Heros in Action: Analyzing Objective-C Binaries through Decompilation and IFDS

Florian Magin[†]
Fraunhofer SIT | ATHENE
Germany
florian.magin@sit.fraunhofer.de

Gwendal Patat[†]
Fraunhofer SIT | ATHENE
Germany
gwendal.patat@sit.fraunhofer.de

Fabian Scherf[†]
Fraunhofer SIT | ATHENE
Germany
fabian.william.scherf@sit.fraunhofer.de

***Abstract*—This paper demonstrates static taint analysis on Objective-C binaries through the integration of the Ghidra framework with the Heros IFDS solver, achieving an inter-procedural, field-sensitive and flow-sensitive analysis. Our contributions include two plugins: one extending Ghidra Objective-C capabilities to improve decompilation accuracy, and another integrating the Heros framework for inter-procedural taint analysis on Ghidra Intermediate Representation (IR).**

To assess our approach, we introduce a new benchmark suite tailored to Objective-C, covering diverse dataflow challenges and promoting further community-driven research. By leveraging existing frameworks, this work demonstrates how established static analysis techniques can be adapted to binary targets, laying a groundwork for advancements in Objective-C binary analysis.

***Index Terms*—Heros, IFDS, Objective-C, Decompilation**

I. Introduction

Although extensive research has been conducted on data-flow analysis for mobile applications, Objective-C and iOS applications have received limited attention in contrast to Java and Android ones. This disparity can be attributed to the dominance of compiled languages, such as Objective-C and Swift, in the iOS ecosystem [1], which introduces unique challenges related to binary analysis [2]. These challenges are further amplified by the closed nature of the iOS environment, creating obstacles such as requirements for proprietary Mac hardware, and jailbroken devices for in-depth app analysis [3].

The current research landscape in binary analysis and data-flow analysis suggests limited overlap between these fields, despite both being critical for the analysis of iOS applications. Binary analysis has often focused on Internet of Things (IoT) firmware images, primarily written in C. This poses distinct analytical challenges, such as reconstructing memory structure layouts, and type information, which are themselves active areas of research [4], [5]. Additionally, the absence of Object-Oriented Programming (OOP) constructs in C removes many complexities associated with it, e.g., dynamic dispatch. Regarding data-flow analysis, in the Android/Java ecosystem, researchers benefit from a rich set of tools and research advancements, primarily due to the open nature of the platform and the high-level abstractions inherent to Java. Frameworks and tools, such as Soot [6], FlowDroid [7] or DroidSafe [8], facilitate analysis by leveraging the statically typed nature of Java structure, and its use of bytecode, inexistent in compiled Objective-C. These tools have allowed the detection of various security vulnerabilities and privacy leaks in Android applications [9]. In comparison, the closed nature of Objective-C binaries introduce new complexities that highlight the need for Objective-C specific analysis frameworks.

Objective-C binaries thus present somewhat of a middle ground between these two domains: Binaries contain sufficient metadata to reconstruct the layouts of objects in memory, and thus circumvent a significant challenge of other native languages, such as C, while introducing OOP concerns. However, due to the tight coupling between apps and runtime environment, restrictions are imposed on the decompiler ability to recover a high-level representation of the application, particularly if the runtime components are not carefully addressed.

Another significant challenge for applying static analysis techniques on Objective-C is the absence of established benchmarks that allow for objective and incremental comparisons. Benchmarks enable step-by-step validation of different approaches, providing controlled modular test cases. While previous research has attempted to analyze dataflow in Objective-C and iOS applications on real-world datasets [10], the iOS ecosystem has since evolved with dynamic linking and the removal of identifiers used in evaluations. In comparison, Java and Android apps have benefited from well-established benchmarks, enabling more straightforward and consistent comparative analyses [11], [12].

In this paper, we present a data-flow analysis approach on Objective-C binaries using the Inter-procedural, Finite, Distributive Subset (IFDS) framework, achieved by integrating the Ghidra decompiler [13], with the Heros framework [14] through a custom plugin. To tackle Objective-C decompilation challenges, we developed an extra plugin to improve decompilation accuracy. Various approaches, such as using LLVM, could have been used for our work. Nonetheless, Ghidra being a well-know and established open-source project written in Java, our main objective was to extend its capabilities. Heros being also available as a Java project, coupling the two represented the most practical choice. Additionally, we introduce a benchmark suite for Objective-C taint analysis and evaluate our approach on a relevant subset, discussing limitations and future improvements.

[†] These authors contributed equally to the work presented in this paper.

Our contributions can be summarized as follows:

- We introduce our Heros P-Code plugin for Ghidra, leveraging the IFDS framework to enable data-flow analysis on Objective-C binaries.
- We present a second plugin for Objective-C to address key decompilation challenges, improving decompilation accuracy, and by extension our dataflow tracking.
- We create a benchmark suite tailored to Objective-C dataflow challenges, enabling structured evaluation of static binary analysis tools.
- We release our plugins[1,2] and benchmark suite[3] as open-source resources.

II. Background

In this section, we provide an overview of the IFDS problem, the Objective-C specific challenges to tackle for data-flow analysis, and the Ghidra Intermediate Representation (IR) used in this work.

A. IFDS Framework

The IFDS framework [15] is a foundational approach used for solving inter-procedural, context-sensitive, and flow-sensitive data-flow analysis problems. It reduces these problems to a graph-reachability problem and solves it using the tabulation algorithm.

The program logic is modeled as an Inter-procedural Control Flow Graph (ICFG), consisting of nodes N, representing program statements, and edges E, modeling control flow transitions. Each edge in the ICFG is associated with a flow function $f : \mathcal{P}(D) \to \mathcal{P}(D)$, specifying how dataflow facts in a finite set D propagate between program statements.

For the specific problem of taint analysis, the flow functions describe the taint flow propagation through the program. Of particular importance are typical *identity*, *gen*, *kill*, and *transfer* flow functions. Figure 1 shows example elements $a, b, c \in D$, representing variables that can be tainted or not. The *identity* function represents a propagation between program statements that do not affect the taint flow. The *gen* function taints the variable a, independently of its current state, e.g., `a = source()`, whereas the *kill* function models a sanitization of the variable b. The *transfer* function expresses the transfer of the taint state from a to b, modelling the statement `b = a`.

Heros provides a generic implementation of the IFDS framework, and the mentioned flow functions. In this work, we use Heros as IFDS solver in our plugin.

B. Objective-C Challenges

We present Objective-C specific challenges in the context of binary analysis that are essential for achieving data-flow analysis with IFDS. We present solutions addressing these challenges in Section III-B.

[1]https://github.com/ReverseApple/GhidraApple
[2]https://github.com/Fraunhofer-SIT/ghidra-syfrit
[3]https://github.com/Fraunhofer-SIT/iFlowBench

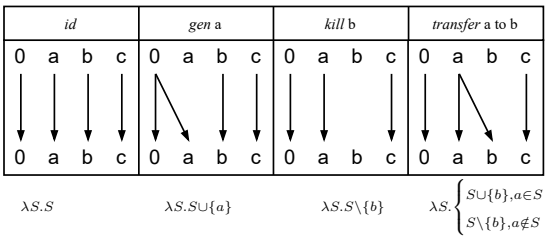

Fig. 1. Graph representation of typical flow functions in IFDS.

Dynamic Dispatch. Objective-C uses dynamic dispatch mechanism and method calls are translated by the compiler into calls to the runtime method `objc_msgSend`. This is similar to virtual dispatches in the JVM bytecode, and brings all its associated problems. Unlike Java, even calls to static methods use this runtime indirection, and there is no separate runtime mechanism for static invokations like `invokestatic`.

Object Layout. Objective-C, like C, ultimately translates field accesses into memory accesses via offsets to the heap pointer returned by the initial object allocation. There is no `getfield` instruction like in the JVM, and all machine instructions that use memory references can be used to access field values. In practice, this problem is more tractable thanks to the compiler translation of all accesses of the form `obj.field` into a dynamic dispatch to the implicitly generated accessor, e.g., `objc_msgSend(obj, "getField")`. However, due to the dynamic nature of Objective-C, these getter methods have no type information available by default, and will only use direct memory accesses internally providing no type or size information either. These default methods can also be overwritten by the developers, and in theory contain arbitrarily complicated logic. It is also possible to use direct references to `field` instead of `self.field` inside any implementation of a method of the object, which results in direct memory accesses again. This makes it not viable to model fields only via their names derived by their accessor method.

Automatic Reference Counting (ARC) Functions. Objective-C uses ARC to track the lifetime of its objects. Many of these functions are relevant to data-flow analysis due to the capability to directly change the state of registers or memory. Simple variants only retain a register value unmodified, e.g., `objc_retain`. while increasing the references.

An example of a more complex ARC function is `ID _objc_getProperty(obj, _cmd, offset, atomic)` which retrieves the field of the given object while also accounting for the potential need for atomic accesses to the field. Without translating such methods into the corresponding field access, field-sensitivity heavily degrades.

Dynamic Typing and limited Function Signatures. Objective-C is a dynamically-typed language in regard to object types. This means that there is only little information available about the type signatures of functions, and the number and locations of their arguments. In contrast, Java methods

retain information about their signature after compilation to bytecode, and the bytecode can be assumed to always treat the objects in accordance with these type constraints due to its statically typed nature.

Additionally, Objective-C brings the argument inference challenges of conventional C, where even the number of arguments is not necessarily known. This gets more complex when accounting for the difference between scalar and float numeric types, which will be stored in different registers.

C. Ghidra Framework

Ghidra is an interactive reverse engineering tool developed and released by the US National Security Agency. It is designed as a standalone tool to facilitate Software Reverse Engineering (SRE). It uses an IR known as P-Code, which abstracts machine-level instructions into a platform-independent language. In Ghidra, this IR enables static analyses, such as dataflow and control flow tracking.

P-Code. The P-Code IR represents individual operations that are applied to virtual registers or memory locations, called `Varnodes`. A `Varnode` is a generalized representation of either a register or memory location, defined by its address space, offset, and size. These `Varnodes` allow P-Code operations to emulate the effects of machine instructions using models for arithmetic, memory access, and control flow. This is similar to Three Address Code (TAC) representations like Jimple IR from the Soot framework.

Ghidra first translates each individual CPU instruction into its corresponding P-Code, reflecting the operation of this single instruction. This so-called *raw P-Code* is then refined and simplified by the decompiler in various stages, where CPU-specific concepts, such as memory address calculations, or flag registers, get translated into higher level concepts. This later stage of P-Code is sometimes called *high-level P-Code*, but we will refer to this process, and its result, as *refined P-Code* in the rest of this paper. This refined P-Code will also be in Static Single Assignment (SSA) form regarding its `Varnode` assignments. During this process, the decompiler re-introduces higher-level programming concepts, such as variables and datatypes. Variables are used as a high-level abstraction for a collection of `Varnodes` that correspond to the same conceptual variable, and packaged in a `HighVariable` object.

Similarly, `HighFunctions` are used to represent an abstracted view of a function, rebuilding high-level signatures and interactions from the machine-level code. `HighFunctions` simplify the analysis of function bodies by grouping together relevant P-Code operations and associating them with higher-level constructs, like local variables using `HighVariables`, parameters, and return values. This enables better representation of how different parts of the binary interact at the function level, which is crucial when performing inter-procedural analyses like our work.

III. Our Objective-C and Heros Plugins

In this section, we present our two custom plugins for Ghidra, developed to enable precise data-flow analysis of Objective-C binaries. The first plugin addresses Objective-C specific challenges by refining P-Code generation, enhancing language-specific decompilation accuracy, and quality of the IR, for subsequent analysis. The second plugin integrates the Heros framework with Ghidra to perform dataflow computations directly on P-Code IR.

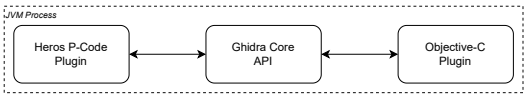

Fig. 2. Interaction between our plugins and the Ghidra ecosystem.

A. Plugins Design and Architecture

Our architecture, as shown in Figure 2, separates language-specific decompilation from core data-flow analysis, leveraging Ghidra plugin extensibility. We developed two modular plugins: the first refines the P-Code for Objective-C by addressing language-specific constructs, thus improving analysis accuracy, while the second integrates the Heros framework to perform inter-procedural data-flow analysis directly on the decompiled P-Code. This modular design enables our Heros plugin to compute dataflow independently of the Objective-C refinements, allowing refined P-Code to benefit both automated and manual analyses. With minimal implementation effort, under 400 lines of Kotlin, our Heros plugin achieves field-sensitive and flow-sensitive analysis, as demonstrated in our benchmark examples in Section IV, and is adaptable to evolving Objective-C analysis needs within Ghidra.

B. Refining P-Code for Objective-C Specific Analysis

Our Heros plugin requires refined P-Code to conduct meaningful data-flow analysis. To achieve this, we build an independent Ghidra plugin for Objective-C, tackling language-specific decompilation problems presented in Section II-B.

Resolving Simple Dynamic Dispatches. Many cases of dynamic dispatches can be resolved without resorting to approximations. The two common categories are calls to static methods and calls to methods where the receiver has been allocated in the callee. We resolve both categories by attempting to backwards slice the receiver parameter based on the SSA form of P-Code, until we reach a memory reference to a static class or an allocation site. We then use these to derive the precise method called at the dynamic dispatch site, and add a reference to the Ghidra Control Flow Graph (CFG). Because these calls have only one possible target method, we instruct Ghidra to treat this reference as `primary`, which will lead to the decompiler rewriting the call to the actual implementation instead of the `objc_msgSend` runtime function. These references are then added to Ghidra and available via the Ghidra API.

Creating Object Layouts from Metadata. Objective-C binaries contain metadata in a dedicated section which describes the heap layout of instance variables for each type defined in the binary. Ghidra parses this format already, we use these results to create the representations for the object

layout on the heap. These data types can then be applied where type information is available, e.g., the receiver argument of a method signature. The decompiler then automatically rewrites memory accesses via pointer arithmetic into a dedicated `PTRSUB` instruction that represents a field reference.

Rewriting ARC Method Calls. We replace calls to ARC methods with P-Code that reflects the dataflow semantics of the method. For example, a call to `val = objc_getProperty(obj, offset, atomic)` will be replaced with the P-Code operations that represent `val = *(obj+offset)`. This only expresses the load from a calculated address, and ignores reference counting and locks for atomicity, neither of which matter for our analysis. This replacement happens early in the decompilation process, and the decompiler will further rewrite it to a proper field access `val = obj.field`.

Remaining Challenges. While our Objective-C plugin addresses several decompilation problems, certain complexities remain unresolved. Particularly around type inference, and context sensitivity for more complex dynamic dispatch handling. Our limitations are further described in Section V-A.

C. Data-flow Analysis with Heros on P-Code

The Heros plugin for Ghidra is the core component of our approach, leveraging Heros as an IFDS solver to perform dataflow analysis across Objective-C binaries. Key elements of our implementation include:

- **ICFG from Ghidra** Using Ghidra CFG recovery, we construct an ICFG where each P-Code instruction serves as a node.
- **IFDS Domain with HighVariables** To represent our dataflow domain, we use Ghidra `HighVariables`, providing a high-level abstraction of storage locations.
- **Flow Functions for P-Code** We define custom flow functions for P-Code operations.

ICFG Model. To analyze the dataflow of a binary with Heros, it is necessary to recover its intra- and inter-procedural control flow. Performing a CFG recovery is a non-trivial problem itself. We use the Ghidra internal CFG recovery to provide Heros with the relevant ICFG information.

To abstract the program logic, we model every single refined P-Code instruction as a node in our ICFG. Heros only requires the ICFG class to conform to an interface, that allows it to query information. These information include the successors of a given node, whether a node is a call node, and the target of a given call node. We implement this functionality on top of the information from Ghidra and thus have no need to maintain a separate data structure for the ICFG.

IFDS Domain. The high-level concept of a variable is no longer present in compiled code. Hence, it is a challenge to model a suitable domain to perform taint analysis with Heros at binary level. We again make use of the Ghidra decompiler to overcome these obstacles.

For our approach, we selected the Java object `HighVariables` from the Ghidra framework as our domain rather than `Varnodes`. The difference is, that `HighVariables` represent grouped elements sharing a common conceptual storage location, thereby functioning as aliases. To achieve field-sensitivity, we extend this concept with the additional `HighField` subtype. This type represents a storage location of an object field, tied to another `HighVariable` as parent container.

P-Code Flow Functions. To analyze the dataflow through our ICFG we model all Flow Functions that the IFDS framework requires, especially for inter-procedural analysis as shown in Figure 3. For our approach, we did as follows.

▷ ***Call Flow Function.*** We transfer the arguments and their associated `HighFields`, to the parameters and their `HighFields` in the callee. For performance, we kill all local variables from the caller.

▷ ***Return Flow Function.*** We transfer the input of the `RETURN` P-Code operation to the output variable of the `CALL` statement at the callsite. Additionally, we transfer the fields of parameters inside the callee back to the fields of their arguments in the caller.

▷ ***Call to Return Flow Function.*** We kill all global variables and all fields of objects that were passed into the callee as arguments, as they could be modified inside the callee.

▷ ***Normal Flow Functions.*** In theory, the Normal Flow Functions composed for intra-procedural analysis would need to be written for each individual statement in the P-Code IR, if accounting for all possible programs. In practice, we only needed to model flow functions for 4 of the 73 P-Code operations to achieve our results: `COPY`, `MULTIEQUAL`, `LOAD`, and `STORE`. These operations capture most relevant semantics of decompiled Objective-C binaries, with remaining P-Code operations covering control flow, and arithmetic on primitive. As Objective-C dataflows rarely involve such arithmetic, we consider it a future addition.

`COPY` is a straightforward *Transfer* for scalar types, and, for composite types, the *Compose* of the *Transfer* Flow Functions of its members. `MULTIEQUAL` is the ϕ-Node statement in P-Code, and uses the existing *Union* Flow Function of Heros. The `LOAD` and `STORE` P-Code operations are both handled similar to each other: They take a location as an argument, and either return the value, or take the value as a second argument. In both cases, we turn the location argument into the `HighField` element of our domain, by extracting the field information from `PTRSUB` operation that defined the location argument to the `LOAD`/`STORE`. The flow function for the `LOAD`/`STORE` is then a *Transfer* between this `HighField` and the domain element representing the value being moved.

IV. Our Objective-C Dataflow Benchmarks

This section presents our approach to evaluate our dataflow analysis. We introduce a new benchmark suite publicly available, specifically tailored for this language. Due to the complexities inherent in fully analyzing real-world iOS applications, as discussed in Sections II-B and V-A, we focus on isolated dataflow cases rather than comprehensive application scenarios. Our benchmarks target Objective-C as

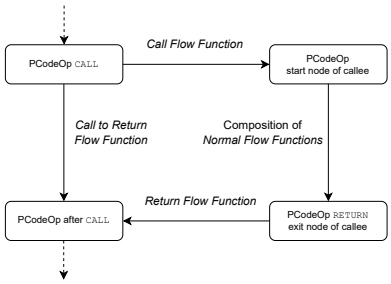

Fig. 3. Flow Functions used during inter-procedural flow.

a language, excluding iOS-specific features, to allow a controlled, language-centered assessment of approaches

A. Benchmark Considerations

Our benchmark suite draws inspiration from cases in established benchmarks, such as DroidBench, as well as isolated examples collected from real-world applications. Cases range from basic tests for flow and object sensitivity, to complex scenarios that require context-sensitive control flow graphs and detailed type information. We address challenges unique to binary analysis, for instance, float arguments returned in special registers and written directly to the stack for format strings. Given page limitations, we cannot exhaustively describe all cases; however, we consider this collection an evolving project that will expand with contributions from the community. The current cases are categorized to reflect the diversity of Objective-C taint analysis challenges: *General Objective-C*, *Global Variables*, *Field and Object Sensitivity* (further divided into *Field Access* and *Field Access with Depth*), *Aliasing*, *Control Flow Structures* (e.g, loops, if-else statements), *Context-sensitive Dispatch*, and *C Interoperability*.

B. Evaluation Methodology

Our benchmark suite is designed to highlight Objective-C taint analysis challenges. Our cases exceed the current capabilities of our plugins on purpose, while aiming at facilitating community collaboration on Objective-C binary analysis.

Accordingly, for our approach evaluation, we focus on a subset of benchmark categories that align with the goals of our current implementation: achieving inter-procedural field-sensitivity. Our tool is explicitly configured to identify unsupported cases, ensuring that only relevant capabilities are tested, to prevent misleading results. This subset includes basic cases and distinguishes between field-insensitive, field-based, field-sensitive, and field-sensitive with depth approaches.

C. Evaluation Results

We evaluated our approach against our benchmark suite to assess its effectiveness in addressing Objective-C data-flow analysis. The subset of cases chosen for testing is presented in Table I and provides insights into the plugin capabilities. Across a total of 91 cases, our plugin was applicable to 65, and successfully handled 52, without timeout, demonstrating strong coverage with high precision and recall. In the largest applicable category *Field Access*, consisting of 38 cases, we observe a high precision of approximately 89% and a moderate recall of 64%. We trace the moderate rate of false negative cases back to limitations, discussed in Section V-A. In simpler categories, such as *General Objective-C* and *Control Flow Structures*, our plugin achieved perfect precision and recall, indicating strong performance in straightforward data-flow scenarios. During our evaluation, we observed a timeout for one benchmark case in the *Global Variables* category, which was therefore not considered for the calculation of precision and recall. Hence, in that category our plugin processed 5 cases with 80% precision (4/5) and 100% recall. While being out of our current scope, the intra-procedural *Aliasing* cases were handled by relying on Ghidra decompilation variable recovery. However, as discussed in our limitations, we make no claim on these capabilities. Other challenging categories, such as *Context-sensitive Dispatch* and *C Interoperability*, remain unsupported, and indicate specific areas for further extension.

As an outcome, our implementation achieves an inter-procedural, field-sensitive and flow-sensitive analysis. For nested fields, we do not employ approximations and loose dataflow in such cases.

TABLE I
HEROS PLUGIN RESULTS ACROSS OBJECTIVE-C BENCHMARK CATEGORIES

Categories	Cases	TP	TN	FP	FN	Precision	Recall
General Objective-C	2	1	1	0	0	1/1	1/1
Global Variables	6	4	1*	1	0	4/5	4/4
Control Flow Structures	14	7	7	0	0	7/7	7/7
Field & Object sensitivity							
▷ Field Access	38	16	11	2	9	16/18	16/25
▷ Field Access with Depth	6	-	-	-	-	-	-
Aliasing							
▷ Intra-procedural	5	3	2	0	0	3/3	3/3
▷ Inter-procedural	4	-	-	-	-	-	-
Context-sensitive Dispatch	3	-	-	-	-	-	-
C Interoperability	13	-	-	-	-	-	-
Total	91	31	22	3	9	31/34	31/40

► * : Timeout.
► - : N/A, category not yet supported by our Heros plugin.

V. DISCUSSION

A. Limitations and Future Work

While our current prototype has certain limitations, we argue that these research directions extend beyond the scope of this initial exploratory work. We view these aspects as challenges for refinement and further development, building on the foundation presented in this paper.

Lack of Type Inference. The dynamically typed nature of Objective-C increases both challenges and necessity for accurate type inference. This problem is likely solvable by applying conventional type inference algorithms, particularly by leveraging recent advancements in the field [4], [16].

No Integration with Library Summaries. As we target the challenges of standalone Objective-C as a first step, we do not tackle the challenge of library summaries yet. Existing approaches such as StubDroid [17] can potentially be adapted to the iOS app ecosystem.

This also extends to the problem of tracking taint through data structures such as arrays and dictionaries, as these are

almost always accessed via method calls in Objective-C. This is different from other compiled languages such as C, where algorithmic data structures are more likely to be implemented as part of the same program or statically linked into it.

Context and Path Sensitive Control Flow. We did not attempt to solve context or path sensitive control flow, as the Ghidra CFG does not provide this information by default. We thus cannot resolve dynamic dispatch targets if their receiver object or method name comes from a caller function or is path sensitive. As we under-approximate function call targets, we loose dataflow in these cases.

Unclear Aliasing capabilities. We make no claim to solve intra-procedural aliasing. However, the decompilation variable recovery solves the intra-procedural aliasing cases of our benchmark suite. We do not provide any alias awareness beyond this benefit of decompilation and consider this to be a separate challenge under active research [18].

Unknown Scalability. Our plugin has not been evaluated for performance on real-world apps, and as such we make no claims about its scalability. Moreover, this performance evaluation cannot be sensibly performed before the other limitations discussed above are addressed. For example, the chosen memory model used for unstructured memory accesses will have a significant impact on the overall performance and correctness of the results, while not necessarily having an observable impact on the benchmark cases.

B. Related Work

Another work using binary decompilation to tackle Java Native Code (JNI) for Android has faced similar challenges and limitations to ours [19]. They state their first main challenge to be that the emitted C Pseudo-Code is not compilable and thus cannot be directly used in source code analyzers. Their second challenge is that the imprecise type inference will lead to spurious casts that degrade analysis results.

We improve upon their work by preempting their first challenge and using the IR instead of the C Pseudo-Code. The P-Code IR has a unique semantic that is sufficient for taint analysis [20], whereas C Pseudo-Code can contain implementation-defined, or undefined, behaviors.

We concur that type analysis is the main remaining challenge for this approach limiting its results, but we do not consider this to be a challenge of the approach itself. Type and Variable Recovery is one of the three core problems in decompilation research [21], and should thus be seen as a problem of the decompilation component itself, and not the data-flow analysis component, coupled in our approach.

VI. Conclusion

This paper presents the application of the Heros IFDS solver to Ghidra P-Code, enabling data-flow analysis on Objective-C binaries through our custom plugins. Our approach addresses Objective-C decompilation challenges and introduces a benchmark suite tailored to Objective-C, facilitating structured evaluation of static analysis tools. The results demonstrate the feasibility of adapting traditional static analysis frameworks to decompiled IR of Objective-C binaries.

Acknowledgment

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. Additionally, we want to acknowledge Angelo DeLuca, who contributed significantly to the Objective-C plugin implementation.

References

[1] D. Domínguez-Álvarez, A. Gorla, and J. Caballero, "On the usage of programming languages in the ios ecosystem," in *SCAM*. IEEE, 2022, pp. 176–180.

[2] X. Meng and B. P. Miller, "Binary code is not easy," in *ISSTA*. ACM, 2016, pp. 24–35.

[3] D. Orikogbo, M. Büchler, and M. Egele, "Crios: Toward large-scale ios application analysis," in *SPSM@CCS*. ACM, 2016, pp. 33–42.

[4] M. Noonan, A. Loginov, and D. R. Cok, "Polymorphic type inference for machine code," in *PLDI*. ACM, 2016, pp. 27–41.

[5] I. Smith, "Binsub: The simple essence of polymorphic type inference for machine code," *CoRR*, vol. abs/2409.01841, 2024.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON*. IBM, 1999, p. 13.

[7] 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 *PLDI*. ACM, 2014, pp. 259–269.

[8] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *NDSS*. The Internet Society, 2015.

[9] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the evolution of android app vulnerabilities," *IEEE Trans. Reliab.*, vol. 70, no. 1, pp. 212–230, 2021.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS*. The Internet Society, 2011.

[11] EC SPRIDE Secure Software Engineering Group, "DroidBench," 2013. [Online]. Available: https://github.com/secure-software-engineering/DroidBench

[12] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *CCS*. ACM, 2018, pp. 1137–1150.

[13] National Security Agency, "Ghidra Reverse Engineering Framework," 2019. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[14] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and soot," in *SOAP@PLDI*. ACM, 2012, pp. 3–8.

[15] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*. ACM Press, 1995, pp. 49–61.

[16] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *NDSS*. The Internet Society, 2011.

[17] S. Arzt and E. Bodden, "Stubdroid: automatic inference of precise data-flow summaries for the android framework," in *ICSE*. ACM, 2016, pp. 725–735.

[18] H. Li, C. Shi, J. Lu, L. Li, and J. Xue, "Boosting the performance of alias-aware IFDS analysis with cfl-based environment transformers," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 2633–2661, 2024.

[19] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of JNI programs via binary decompilation," *IEEE Trans. Software Eng.*, vol. 49, no. 5, pp. 3089–3105, 2023.

[20] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, "A formal semantics for p-code," in *VSTTE*, ser. Lecture Notes in Computer Science, vol. 13800. Springer, 2022, pp. 111–128.

[21] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy sailr! there is no need to DREAM of C: A compiler-aware structuring algorithm for binary decompilation," in *USENIX Security Symposium*. USENIX Association, 2024.