SWAN: A Static Analysis Framework for Swift

Daniil Tiganov
University of Alberta
tiganov@ualberta.ca

Jeff Cho
University of Alberta
jeff.cho@ualberta.ca

Karim Ali
University of Alberta
karim.ali@ualberta.ca

Julian Dolby
IBM Research
dolby@us.ibm.com

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. While LLVM and Clang support some analyses for Swift, they are either primarily dynamic analyses or 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 paper, we present SWAN, an open-source framework that allows robust program analyses of Swift programs using IBM's T.J. Watson Libraries for Analysis (WALA). To provide a wide range of analyses for Swift, SWAN leverages the well-established libraries in WALA. SWAN is publicly available at https://github.com/themaplelab/swan. We have also made a screencast available at https://youtu.be/AZwfhOGqwFs.

CCS CONCEPTS

• **Theory of computation → Program analysis**.

KEYWORDS

Swift, static analysis, taint analysis

1 INTRODUCTION

Static analysis reasons about the potential runtime behaviour of a program without necessarily executing it. Using this technique may help protect user privacy [1] and optimize applications [2]. Despite the potential benefits of static analysis, there is a lack of available tools for Swift [4], Apple's recommended choice for development on iOS [15] and macOS [17]. In 2019, the web traffic analysis tool StatCounter estimated that iOS devices comprised approximately 24.79% of mobile devices in the world [10] and macOS devices accounted for 16.46% of desktop devices [9]. Trends also show that the popularity of both operating systems in 2020 has increased by 4.41% and 3.82%, respectively. Therefore, the ability to analyze Swift apps has significant impact on millions of users around the world.

To bridge the gap between the increasing popularity of Swift and the lack of available analysis tools, we introduce SWAN, an open-source static analysis framework for Swift. We designed SWAN with the app developer as our main target audience. Therefore, SWAN offers both a Command-Line Interface (CLI) and a Graphical User Interface (GUI). Both interfaces offer the same functionalities but address different use cases. For example, the CLI enables the developer to integrate SWAN in their continuous integration workflow, providing analysis results at major development milestones. On the other hand, the GUI enables SWAN to provide the developer with up-to-date analysis results whenever they modify their code. This immediate feedback helps developers focus on the task at hand, which further helps them fix more bugs in less time [7].

While designing SWAN, we have also taken into consideration future contributions to its underlying analysis engine. To enable contributions from the wider static analysis community, we opted for building SWAN on top of the well-established IBM T.J. Watson Libraries for Analysis (WALA) [6]. We re-used various analysis components that WALA has built over the years, and has proven to work well for analyzing different programming languages such as Java, JavaScript, and Python. This design decision enables app developers to use the various analyses that SWAN offers out of the box (e.g., taint analysis, pointer analysis, call graph construction, and inter-procedural dataflow analysis) without having to implement their own analysis. Moreover, SWAN has a modular architecture that enables researchers to build their own analyses on top of it by leveraging its existing analysis infrastructure.

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 [1], Java [13], and JavaScript [23].

2 HOW CAN APP DEVELOPERS USE SWAN?

App developers may use SWAN through either one of its frontends: a command-line interface and a VSCode [24] extension. We will demonstrate the features of both frontends through the built-in taint analysis of SWAN, which tracks data leaks in a given program.

2.1 Command-Line Interface

SWAN provides a CLI script called `run-swan-single`. This script analyzes a single Swift file[1]. The user may define sources of private information (i.e., sources), potential locations where data may leak to (i.e., sinks), and methods that properly secure private information (i.e., sanitizers) in the source file as code comments. Figure 1 represents a sample Swift program that exhibits a tainted dataflow. The script runs on the input file that has a source and a sink defined as code comments. When SWAN finishes its analysis, it prints the results to the terminal. Figure 2 shows how SWAN formats the results in a tree structure, where each path consists of a source, sink, and path edges (i.e., intermediates).

2.2 VSCode Extension

SWAN provides a GUI via a custom VSCode extension for analyzing Swift programs and viewing its results. To use the extension, the user must first configure SWAN under *Settings→SWAN*. Figure 3

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

[1] We are developing a frontend to support Xcode projects. In the past, our frontend has successfully analyzed Xcode projects. However, due to recent changes to Xcode, Swift, and macOS versioning, our Xcode frontend is no longer functional.

```
// SWAN:sources: "source() -> Swift.String"
// SWAN:sinks: "sink(sunk: Swift.String) -> ()"
func source() -> String { return "I'm bad"; }
func sink(sunk: String) { print(sunk); }
func random() -> String { return "whatever"; }
let whatever = random();
let src = source();
let combined = whatever + src;
sink(sunk: combined);
```

Figure 1: A Swift program with tainted dataflow.

```
$./utils/run-swan-single -sdk $SDK_PATH -path ./ca.
      maple.swan.swift.test/tests/taint-single-source/
      StringConcat.swift
[...]
========= RESULTS =========
-- PATH
  -- SOURCE
     7:11 in [...]/StringConcat.swift
     let src = source();
                ^
  -- INTERMEDIATE
     8:16 in [...]/StringConcat.swift
     let combined = whatever + src;
                     ^
  -- SINK
     9:1 in [...]/StringConcat.swift
     sink(sunk: combined);
     ^
====== END OF RESULTS ======
```

Figure 2: An example illustrating how to use the SWAN CLI frontend to analyze a single Swift file.

```
{
"swan.SDKPath": "/Applications/Xcode.app/Contents/
      Developer/Platforms/MacOSX.platform/Developer/
      SDKs/MacOSX.sdk/",
"swan.CustomSSS": {
  "swan.Sources": ["source() -> Swift.String"],
  "swan.Sinks": ["sink(sunk: Swift.String) -> ()"],
  "swan.Sanitizers": []
},
"swan.SingleFilePath": "/<user>/Documents/swan/ca.
      maple.swan.swift.test/tests/taint-single-source/
      StringConcat.swift",
"swan.TaintAnalysisMode": "Refined"
}
```

Figure 3: An example illustrating the contents of the configuration file (`settings.json`) for the SWAN VSCode extension.

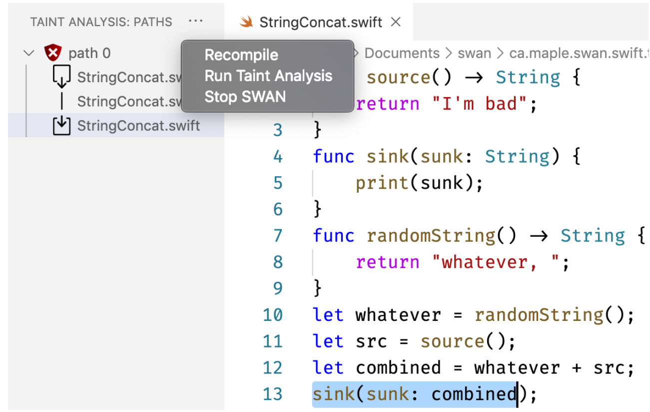

Figure 4: The main GUI of the SWAN VSCode extension.

shows an example configuration for the SWAN `settings.json` file. Figure 4 shows the main GUI elements of the extension. To easily edit the configuration file, SWAN provides function name autocompletion. If the configuration file changes, the user may still quickly re-run the client analysis, without recompiling the code. To recompile the Swift program, the user must press *Recompile*.

After configuring the extension, the user may start SWAN by selecting the SWAN tab and pressing *Run Taint Analysis* in the sidebar. The extension then automatically attaches to an existing SWAN Java Virtual Machine (JVM), if one is running, or starts a new JVM if one is not already running. We recommend to first launch the JVM separately to view the console output, especially for debugging purposes. To enable bidirectional communication with the running SWAN JVM, the extension uses sockets.

For our example taint analysis, SWAN displays the results in the sidebar in filetree-like form. Each vulnerable path is an element in the tree with a red cross beside it. Its children are the nodes in the path. The first child is the source, the last is the sink, and any nodes in between are intermediates. The user may select any path node, and the extension will open the file at the corresponding source location. In Figure 4, the user has selected the last node. Therefore, SWAN highlights Line 13 in the source file, showing the user that tainted data reaches the function `sink()` as a parameter.

3 THE MAIN WORKFLOW OF SWAN

SWAN has a linear workflow where each component produces the data requested by its parent component. Figure 5 represents the workflow components and numerically labels them according to their execution order. At the beginning of the workflow 1, a SWAN frontend instantiates a JVM using its corresponding *SWAN Driver* 2 with build and analysis options. The user may configure these options in the frontend. The build options contain arguments needed to call the Swift compiler properly. Therefore, SWAN propagates them through its components all the way to *SWAN Hook* 6, where SWAN eventually calls the Swift compiler.

To represent the program dataflow, SWAN uses the System Dependence Graph (SDG), an internal WALA data structure. To generate the SDG, *SWAN Driver* first calls the frontend-agnostic *WALA Driver* 3, which then calls *WALA Analysis Engine* 4 to construct

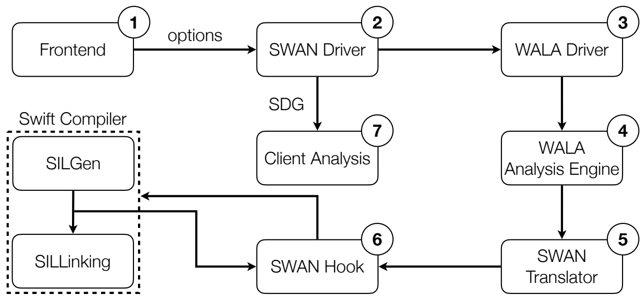

Figure 5: The main workflow of SWAN.

a call graph (CG) for the input program. Using WALA libraries, *WALA Driver* then generates the SDG from that CG and returns it back to *SWAN Driver*. Besides storing call information, the CG data structure stores the class hierarchy and the Intermediate Representation (IR) of the input program. To construct the CG, *WALA Analysis Engine* first translates the input Swift program to WALA IR, and then generates the CG from that IR. Internally, WALA uses the Common Abstract Syntax Tree (CAst) as an IR between the input language and WALA IR itself. To translate the input Swift code to WALA CAst, the *WALA Analysis Engine* uses *SWAN Translator* ⑤.

After extensive discussions with the Apple Swift team, we determined that it is best for SWAN to consume Swift Intermediate Language (SIL), an internal IR within the Swift compiler pipeline, as input instead of the Swift Abstract Syntax Tree (AST). To maintain fidelity to the original source code, SWAN operates on *raw* SIL before the Swift compiler applies any code transformations or optimizations. Through a Java Native Interface (JNI) call, *SWAN Hook* ⑥ invokes the Swift compiler and provides it with a callback handler that receives the SIL during compilation. The callback handler uses a custom visitor to process all compilation contexts: `SILModule`, `SILFunction`, `SILBlock`, and `SILInstruction`. This visitor packages all information needed for translating it into JNI `jobjects` (i.e., JVM objects in C++ form). To translate the code to WALA CAst, *SWAN Hook* sends back the packaged information to *SWAN Translator* using JNI.

Once SWAN translates the input Swift code to WALA CAst, SWAN may run a client analysis (e.g., taint analysis) on the program, represented in SDG form ⑦. SWAN then returns the results to the frontend to be displayed in the appropriate format.

4 TRANSLATING SWIFT SOURCE CODE

SWAN leverages the existing infrastructure in WALA to translate the SIL representation of an input Swift program into WALA IR. To achieve that, SWAN first translates the input SIL to CAst, which requires an intermediate level of abstraction due to the significant differences between the two IRs. In particular, SIL uses pointers whereas CAst uses references. To serve as this abstraction, we have developed SWAN IR as a simple, hybrid IR between SIL and WALA IR. Table 1 lists the main SWAN IR instructions, which currently support all but two SIL instructions: `partial_apply` and `assign_by_wrapper`. Similar to SIL, SWAN IR is only composed of functions, basic blocks, and instructions.

Table 1: A list of the main SWAN IR instructions.

Instruction	Notation
`new`	v0 := new $String
`assign`	v0 := v1
`literal`	v0 := #foo, v0 := #12
`goto`	goto bb0
`conditional goto`	cond_br v0 true: bb1, false: bb2
`throw`	throw v0
`conditional throw`	throw if v0
`return`	return v0
`field read`	v0 := v1.foo
`field write`	v0.foo := v1
`function ref`	v0 := func_ref test.foo() ->Swift.Int
`builtin`	v0 := builtin Swift.Int.init[...]
`apply`	v0 := v1(v2, v3)
`try apply`	try v0(v1) normal: bb1, error: bb2
`binary op`	v0 := v1 + v2
`unary op`	v0 := ! v1

4.1 Field Aliases

SWAN Translator converts a SIL pointer to an object with the field `value` representing its underlying value. This conversion is nontrivial. For example, the SIL instruction `ref_element_addr` derives the address of a class field, and writes it to a value that may be accessed later. Therefore, treating the instruction as a regular field read is not sufficient. To address this problem, the SWAN IR symbol table has a special type called *Field Alias* that aliases a field access path. To handle instructions such as `ref_element_addr`, SWAN uses this type to lazily defer the field access until the field value is accessed. Figure 6 illustrates a SIL example where `ref_element_addr` derives the address of the class field `A.foo` (Line 45) and stores it to `%2`. This class field is later read by the local variable `%4` (Line 47) through the alias `%3` (Line 46). To translate this example correctly, SWAN uses the *Field Alias* type.

4.2 SIL Coroutines and Instructions

SWAN Translator simplifies complex components of SIL. In particular, SWAN handles asymmetric coroutines by inlining them into their caller. Moreover, SWAN ignores instructions that we have determined do not mutate or move data in a manner that is relevant to dataflow analysis such as low-level memory management instructions.

4.3 SIL Builtin Functions

SIL handles many operations through builtin functions. For instance, the three data structures that Swift provides (i.e., `Array`, `Set`, and `Dictionary`) are entirely accessed via builtin function calls. Simple tasks such as literal manipulation and string operations are also handled using builtins. To support builtin functions in SWAN, we have developed a SWAN IR parser that reads handwritten summaries. Currently, SWAN supports numerous builtins such as literal and array operations. To fully support the main Swift data structures, we are continuously adding support for more builtins, updating the SWAN IR instructions whenever necessary.

```
---- Swift ----
class A {
  var foo = "bar";
}
---- SIL of A.foo.getter ----
sil hidden [transparent] [ossa] @$s4temp1AC3fooSSvg : $@convention(method) (@guaranteed A) -> @owned String {
bb0(%0 : @guaranteed $A):
  debug_value %0 : $A, let, name "self", argno 1
  %2 = ref_element_addr %0 : $A, #A.foo
  %3 = begin_access [read] [dynamic] %2 : $*String // copy %2 to %3
  %4 = load [copy] %3 : $*String // realize alias %3 as a field read to %4
  end_access %3 : $*String
  return %4 : $String
}
---- SWANIR of A.foo.getter ----
func $String `temp.A.foo.getter : Swift.String`(v0 : $A) {
  bb0(v0 : $A) :
    v1 := v0.foo
    return v1
}
```

Figure 6: An example illustrating how `ref_element_addr` may cause field aliasing.

To enable translation to other languages, we have designed SWAN to provide a modular workflow. Other framework designers may use the SWAN IR visitor to easily translate other input languages to WALA IR.

5 RELATED WORK

5.1 Android Analysis Frameworks

The Android platform has seen an abundance of analysis frameworks over the past decade. FlowDroid [1] is a lifecycle-aware and context-sensitive, flow-sensitive, field-sensitive, and object-sensitive taint analysis tool for Android apps. The design of FlowDroid inspired us in the design of SWAN. Similar to SWAN, SCanDroid [3] also uses WALA, but for the purpose of matching Android app manifests to dataflow analyses to ensure that apps do not overreach their permissions. DroidInfer [14] 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 the first open-source static analysis framework for Swift.

5.2 LLVM-Based Analyses

While LLVM [11] and Clang [12] 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 information are typically lost during the compilation of Swift source code to low-level LLVM IR. 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.

The Phasar framework [19] provides call graph construction and dataflow analyses on LLVM IR, enabling it to analyze Swift applications. However, similar to other LLVM-based frameworks, Phaser focuses more on low-level LLVM constructs, whereas SWAN analyzes the SIL representation [5], preserving important information from the Swift source code such as the file and line number of the corresponding originating code, which is important for notifying developers of the locations of identified issues.

5.3 Swift Analysis Tools

Most publicly available analysis tools for Swift are linters such as SwiftLint [18] and Tailor [20]. Those tools only help enforce Swift code standards and best practices.

SonarSwift [22] is a static Swift code analyzer which allows users to define rules for bugs, code smells, and vulnerabilities to find in their codebase. Some existing rules listed on the website [21] include not using identical expressions on both sides of a binary operator (i.e., bug), not duplicating string literals (i.e., code smell), and avoiding using DES (i.e., vulnerability). However, at the time of this writing, 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.

6 CONCLUSION

We presented SWAN, a static analysis framework for Swift that we built on top of the WALA analysis framework. SWAN provides various analyses to its users including call graph analysis, pointer analysis, and inter-procedural dataflow analysis. To enable wider use of SWAN, we have developed two user interfaces for it: a command-line interface and a VSCode extension. We have also designed SWAN to be a modular framework where some of its components (e.g., SWAN IR and SDG-based dataflow analysis) may be used for other WALA-based analysis frameworks. In the future, we plan to build more support for analyzing iOS apps in SWAN, which will put it on par with its Android counterparts with respect to analyzing mobile applications. SWAN is open source [16], and we welcome contributions under the Eclipse Public License 2.0 [8].

ACKNOWLEDGMENTS

We would like to thank all the following contributors to the SWAN project: Noah Weninger, Leo Li, Mark Mroz, Yaser Alkayale, Lydia Wu, Chen Song, Bryan Tam, and Anthony Hill. We would also like to thank Dillon Pratt for his help with video-editing our screencast. This material is based upon work supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*. 259–269. https://doi.org/10.1145/2594291.2594299

[2] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

[3] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). 641–660. https://doi.org/10.1145/2509136.2509549

[4] Swift Community. 2015. *The Swift Programming Language*. Retrieved Jan 24, 2020 from https://swift.org/

[5] Swift Community. 2020. *Swift Intermediate Language (SIL)*. Retrieved May 28, 2020 from https://github.com/apple/swift/blob/master/docs/SIL.rst

[6] WALA Community. 2006. *T.J. Watson Libraries for Analysis*. Retrieved May 28, 2020 from https://github.com/wala/WALA/

[7] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. 2017. Just-in-time Static Analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*. 307–317. https://doi.org/10.1145/3092703.3092705

[8] Eclipse Foundation. 2017. *Eclipse Public License 2.0*. Retrieved May 28, 2020 from https://www.eclipse.org/legal/epl-2.0/

[9] StatCounter GlobalStats. 2019. *Desktop Operating System Market Share Worldwide*. Retrieved Mar 17, 2020 from https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201901-201912

[10] StatCounter GlobalStats. 2019. *Mobile Operating System Market Share Worldwide*. Retrieved Mar 17, 2020 from https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201901-201912

[11] LLVM Developer Group. 2003. *The LLVM Compiler Infrastructure*. Retrieved May 28, 2020 from https://llvm.org/

[12] LLVM Developer Group. 2007. *Clang: a C language family frontend for LLVM*. Retrieved May 28, 2020 from https://clang.llvm.org/

[13] The Sable Group. 1999. *Soot - A Java optimization framework*. McGill University, Montréal, QC, Canada. Retrieved June 4, 2020 from https://github.com/Sable/soot

[14] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for Android. In *International Symposium on Software Testing and Analysis (ISSTA)*, Michal Young and Tao Xie (Eds.). 106–117. https://doi.org/10.1145/2771783.2771803

[15] iOS Team. 2007. *iOS 13 - Apple (CA)*. Apple, Cupertino, California, USA. Retrieved Jan 24, 2020 from https://www.apple.com/ca/ios/

[16] The Maple Lab. 2017. *A Swift Static Analysis Framework*. University of Alberta, Edmonton, Alberta, Canada. Retrieved May 28, 2020 from https://github.com/themaplelab/swan

[17] macOS Team. 2001. *macOS Mojave - Apple (CA)*. Apple, Cupertino, California, USA. Retrieved Jan 24, 2020 from https://www.apple.com/ca/macos/mojave/

[18] Realm. 2015. *SwiftLint - A tool to enforce Swift style and conventions*. Retrieved May 28, 2020 from https://github.com/realm/SwiftLint

[19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. 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)*. 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

[20] Sleekbyte. 2015. *Tailor - Cross-platform static analyzer and linter for Swift*. Retrieved May 28, 2020 from https://tailor.sh/

[21] SonarSource. 2017. *Swift Static Code Analysis Rules*. Retrieved May 28, 2020 from https://rules.sonarsource.com/swift

[22] SonarSwift. 2015. *Code Quality and Security for Swift*. SonarSource, Geneva, Switzerland. Retrieved May 28, 2020 from https://www.sonarsource.com/swift/

[23] TAJS Team. 2009. *Type Analyzer for JavaScript*. Århus University, Århus, Denmark. Retrieved June 4, 2020 from https://github.com/cs-au-dk/TAJS

[24] Visual Studio Team. 2015. *Visual Studio Code - Code Editing. Redefined*. Microsoft, Redmond, Washington, USA. Retrieved Feb 19, 2020 from https://code.visualstudio.com

A SCREENCAST WALKTHROUGH

This section is a walkthrough of the SWAN screencast available at https://youtu.be/AZwfhOGqwFs.

A.1 Introduction 0:00-0:35

- SWAN is a static analysis framework for analyzing Swift applications for security vulnerabilities using taint analysis.
- SWAN uses IBM T.J. Watson Libraries for Analysis (WALA) as its analysis engine.
- SWAN has two frontends: a command line interface (CLI) script called `run-swan-single`, and a VSCode extension.
- The CLI script will be demonstrated first.

A.2 Test Case 0:35-2:12

- Before the CLI script is demonstrated, we need an interesting test case to run SWAN on.
- This test case demonstrates SWAN has field, context, and object sensitivity as well as support for function objects and taintedness preservation through string concatenation. This test case was meant to be interesting, but is not exhaustive of SWAN's analysis capabilities.
- Sources and sinks must be defined directly inside of the test case with a comment, but only for the CLI frontend.
- This portion of the video describes the components of the test case:
  - Generic source and sink function.
  - Class `A` with field `g` of type `B` (`g` is instantiated when `A` is instantiated).
  - Class `B` with fields `f` and `v`, both of type `String`. `f` and `v` are initialized to an empty string (not tainted).
  - Function `sinkFunc` returns the sink function. This show's that SWAN supports function objects.
  - Function `foo` takes in an object of type `B` and sets its `f` field to the given string, concatenated with a non-tainted string. This show's that SWAN preserves taintedness through string concatenation. Moreover, SWAN supports most Swift operators and even operator overloading (not demonstrated in the screencast).
  - Lastly, the code that creates the dangerous paths is described.
    - Object of type `A` is created and assigned to variable `a`.
    - Variable `b` is aliased to `a.g`.
    - Object of type `B` is created and assigned to variable `b2`. This object is created to later demonstrate that SWAN has object sensitivity.
    - A tainted string is created and written to `b.f` using the function `foo`.
    - A function object, `f`, representing the sink function is created using `sinkFunc`.
    - `b.f` is given to `f`. This is where we should detect a tainted variable being sunk (a security vulnerability).
    - To test and demonstrate field sensitivity, we also sink the other field of `b`, `b.v`. We should not detect a vulnerability here because `b.v` was never tainted. Field sensitivity means that we can differentiate between multiple fields, especially of the same type, of an object.
    - Lastly, `b2.f` is sunk to show we have object sensitivity. That is, SWAN can differentiate between multiple objects of the same type in the heap.

A.3 CLI Script 2:12-2:45

- The `run-swan-single` script is demonstrated here.

- `run-swan-single` requires a path to the macOS SDK (available inside of `Xcode.app`), and a path to the test file.
- The script quickly analyzes the file and prints the results to the terminal.
- The results consist of a single dangerous path represented by nodes corresponding to code segments. The path begins with the tainted variable allocation site and ends with the tainted variable being sunk on the call to the function object, `f`. There also exist three intermediate nodes. The first two intermediate nodes are the call to `foo` (where the tainted variable is written to `b.f`). There are two nodes here due to the nature of WALA's System Dependency Graph (SDG) node structure, where every call has a caller and callee node. The third intermediate node is the string concatenation operation.
- These results are consistent with what we expected. We did not detect dangerous paths for the other two sinks.

A.4 VSCode Extension 2:45-4:10

- SWAN's VSCode extension first needs to be configured with an SDK path and path to the test file, just like with the CLI script. In the demonstration, these settings are already configured.
- Sources and sinks can be configured in the `settings.json` file.
- It isn't always obvious what a function's signature is because it differs from the source code function signature. Therefore, SWAN provides an autocompletion feature inside of `settings.json`, but is only available after running first time analysis.
- We navigate to the SWAN tab and use the menu in the sidebar to run first time analysis. When running SWAN on a file for the first time, the extension will check if there is a SWAN analysis server (JVM) already running. If it does not find one, like in this case, it will start a new SWAN analysis server. SWAN has a script for launching the analysis server separately, but this feature is mainly for debugging and is outside of the scope of this demonstration.
- Now we can utilize autocompletion to populate the sources and sinks inside of `settings.json`.
- We rerun the analysis. This time we do not compile the source, and, therefore, the analysis is almost instant.
- We now see that SWAN has detected a dangerous path because the sidebar has been populated with a tree-like path. These nodes are the same nodes we saw in the CLI script output. Clicking on a node will open up the source file and highlight the corresponding code segment.
- To demonstrate how quickly SWAN can recompile and reanalyze the test case, we modify the test case by sinking the tainted variable again, and then we rerun the analysis.
- Lastly, we stop the analysis server.

A.5 Ending 4:10-4:18

SWAN is available at https://github.com/themaplelab/swan.