6.6. The results have several outliers across the algorithms (e.g., github-contributions-ios, trailer, Swiftagram). Therefore, we use median values for this comparison. Factors that may create such outliers include heavy usage of interprocedural function pointers and app size (in terms

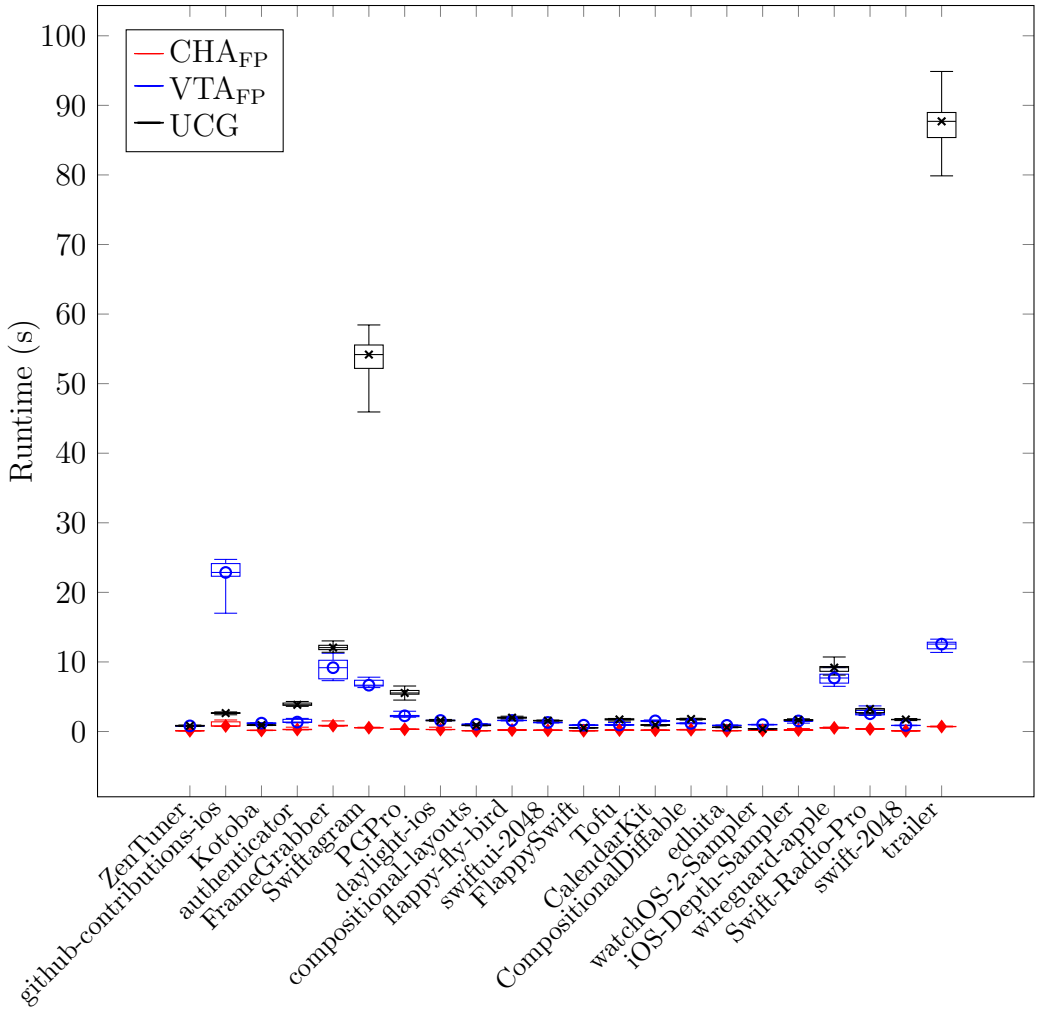

Figure 6.2: Plotted runtimes from Table 6.6.

of LOC count), which are shown in Table 6.2. GITHUB-CONTRIBUTIONS-IOS, TRAILER, and SWIFTAGRAM are the three largest apps, with LOC ranging from 344,078 to 265,332, and are all outliers. Furthermore, some of them have many interprocedural function pointers. GITHUB-CONTRIBUTIONS-IOS has 281 non-trivial function references and 1,579 non-trivial dynamic references. TRAILER has a combined total of 756 non-trivial references. Lastly, SWIFTAGRAM has a combined total of 237 non-trivial references.

Compared to CHA$_{FP}$, VTA$_{FP}$ is 8.1× slower using a geometric mean. The difference varies between 4.4× for TOFU and 28.6× for GITHUB-CONTRIBUTIONS-IOS.

The difference in running times between VTA_{FP} and UCG also varies widely. However, the difference is smaller compared to the difference between CHA_{FP} and VTA_{FP}, with a geometric mean difference of 1.2×. Across all benchmarks, the median running time of VTA_{FP} is 1,433 ms while UCG has a median running time of 1,731 ms. In seven benchmarks, UCG performs better than VTA_{FP}, but with a maximum runtime difference of 0.1× (GITHUB-CONTRIBUTIONS-IOS). On the other hand, VTA_{FP} outperforms UCG for AUTHENTICATOR (2.8×), SWIFTAGRAM (8.1×), PGPRO (2.5×), and TRAILER (7.0×). Nevertheless, UCG still computes call graphs in under 13 seconds for all but two benchmarks (SWIFTAGRAM and AUTHENTICATOR).

For all benchmarks, CHA_{FP} computes its call graph in under 1 second. VTA_{FP} is 8.1× slower than CHA_{FP}. UCG is 1.2× slower than VTA_{FP}. While UCG is slower than VTA_{FP} for some benchmarks, VTA_{FP} and UCG finish their call graph analysis in under 13 seconds for most benchmarks. Therefore, both algorithms generally have similar performance in practice.

6.2.4 Discussion

Our algorithms provide multiple options for Swift call graph construction, each with their own benefits. We found that CHA_{FP} is fast but is imprecise compared to VTA_{FP} and UCG. VTA_{FP} is much more precise than CHA_{FP} but is less performant. Lastly, UCG is slightly more precise than VTA_{FP} but may be less performant in some cases. Therefore, VTA_{FP} is likely the best algorithm to use in most situations. However, UCG provides the best precision and is never less precise than VTA_{FP}, making it the best choice in situations where CG precision is very important. For instance, if we use SWAN's call-graph construction for a dead-code elimination analysis to reduce app binary size [14], UCG's increased precision may help reduce binary size better than VTA_{FP} would.

Table 6.7: Detected crypto API misuses in the benchmark applications.

Benchmark	Rule 1 ECB	Rule 2 IV	Rule 3 KEY	Rule 4 SALT	Rule 5 ITER	Rule 7 PWD
UniversalFrame		2	2			
xcode-projects		1	1			
SwiftyLocalKit	2		2			
InvestScopio		2	2			
SujungVillage-User-iOS		3	3			
open_imsdk_for_swift	2					
CCCryp	2					
encrypt	2		2			
mqttChat	2		2			
lhcp	2		2			
SwiftBasicKit	2		2			
HDWallet					1	
tlatia				1	1	1
Total (44)	**14**	**8**	**18**	**1**	**2**	**1**

6.3 Crypto API Misuse in Real-World Apps

We evaluated SWAN's crypto analysis on collection of open-source Swift applications that use the CryptoSwift API and have crypto misuses. In this section, we report our results and show examples of misuses from the applications.

6.3.1 Benchmark Applications

The benchmarks that we used for this portion of the evaluation are different than the ones we used for evaluating processing overhead and call-graph construction. We focused on benchmarks that use crypto APIs and have crypto misuses. We manually searched GitHub repositories using specific queries,

such as ‘‘`import CryptoSwift`’’ `AES language:Swift` and ‘‘`import CryptoSwift`’’ `PBKDF2 salt language:Swift`. This resulted in 4,960 code results. We limited our search to the first three pages (30 results) of the search results. We looked through the first 30 results and ignored redundant repositories and code results inside of the CryptoSwift library (many repositories include the full CryptoSwift library). In total, we found 57 applications across all of the queries that, using manual inspection, contain crypto API misuses. Out of those 57 applications, we only managed to successfully build 13. Table 6.7 lists the benchmarks along with the number of violations for each rule. Some notable benchmarks include InvestScopio, an app available on the App Store that predicts returns on investments, SujungVillage-User-iOS, a dormitory service managing app, and mqttChat, a generic chat application. Some apps provide utilities for use in other applications, such as SwiftyLocalKit, SwiftBasicKit, CCCryp, and tlatia, which is a password-based utility for encrypting files.

Our benchmark selection process is biased towards applications with manually verified crypto misuses. We did not select benchmarks that correctly use the CryptoSwift API. Therefore, we cannot (and do not) use our selected benchmarks for evaluating whether SWAN can avoid false-positives (i.e., cases where there are no API misuses but SWAN still reports misuses).

6.3.2 Results

SWAN's crypto analysis detected 44 defects in total across all benchmarks. We used CHA_{FP} for call-graph construction, but using VTA_{FP} or UCG instead has no effect on our analysis results. Rule 1 (usage of ECB block mode) and Rule 3 (constant key) are the most frequent violations, and Rule 4 (constant salt) and Rule 7 (constant password) are the least frequent. We manually inspected and compared each benchmark with SWAN's results and determined that all but one detected violations are true positives. We manually examined the arguments passed to supported crypto API calls and did not find any false negatives.

SWAN discovered three different types of violations (constant salt, low

```
381 struct Key {
382   let keyLength = 32
383   let password: String
384   let salt: String = "icGFEkNbIuNK1cz1qNq5kMV1jqkpvduX"
385   let keyDerivation: PKCS5.PBKDF2
386
387   var key: [UInt8] {
388     do {
389       return try keyDerivation.calculate()
390     } catch {
391       return []
392     }
393   }
394
395   init? (withPassword password: String) {
396     self.password = password
397     do {
398       keyDerivation = try PKCS5.PBKDF2(
399         password: Array(self.password.utf8),
400         salt: Array(salt.utf8),
401         keyLength: keyLength)
402     } catch {
403       return nil
404     }
405   }
406 }
```

Figure 6.3: A simplified excerpt from the TLATIA benchmark that contains crypto API misuses.

iteration count, and constant password) in TLATIA. Figure 6.3 shows the violating code from the application's `Key.swift` file. SWAN reports the three violations on line 398. The struct `Key` sets its `salt` field to a constant string on line 384. This field goes through two transformations on line 400 before passing it to the `PKCS5.PBKDF2` constructor: `String.utf8()` and conversion to a byte array (`Array.init()`). Since SWAN models both of these functions, the analysis detects dataflow from the original salt string to the salt parameter of the `PKCS5.PBKDF2` constructor. SWAN also reports a low iteration count because the program does not provide an iteration count to the constructor, and the default value for the iteration count is 4,096, which is below the analysis' minimum threshold of 100,000 (the minimum amount of iterations required for secure encryption–see Chapter 5.3.2). The final violation is a constant password. However, this is in fact a false-positive because the

```
407 fileprivate let key = "app"+".live"+".content".md5()
408
409 public class ZCryptoContentManager: NSObject {
410   public static func encrypt(_ str: String) -> String {
411     do {
412       let aes = try AES(key: Padding.zeroPadding.add(to:
      key.bytes, blockSize: AES.blockSize), blockMode:
      ECB())
413
414       let encrypted = try aes.encrypt(str.bytes)
415       guard let encryptedBase64 = encrypted.toBase64()
      else {
416           return str
417       }
418       return encryptedBase64
419     } catch let error { ... }
420     return str
421   }
422 }
```

Figure 6.4: A simplified excerpt from the SwiftBasicKit benchmark that contains crypto API misuses.

program never sets the `password` field to a constant value (the program gives `Key.init()` a password from the user). SWAN reports a violation because, at the SIL level, the password field has a default constant value, even though in practice it is not possible for the program to use the default value. This is the only false-positive that SWAN reported in the benchmarks. Avoiding this type of false-positive requires modifying SWAN's crypto analysis to avoid reporting a violation involving default values auto-generated by SIL. However, these are not the same as default values specified by the API, such as the default iteration count, because those could result in genuine violations.

Figure 6.4 shows a simplified excerpt from the SwiftBasicKit benchmark, where SWAN found two ECB usage violations and two constant key violations. Figure 6.4 only shows one of each violation on line 412. The program gives `ECB` as a direct parameter, but the `key` parameter dataflow is more complex. The `key` value is defined on line 407 as a concatenation of multiple static strings, and then the program hashes the concatenated string using MD5. MD5 is not considered secure [61], so SWAN models `String.md5()` to simply return the string. Therefore, SWAN continues to treat `key` as a con-

```
423 // enryptionFunction.swift
424 func aesEncrypt(key: String) throws -> String {
425   let key: [UInt8] = Array(key.utf8) as [UInt8]
426   let aes = try! AES(key: key, blockMode: ECB(),
       padding:.pkcs5)
427   let encrypted = try aes.encrypt(Array(self.utf8))
428   return encrypted.toBase64()
429 }
430
431 // ViewController.swift
432 class ViewController: UIViewController {
433   override func viewDidLoad() {
434     super.viewDidLoad()
435     let value = "My value to be encrypted"
436     let key = "MySixteenCharKey"
437     let encryptedValue = try! value.aesEncrypt(key: key)
438   }
439 }
```

Figure 6.5: A simplified excerpt from the ENCRYPT benchmark that contains crypto API misuses.

stant string after the hashing. Next, the program pads the key's bytes with zeros to fit the block size of AES on line 412. SWAN models the padding function to simply return the first argument, which in this example is a constant string. Therefore, SWAN detects that the program gives a constant key to `AES()`, which is an API misuse.

Figure 6.5 shows a multi-file, interprocedural constant key misuse on line 426 from the ENCRYPT benchmark. The same line also contains a simple ECB usage violation. This particular benchmark is a small, example-style application. The program defines the key as a constant string on line 436 and then passes it to `aesEncrypt()` on line 437. On line 425, the key goes through two transformations: `String.utf8()` and conversion to a byte array (`Array.init()`), similarly to Figure 6.3. SWAN has models for these two functions and has interprocedural dataflow support, and therefore determines that the `key` argument on line 426 comes from the constant string value on line 436.

SWAN detects 44 crypto API miuses in total across all benchmarks, with only one false-positive.

6.3.3 Discussion

SWAN's crypto analysis finds various misuses in real-world applications, with very limited false-positives. The misuses' dataflow ranges from direct function call arguments, to string concatenation, to hashing and data type transformations, to interprocedural dataflow. Without dataflow analysis and function modelling, SWAN would not find many of these misuses. Our results demonstrate SWAN's analysis capabilities and its ability to find real misuses.

Due to our limited sample size, we cannot make any conclusions about the general distribution of crypto misuses in Swift applications using our results. However, based on our results in merely 13 tested applications and the total number of candidate benchmarks (57) that we found to have misuses, we believe that crypto API misuses of the CryptoSwift library are a significant problem in Swift applications. SWAN can help find these misuses, which are also security vulnerabilities and may potentially lead to data breaches if not resolved.

6.4 Regression Test Suite

In this section, we describe our testing infrastructure and tests that we use on an ongoing basis to verify SWAN's functionality and precision. These tests are handcrafted and anecdotal, but allow us to verify and evaluate individual components of SWAN. Furthermore, the infrastructure allows us to avoid regression in terms of precision and functionality. These tests themselves are a small contribution for Swift analysis because they provide a micro-benchmark for Swift that other (future) Swift analysis tools could potentially utilize to verify their functionality and precision.

6.4.1 Test Suite

SWAN uses a suite of tests for various language features and analysis capabilities. The suite includes tests for verifying that SWAN can track data through Swift's containers: `Array`, `Dictionary`, and `Set`. Sensitive data may be stored

```
440 func test_shuffled() {
441   let src = source(); //!testing!source
442   let arr1 = ["a", src, "b", "c", "d"];
443   let arr2 = arr1.shuffled();
444   sink(sunk : arr2[0]); //!testing!sink
445 }
446
447 func test_findFirst() {
448   let src = source(); //!testing!source
449   let arr = ["a", src, "b", "c", "b", "c"];
450   let FirstElement = arr.first(where: {$0 == "b" });
451   sink(sunk : FirstElement!); //!testing!sink
452 }
```

Figure 6.6: Two tests from SWAN's test suite that verify SWAN can track dataflow through Swift's arrays.

in these containers and therefore SWAN's models feature extensive coverage of these containers.

Figure 6.6 shows two such tests from SWAN's test suite for the `Array` container. The function on line 440 tests dataflow through the `Array.shuffled()` method, and the function on line 447 tests dataflow through the `Array.first()` method. SWAN treats arrays as objects that can hold a single value (see Section 3.6). Therefore, SWAN models `Array.shuffled()` to simply return the array without modifying it. SWAN also models `Array.first()` to return the array's (single) value. We expect that SWAN should detect a vulnerability on line 440 because after shuffling the array, the first value may indeed be the tainted value. On the other hand, the vulnerability in `test_findFirst()` on line 451 is technically a false-positive because the first element that matches the condition on line 450 would be the third element in the array, `"b"`. However, because SWAN does not aim to precisely handle arrays, this false-positive result is acceptable.

SWAN has similar dataflow tests for language features such as recursion, closures, dynamic dispatch, objects with fields, and Swift's first-class function objects. The test suite also has typestate tests for SWAN's analysis for detecting inefficient configurations of the Core Location API (see Section 5.2.2). Lastly, the test suite has individual tests for each of the crypto API misuse rules that SWAN supports. These tests usually feature both simple and com-

plex dataflow.

6.4.2 Annotation Tester

SWAN features a standalone annotation verification CLI tool for its tests that scans the source code for annotations, compares them to the results, and then outputs any discrepancies. Examples of such annotations are seen in Figure 6.6 on lines 441, 444, 448 and 451, which are all taint analysis annotations. Other types of available annotations are typestate annotations (e.g., `?FileOpenClose?error` for the spec in Figure 5.4) and crypto annotations (e.g., `$KEY$error` for a constant key violation). The annotations may optionally be labelled as false-positives or false-negatives by adding `!fp` or `!fn` to the annotation, respectively. Since these annotations are line-number agnostic, they allow for tests to be easily modified. Furthermore, these annotations may be added to (benchmark) applications and are not limited to the context of SWAN's test suite.

Chapter 7

Related Work

7.1 Analysis Frameworks

In this section, we describe relevant tools and frameworks. SWAN serves as a general Swift static analysis framework, features enhanced call-graph construction approaches, and may be applied to domain-specific problems, such as crypto API misuses. Therefore, we discuss related work in each of these areas.

7.1.1 Android Analysis Tools

The Android platform has seen an abundance of analysis frameworks over the past decade. FlowDroid [8] is a lifecycle-aware, context-sensitive, flow-sensitive, field-sensitive, and object-sensitive taint analysis tool for Android apps. SCanDroid [11] uses WALA [63] and matches Android app manifests to dataflow analyses to ensure that apps do not overreach their permissions. DroidInfer [28] uses context-free language reachability to perform type-based and context-sensitive taint analyses for Android apps. While all these frameworks work well for the Android platform, there is no openly-available equivalent counterpart for the Swift platform. SWAN bridges this gap by providing a robust open-source static analysis framework for Swift. SWAN also borrows SPDS [48] from the Java/Android analysis work as its analysis engine.

7.1.2 LLVM-Based Analyses

While LLVM [39] and Clang [38] support some low-level analyses, they are not suitable for deeper analyses of Swift applications such as precise taint tracking. This is because most Swift-specific structures and source information are typically lost during the compilation of Swift source code to low-level LLVM IR, which makes reporting errors back to the user in the original source not possible to the best of our knowledge. Moreover, the most useful analyses that Clang provides (i.e., memory sanitizer and thread sanitizer) are primarily dynamic analyses. Unlike static analyses, dynamic analyses require running the Swift program under analysis multiple times with various inputs to ensure enough coverage of the program behaviour. SWAN overcomes this limitation by providing a framework for static analysis of Swift programs.

Because Swift compiles to LLVM IR, one may use an LLVM IR analysis tool to analyze Swift programs. The Phasar framework [44] provides call graph construction and dataflow analyses on LLVM IR, theoretically enabling it to analyze Swift applications. However, an analysis targeting LLVM IR immediately faces practical concerns when applied to Swift. For example, it is easy to acquire LLVM IR for a single Swift file, but acquiring LLVM IR for an Xcode project, which is the primary Swift program source format, using the compiler frontend is not possible to the best of our knowledge. Even if acquiring LLVM IR was possible, the IR does not contain source information, making reporting useful results back to the developer difficult. Therefore, applying an LLVM-based analysis framework to Swift is impractical.

7.1.3 Swift Analysis Tools

Many publicly available analysis tools for Swift are linters such as SwiftLint [52]. These tools only help enforce Swift code standards and best practices. Furthermore, these tools are not extendable to provide sophisticated analysis for detecting defects such as security vulnerabilities.

Most existing analysis approaches for iOS apps tend to consume app binaries through decompilation [19], [23], [65]. While these approaches are useful

for detecting specific properties, they do not provide a call graph based on the source code or with source information.

Many industry static application security testing (SAST) products offer Swift support but require a license to use. For instance, SonarSwift [46] is a static Swift code analyzer which allows users to define rules for bugs, code smells, and vulnerabilities to find in their codebase. However, Swift is not supported in the free version of the software and requires a paid license. As a result, we are unable to verify its correctness and effectiveness at Swift static analysis. Other tools that support Swift but require a paid license to use include Coverity [53], Checkmarx [16], and Fortify SCA [40].

Recently, GitHub made their Swift analysis for CodeQL open-source [26]. However, at the time of writing, they have not yet announced or officially released Swift support nor added Swift support to their user-facing toolchain. Despite this, we built their Swift toolchain using CodeQL's development tools. CodeQL supports the same crypto API misuse rules that SWAN supports. Therefore, we ran CodeQL on the apps we used for evaluating SWAN's crypto support. Similar to SWAN, CodeQL uses an auto-builder tool that uses `xcodebuild` to extract Swift code during compilation. However, CodeQL's source code extractor fails to extract user code from the apps. In our testing, CodeQL either failed to build the application, detected no defects, or detected defects only in the CryptoSwift library and not inside user code. We suspect that CodeQL's code extractor for Swift is not yet fully ready for use because it does not correctly select the build scheme. SWAN requires the user to provide the build scheme whereas CodeQL tries to automatically select one. Due to these problems we encountered, we could not include a comparison to CodeQL's Swift analysis in our evaluation.

7.2 Call-Graph Construction

Given that SIL uses function pointers for dynamic dispatch, existing call-graph construction strategies for OOLs, without any modification to handle function pointers, are not sufficient. The most popular and established analyses, Class

Hierarchy Analysis (CHA) [18], Rapid Type Analysis (RTA) [12], and Variable Type Analysis (VTA) [51], all do not resolve function pointers. CHA and RTA would not handle non-trivial function pointers in any way because they would not know the allocation sites of non-trivial function pointers. VTA would resolve the operands of dynamic dispatch instructions, but if the resulting function pointer is non-trivial and requires dataflow analysis, then it too cannot resolve where the function pointer is applied.

Existing work that addresses function pointers, such as Phasar [45], often use hard-and-fast solutions, for instance by considering the type signature of a function pointer and resolving to all functions that match that signature. Such solutions, while sound, are not very precise because many unrelated functions can share the same type signature (e.g., a function with no arguments or a return value). Therefore, these solutions will compute highly imprecise call-graphs for SIL.

While prior work have tackled function pointers in C using sophisticated analysis techniques [9], [24], [35], [41], C does not feature dynamic dispatch. Therefore, these solutions are also not sufficient for Swift/SIL. Moreover, SIL function pointers may be written to fields, and thus we require a field-sensitive pointer analysis, which existing solutions do not use. Others call-graph construction algorithms for C++, such as RTA, have entirely ignored C++'s function pointers, likely due to their relatively infrequent usage.

7.3 Crypto API Misuse

Krüger et al. [32] have developed a static analysis tool called *CogniCrypt*, which automatically detects crypto-API misuses in Java/Android applications. CogniCrypt can also assist in fixing the misuses and can generate code for the user that securely implements various operations, such as data encryption. Their work uses a domain-specific language (DSL), called *CrySL* [33], which allows their analysis to extend to other APIs easily. CogniCrypt integrates into the Eclipse IDE, and it is the first integrated tool to detect misuses *and* generate secure code patterns. It is a good reference for developing similar

tools for other languages and platforms. However, while SWAN uses the same analysis engine as CogniCrypt (SPDS), porting CogniCrypt to support Swift requires significant effort and was out of the scope of this work. Moreover, it is not clear whether such a port would make sense due to the difference between Java/Android crypto APIs and Swift crypto APIs.

Most existing solutions that detect crypto API misuse in iOS apps use the binary analysis approach. This approach is fundamentally inferior to the source-code approach we use because little to no source information can be provided back to the developer when a misuse is detected. Furthermore, this approach faces many engineering challenges because decompilation of iOS apps is difficult and never perfect. Feichtner et al. decompile ARM binaries to LLVM IR and use slicing techniques to determine whether a crypto API call has the correct parameters based on the same fundamental six rules that our work uses [25]. They identified that 82% of the iOS apps they analyzed had at least one API misuse. Their work finds misuses of the iOS *CommonCrypto* library, whereas SWAN's crypto analysis finds misuses of the CryptoSwift library. Therefore, we did not compare SWAN to their analysis tool. Furthermore, to our knowledge, our work is the first to find misuses of CryptoSwift. Li et al. use a combination of static and dynamic analysis to detect crypto API misuses in iOS apps by analyzing runtime call traces [37]. Their work predates Swift and therefore we also did not compare SWAN with it.

Chapter 8

Conclusion

Static analysis reasons about a program's behaviour without executing the program's code and may be used to detect security vulnerabilities, such as cryptography API misuses that may expose the program to attacks. Despite Swift being the most popular programming language for developing iOS applications and iOS devices having a large market share [50], there has been a lack of open-source static analysis tools for Swift analysis. While some commercial tools [16], [53] do offer analysis support for Swift, they are impractical for the static analysis research community due to their high cost and closed-source nature. To address this, in this thesis, we presented SWAN, a static analysis framework for Swift. SWAN features configurable taint and typestate analysis, call-graph construction, and crypto API misuse analysis. SWAN's analysis can track dataflow across Swift modules and through libraries, and SWAN's modelling functionality allows modelling black-box functions, such as Swift standard library functions.

We evaluated various components of SWAN on 22 open-source Swift applications to assess SWAN's performance, precision, and effectiveness on real-world applications. Together, these aspects determine the overall practicality of SWAN as a static analysis framework. We assessed SWAN's overhead and determined that SWAN prepares the majority of benchmarks for analysis in under 5 seconds. After SWAN processes and prepares a benchmark, it is ready to run an analysis, such as call-graph construction. SWAN computes call-graphs using a suite of call-graph construction algorithms that vary in

performance and consists of CHA_{FP}, VTA_{FP}, and UCG. Therefore, we also evaluated the algorithms on the 22 benchmarks. Our slowest but most precise algorithm, UCG, builds a call-graph in under 13 seconds for all but two benchmarks. VTA_{FP}, which is nearly as precise as UCG, builds a call-graph in under 9 seconds for all but two benchmarks. Lastly, CHA_{FP} builds a call-graph in under 1 second for all benchmarks, thereby providing the quickest but least precise call-graph. Lastly, we tested SWAN's crypto API misuse analysis on a different set of 13 open-source applications with known crypto API misuses. SWAN finds 44 misuses in total across all 13 apps, with only one false-positive, demonstrating that SWAN can find crypto API misuses in real Swift applications. To detect some of these misuses, SWAN tracks dataflow interprocedurally, through multiple files, and through data manipulation functions (e.g., concatenation, hashing, and string to byte conversion), which we have modelled.

SWAN serves as a practical platform for Swift analysis that developers and researchers can utilize to construct call-graphs and detect defects in Swift applications. Furthermore, SWAN may be extended to support other types of analyses, and therefore enables its users (e.g., the static analysis community) to write further analyses for Swift. Since SWAN's architecture is modular and features its own intermediate representation (IR), SWIRL, this leaves the door open for integration into other analysis engines/frameworks, whereby SWIRL could be translated to the IR of another engine.

References

[1] Apple, *Ios 16*, https://www.apple.com/ca/ios/, Accessed: Apr. 4, 2023.

[2] ——, *Macos ventura*, https://www.apple.com/ca/macos/, Accessed: Apr. 4, 2023.

[3] ——, *Package manager*, https://www.swift.org/package-manager/, Accessed: Mar. 9, 2023.

[4] ——, *Swift*, https://developer.apple.com/swift/, Accessed: Mar. 9, 2023.

[5] ——, *Swift intermediate language (sil)*, https://github.com/apple/swift/blob/master/docs/SIL.rst, Accessed: Mar. 9, 2023.

[6] ——, *Swift standard library*, https://developer.apple.com/documentation/swift/swift-standard-library, Accessed: Mar. 9, 2023.

[7] ——, *Xcode 14*, https://developer.apple.com/xcode/, Accessed: Mar. 9, 2023.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Programming Language Design and Implementation (PLDI)*, 2014, pp. 259–269. DOI: 10.1145/2594291.2594299. [Online]. Available: https://doi.org/10.1145/2594291.2594299.

[9] D. C. Atkinson and W. G. Griswold, "Effective whole-program analysis in the presence of pointers," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6, Lake Buena Vista, Florida, USA: Association for Computing Machinery, 1998, pp. 46–55, ISBN: 1581131089. DOI: 10.1145/288195.288217. [Online]. Available: https://doi.org/10.1145/288195.288217.

[10] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008. DOI: 10.1109/MS.2008.130. [Online]. Available: https://doi.org/10.1109/MS.2008.130.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, A. L. Hosking, P. T. Eugster, and C. V. Lopes, Eds., 2013, pp. 641–660. DOI: 10.1145/2509136.2509549. [Online]. Available: https://doi.org/10.1145/2509136.2509549.

[12] D. F. Bacon and S. L. Graham, "Fast and effective optimization of statically typed object-oriented languages," AAI9828589, Ph.D. dissertation, 1997, ISBN: 059181143X.

[13] A. Bangash, D. Tiganov, K. Ali, and A. Hindle, "Energy efficient guidelines for ios core location framework," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2021, pp. 320–331. DOI: 10.1109/ICSME52107.2021.00035. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSME52107.2021.00035.

[14] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production ios mobile applications," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 363–377. DOI: 10.1109/CGO51591.2021.9370306.

[15] A. Chatzikonstantinou, C. Ntantogian, G. Karopoulos, and C. Xenakis, "Evaluation of cryptography usage in android applications," in *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, ser. BICT'15, New York City, United States: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 83–90, ISBN: 9781631901003. DOI: 10.4108/eai.3-12-2015.2262471. [Online]. Available: https://doi.org/10.4108/eai.3-12-2015.2262471.

[16] Checkmarx, *Checkmarx sast: Scan with ease at the source code level*, https://checkmarx.com/product/cxsast-source-code-scanning/, Accessed: Mar. 9, 2023.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, ISSN: 0164-0925. DOI: 10.1145/115372.115320. [Online]. Available: https://doi.org/10.1145/115372.115320.

[18] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ser. ECOOP '95, Berlin, Heidelberg: Springer-Verlag, 1995, pp. 77–101, ISBN: 3540601600.

[19] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "Iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 44–56, ISBN: 9781450338325. DOI: 10.1145/2810103.2813675. [Online]. Available: https://doi.org/10.1145/2810103.2813675.

[20] dkhamsing, *Open-source ios apps*, https://github.com/dkhamsing/open-source-ios-apps, Accessed: Mar. 9, 2023.

[21] L. N. Q. Do, J. R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, 2022. DOI: 10.1109/TSE.2020.3004525.

[22] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 73–84, ISBN: 9781450324779. DOI: 10.1145/2508859.2516693. [Online]. Available: https://doi.org/10.1145/2508859.2516693.

[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, The Internet Society, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/pios-detecting-privacy-leaks-ios-applications-paper.

[24] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, Orlando, Florida, USA: Association for Computing Machinery, 1994, pp. 242–256, ISBN: 089791662X. DOI: 10.1145/178243.178264. [Online]. Available: https://doi.org/10.1145/178243.178264.

[25] J. Feichtner, D. Missmann, and R. Spreitzer, "Automated binary analysis on ios: A case study on cryptographic misuse in ios applications," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '18, Stockholm, Sweden: Association for Computing Machinery, 2018, pp. 236–247, ISBN: 9781450357319. DOI: 10.1145/3212480.3212487. [Online]. Available: https://doi.org/10.1145/3212480.3212487.

[26] GitHub, *Codeql*, https://codeql.github.com, Accessed: Mar. 9, 2023.

[27] T. S. Group, *Soot - a java optimization framework*, Montréal, QC, Canada: McGill University, 1999. [Online]. Available: https://github.com/Sable/soot.

[28] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *International Symposium on Software Testing and Analysis (ISSTA)*, M. Young and T. Xie, Eds., 2015, pp. 106–117. DOI: 10.1145/2771783.2771803. [Online]. Available: https://doi.org/10.1145/2771783.2771803.

[29] B. Kaliski. (Sep. 2000). "Pkcs #5: Password-based cryptography specification version 2.0," [Online]. Available: https://www.rfc-editor.org/rfc/rfc2898.

[30] kingthorin, *Sql injection*, https://owasp.org/www-community/attacks/SQL_Injection, Accessed: Mar. 9, 2023.

[31] A. Krizmanic and A. Jess, *Is objective-c still relevant in 2023 or is swift the only real choice?* https://www.itmagination.com/blog/is-objective-c-still-relevant-in-2022-or-is-swift-the-only-real-choice, Accessed: Apr. 4, 2023.

[32] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "Cognicrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017, Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 931–936, ISBN: 9781538626849.

[33] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2021. DOI: 10.1109/TSE.2019.2948910.

[34] M. Krzyzanowski, *Cryptoswift*, 2022. [Online]. Available: https://github.com/krzyzanowskim/CryptoSwift.

[35] C. Lattner, A. Lenharth, and V. Adve, "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, Jun. 2007.

[36] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail? a case study and open problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14, Beijing, China: Association for Computing Machinery, 2014, ISBN: 9781450330244. DOI: 10.1145/2637166.2637237. [Online]. Available: https://doi.org/10.1145/2637166.2637237.

[37] Y. Li, Y. Zhang, J. Li, and D. Gu, "Icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications," in *Network and System Security*, M. H. Au, B. Carminati, and C.-C. J. Kuo, Eds., Cham: Springer International Publishing, 2014, pp. 349–362, ISBN: 978-3-319-11698-3.

[38] LLVM Developer Group, *Clang: A c language family frontend for llvm*, https://clang.llvm.org/, Accessed: Mar. 9, 2023.

[39] ——, *The llvm compiler infrastructure*, Accessed: Mar. 9, 2023. [Online]. Available: https://llvm.org/.

[40] MicroFocus, *Fortify static code analyzer*, https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer, Accessed: Mar. 9, 2023.

[41] A. Milanova, A. Rountev, and B. G. Ryder, "Precise call graphs for c programs with function pointers," English, *Automated Software Engineering*, vol. 11, no. 1, pp. 7–26, 2004. DOI: 10.1023/B:AUSE.0000008666.56394.a1.

[42] S. Rasthofer, S. Arzt, R. Hahn, and M. Kolhagen, *(in)security of backend-as-a-service*, 2015.

[43] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61, ISBN: 0897916921. DOI: 10.1145/199448.199462. [Online]. Available: https://doi.org/10.1145/199448.199462.

[44] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for C/C++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019, pp. 393–410. DOI: 10.1007/978-3-030-17465-1_22. [Online]. Available: https://doi.org/10.1007/978-3-030-17465-1%5C_22.

[45] ——, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds., Cham: Springer International Publishing, 2019, pp. 393–410, ISBN: 978-3-030-17465-1.

[46] SonarSource, *Swift - clean code for your swift projects*, https://www.sonarsource.com/swift/, Accessed: Mar. 9, 2023.

[47] J. Späth, *Spds*, https://github.com/codeshield-security/spds, Accessed: Mar. 9, 2023, CodeShield GmbH.

[48] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290361. [Online]. Available: https://doi.org/10.1145/3290361.

[49] StatCounter GlobalStats, *Desktop operating system market share worldwide*, https://gs.statcounter.com/os-market-share/desktop/worldwide/2022, Accessed: Apr. 4, 2023.

[50] ——, *Mobile operating system market share worldwide*, https://gs.statcounter.com/os-market-share/mobile/worldwide/2022, Accessed: Apr. 4, 2023.

[51] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 264–280, ISBN: 158113200X. DOI: 10.1145/353171.353189. [Online]. Available: https://doi.org/10.1145/353171.353189.

[52] SwiftLint Community, *Swiftlint - a tool to enforce swift style and conventions*, https://github.com/realm/SwiftLint, Accessed: Mar. 9, 2023.

[53] Synopsys, *Coverity static application security testing*, https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html, Accessed: Mar. 9, 2023.

[54] Tailor Community, *Tailor - cross-platform static analyzer and linter for swift*, https://github.com/sleekbyte/tailor, Accessed: Mar. 9, 2023.

[55] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications.," *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[56] T. Team, *Type analyzer for javascript*, Århus, Denmark: Århus University, 2009. [Online]. Available: https://github.com/cs-au-dk/TAJS.

[57] TensorFlow, *Swift for tensorflow (archived)*, https://github.com/tensorflow/swift, Accessed: Mar. 9, 2023.

[58] ——, *Swift for tensorflow (in archive mode)*, https://www.tensorflow.org/swift/guide/overview, Accessed: Mar. 9, 2023.

[60] D. Tiganov, L. N. Q. Do, and K. Ali, "Designing uis for static-analysis tools," *Commun. ACM*, vol. 65, no. 2, pp. 52–58, Jan. 2022, ISSN: 0001-0782. DOI: 10.1145/3486600. [Online]. Available: https://doi.org/10.1145/3486600.

[61] S. Turner, *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*, RFC 6151, Mar. 2011. DOI: 10.17487/RFC6151. [Online]. Available: https://www.rfc-editor.org/info/rfc6151.

[62] Uber, *Ribs: Uber's cross-platform mobile architecture framework.* https://github.com/uber/RIBs, Accessed: Mar. 9, 2023.

[63] WALA Community, *T.j. watson libraries for analysis (wala)*, https://github.com/wala/WALA/, Accessed: Mar. 9, 2023.

[64] XcodeGen Community, *Xcodegen*, https://github.com/yonaskolb/XcodeGen, Accessed: Mar. 9, 2023.

[65] M. Zheng, H. Xue, Y. Zhang, T. Wei, and J. C. Lui, "Enpublic apps: Security threats using ios enterprise and developer certificates," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, Singapore, Republic of Singapore: Association for Computing Machinery, 2015, pp. 463–474, ISBN: 9781450332453. DOI: 10.1145/2714576.2714593. [Online]. Available: https://doi.org/10.1145/2714576.2714593.