David Mißmann

Detecting Misuse of System-Provided Cryptographic Functions in iOS Binaries Using Static Analysis

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Johannes Feichtner
Prof. Stefan Mangard

Institute for Applied Information Processing and Communications

Graz, May 2016

AFFIDAVIT

Date	Signature

Abstract

With the wide spread of smartphones and the various applications that are run on them, the amount of processed personnel and sensitive information increased. Developers of applications often assure that this kind of data is secured against leakage to unauthorized parties by using cryptographic functions. However, this assurance only holds, if the parameters for these cryptographic functions are chosen correctly. In this thesis, we present a static analysis method that is able to detect cryptographic misuse of cryptographic functions provided by iOS. We present a solution for decompiling iOS binaries to the LLVM intermediate representation (IR) for performing a static analysis. The decompiled code reproduces not only the behavior of the program, but also keeps the way of how data is stored and accessed in registers by simulating this with a register-like data structure. The analysis itself is based on static program slicing and this method was extended to be able to extract execution paths from program slices, which only contains those instructions that modify a parameter of interest. Program slicing heavily relies on a supporting pointer analysis that provides information about where pointers point to during execution for computing data dependencies. We present a modified version of Andersen's pointer analysis that is able to handle the decompiled LLVM IR with the imitated registers of a CPU. To detect cryptographic misuse, we defined a set of rules that covers symmetric encryption of the CommonCrypto library that is provided by the system. These rules check the execution paths generated by our implementation of program slicing to find violations.

Our implementation was evaluated on open source applications to verify the correctness of the analysis results, as well as to find weak spots of our method where it leads to false results. Further, a set of publicly available applications was downloaded from the App Store to check, if these published applications use cryptographic functions in a correct way. It turned out that 49 of 51 applications, for which our analysis was able to provide results, violate at least one rule. The most common violation, found in 34 of the 51 applications, was that a non-random initialization vector was used. Another often observed violation, spotted in 32 of these applications, was the use of a constant key or password for encryption, which completely discloses the underlying data.

Kurzfassung

Durch die weite Verbreitung von Smartphones und der Vielzahl an Applikationen, die darauf ausgeführt werden, steigt auch die Menge an persönlichen oder sensibler Daten. Entwickler solcher Applikationen versichern daher oft, dass diese Art von Daten vor dem Zugriff Unberechtigter durch das Verwenden kryptografischer Funktionen gesichert sind. Diese Sicherheit kann aber nur gewährleistet werden, wenn die Parameter für diese Funktionen korrekt gewählt wurden. In dieser Arbeit präsentieren wir eine Methode, die mittels statischer Analyse in der Lage ist diese Parameter für, die von iOS bereitgestellten kryptografischen Funktionen, auf ihre Sicherheit zu überprüfen. Wir stellen ein Verfahren vor, dass in der Lage ist, iOS Binärdateien in die LLVM intermediate representation (IR) zu dekompilieren, um diese dann in der statischen Analyse zu verwenden. Dieser dekompilierte Code gibt nicht nur das Verhalten und den Ablauf der verwendeten Applikation wieder, sondern auch die Art und Weise, wie Daten mittels Registern verarbeitet werden. Dies wird über eine, den Registern des Prozessors nachempfundene Datenstruktur umgesetzt. Die Methode zur Analyse basiert auf Static Program Slicing, das um die Möglichkeit zur Erstellung von Ausführungspfaden erweitert wurde. Diese Pfade enthalten nur jene Instruktionen, die einen spezifischen Parameter einer kryptografischen Funktion modifizieren. Program Slicing hängt stark von einer unterstützenden Analyse zur Berechnung der referenzierten Werte einer Pointer Variablen zur Laufzeit ab. Wir präsentieren eine modifizierte Version von Andersens Pointer Analyse, die in der Lage ist den vom Decompiler erstellten Code, mit seiner Register-ähnlichen Datenstruktur, handzuhaben. Die falsche Verwendung von kryptografischen Funktionen ist über Regeln, die die symmetrische Verschlüsselung, mittels der von iOS bereitgestellten CommonCrypto Bibliothek, betreffen, definiert. Diese Regeln überprüfen die Pfade, die durch das Static Slicing generiert wurden, auf Verletzungen dieser.

Die Implementierung wurde mittels mehrerer Open Source Applikationen auf die Genauigkeit der gelieferten Ergebnisse evaluiert, aber auch um die Schwachstellen, die ungenaue oder falsche Ergebnisse liefern, aufzuzeigen. Weiters wurde das Framework auch auf Applikationen aus dem App Store ausgeführt, um abzuschätzen, in welcher Weise kryptografische Funktionen in veröffentlichten Applikationen verwendet werden und in 49 von 51 Applikationen, mit denen eine Analyse durchführbar war, wurde zumindest eine Regel verletzt. Die am öftesten Verletzte Regel betrifft das Verwenden eines nicht-zufälligen Initialisierungsvektors. Diese wurde in 34 der 51 Applikationen verletzt. Weiters wurde in 32 dieser Applikationen ein konstanter Schlüssel, beziehungsweise ein konstantes Password zum Verschlüsseln entdeckt, wodurch die Verschlüsselung zwecklos ist.

Acknowledgements

First, I would first like to thank my advisor Johannes Feichtner for guiding me through this thesis, as well as for his help with my masters project that raised my interest in this topic and led to this thesis. His comments and suggestions, especially on the drafts of this thesis, were invaluable.

Further, I want to thank my parents and my sisters for the endless support throughout my years of study and my life in general. This accomplishment would not have been possible without them.

Finally, I would like to thank my other half, Christina, for her backing and constant motivating words together with her belief in me.

David Mißmann
Graz, Austria, May 2016

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Nowadays, smartphones are present in almost everyone's daily life. They are used for a large variety of tasks and some of them process critical data that may be processed with cryptographic functions to ensure security. This security heavily relies on the correct choice of the parameters for these cryptographic functions. If these functions are used in a wrong way, private or sensitive information may be exposed to other parties. In the worst case scenario, the "secured" information is completely disclosed while other incorrect used functions may weaken the security. Unfortunately, the developers of applications often do not publish details about the exact parameters passed to critical functions. The only source of informations for these applications, to check whether cryptographic functions are used correctly or not, is to inspect the compiled binary. Depending on the platform those binaries usually contain either plain machine code or an other low level representation of the application. A manual inspection of applications is not feasible, considering the size and complexity of applications nowadays. In addition to size and complexity, the original source code of an application will, most likely, not be available. So, only a compiled binary executable is available that is not easily readable and makes a manual analysis exhausting and error-prone.

This work targets these problems for the iOS platform and presents a framework that is capable of checking for the correct use of cryptographic functions provided by the iOS system in an automated fashion. To provide this, the program binary is decompiled into a higher abstraction language, an intermediate representation, which is more suitable for program analysis, where decompiling describes the inverse operation to compiling a program. This intermediate representation is a translation of the whole application, which means that it contains all functions and statements of this application. The statements that have to be checked for ensuring correct use of cryptographic functions, are usually a small subset of the whole application. Program slicing is a method that retrieves only the relevant statements of an application where the relevant statements are identified by computing dependencies originating in a statement defined by the so-called *slicing criterion*. Relevant means that only those statements are included in a slice that affect a parameter passed to a cryptographic function. Program slicing needs the information about which values are modified and referenced at each statement. Generating this information is not a trivial task since iOS applications are developed in languages that support pointers. A single pointer variable may point to multiple locations and different pointer variables may point to the same location during execution. Without having this information about pointers, program slicing will not be able to compute the slices correctly since the referenced and modified values are incorrect. To obtain this knowledge, a pointer analysis is performed that generates *points-to-sets*, which contain a set of locations for each pointer variable and each of those locations might be pointed to by this variable.

However, having a program slice that contains only the interesting parts that are related to the use of cryptographic functions does not provide the information whether a specific parameter has the correct value or not (correct means that the parameter does not violate a defined rule). The relevant parameters for a rule to check are backtracked to their origin using the information produced during program slicing, which includes information about pointer variables.

Being able to backtrack parameters in a sliced program produced from decompiled code of an iOS binary it is possible to determine, if cryptographic security is not fully given in an application of interest.

1.1 Problems Addressed in this Work

In this work we present a system that is capable of performing a static program analysis on binaries for the iOS operating system. To achieve this, various challenges have to be solved. First of all, there is basically only the binary available that contains machine code, so no higher level source code is accessible for analysis.

1.1.1 Simplifying Machine Code and Reducing Platform Dependency

Having an application binary alone means that only machine instructions are available to analyze its behavior. Performing analysis on this low level is very error-prone and tedious. Additionally, running an analysis on this level of abstraction would result in requiring a separate implementation of the whole analysis framework for each architecture. To overcome this issue of being bound to a single architecture and having the hard to handle machine instructions, we will translate the input binary with its machine instruction to a language of higher abstraction, the LLVM intermediate representation (IR). This language is still an assembly language, but it has the benefit of being platform independent and there are already numerous implementations of static analysis methods of which some can be used.

1.1.2 Source Language Specific Problems

Even though the binary was compiled it will still contain characteristics of the applications original source language and they will be transferred to the LLVM IR code as well. Most iOS binaries are compiled from the Objective-C language, which is a super-set of the C language. While there is another language called *Swift* that can be used for developing iOS binaries this work will only handle applications developed using Objective-C. Objective-C is a so-called *runtime oriented language*, which means that as many decisions as possible are made during runtime. This holds especially for calling methods. The decision which method has to be called is made by the Objective-C runtime library. The application itself only tells the runtime library the name of the method to call and for which instance or class the method should be called. This information is passed to the library via so-called messages. The code for the creation of those messages is located in the binary and analyzing these parts of the code it is possible to reconstruct the messages. Based on the information about the instance or class and the method name the candidates that may be called can be found.

Getting the "content" of these messages is a crucial requirement before pursuing a further analysis because they define the control flow between functions and the data, that has to be tracked in this work, will usually pass through multiple function before being used.

1.1.3 Static Analysis

The static analysis in this work is done to identify instructions and variables that have an impact on a specified parameter for a cryptographic function. An essential point is to know which values are referenced by a variable in the program. Machine instructions have only a limited set of registers, provided by the CPU, available for use in instructions as operands. These registers are often used to reference locations in memory for accessing data there. This is similar to the concept of pointers in programming languages where different pointer variables can point to the same location in memory. Getting the information where different variables, or in our case registers, might point to during execution is a necessary requirement before being able to compute dependencies inside the program. Having this information is also required by the previously mentioned message sending in Objective-C for resolving function calls since messages are only references to class and method informations stored in the binary and they are accessed by pointing to this information using registers.

Using this information it is possible to determine the statements of a program that are relevant for a specific value, i.e. the statements that basically set the value. However, this only reduces the number of statements that have to be inspected for finding those parts of the program that may use cryptographic functions in a wrong way. The remaining problem is to find the flow of information that ends up in a variable of interest. This means that sequences of statements have to be found that set the final value (final means the value that is used in a cryptographic function) of a variable.

1.2 Structure of this Document

The introduction chapter is followed by a chapter of related work that gives a brief summary about previously published work that is relevant for this thesis.

After these first two chapters the document is divided into three major parts: background, implementation and evaluation.

The first part focuses on the background knowledge required for the implementation. In Chapter 3 the decompilation is described by outlining the relevant information about the supported CPU and the already existing *Dagger* decompiler. Chapter 4 shows how static program slicing is applied to extract a subset of the applications instructions that influence a variable of our interest. The required pointer analysis method, and its modifications to fit the needs of the decompiled code, are discussed there as well and further it is described how information about object types is partly restored to allow an analysis. This also includes the detection of calls to functions that are not defined in the binary, but in some external library. Chapter 5 extends the method of program slicing to find execution paths of relevant values by backtracking the values using the information gained from static slicing. Chapter 6 defines how the analysis decides whether cryptographic misuse happens or not by describing a set of rules regarding symmetric encryption operations.

The second part of this thesis discusses the implemented framework. Chapter 7 describes the changes made to the existing *Dagger* framework to allow the generation of an intermediate representation from an iOS application that can be used for static analysis. It also describes the required modifications of the used program slicing implementations *LLVMSlicer* to work with the implemented pointer analysis and how external functions are handled during slicing and backtracking.

In the third part of this work the implementation was executed with various iOS applications for evaluating it. In Chapter 9 open source applications are taken as input for the implemented framework to show the correctness of the delivered results as well as discussing those cases where the results are inaccurate or wrong. After that a set of applications published on the App Store was downloaded which then again is evaluated to point out the common mistakes regarding cryptography, that are made in public available applications. Chapter 10 concludes the found results and describes possible improvements that would help to deliver more accurate results.

Chapter 2

Related Work

This chapter surveys previous work done in the field of analyzing applications for mobile devices. In Chapter 2.1 we focus on existing research done in the field of program slicing, on which our approach is based on. Chapter 2.2 highlights the existing approaches in pointer analysis, which supports program slicing at computing data dependencies for delivering accurate results.

Static analysis for iOS binaries was previously done by Egele et al. [6]. The binaries they have used were only disassembled and the disassembled code was then analyzed. They created a control flow graph (CFG) from the binaries that were compiled from Objective-C code and then did a reachability analysis to identify possible privacy leaks. Due to the characteristics of Objective-C they had to recover type information, which is usually not available in assembly code, to be able to create a correct CFG.

On the most used mobile platform, Android, there are more publications for doing static analysis. In [7] they evaluated Android applications to identify applications that are using cryptographic APIs in a wrong way. They translated the Dalvik bytecode into an intermediate representation that is used to create a CFG. Using static slicing they tried to find the origin of the parameters used to call a cryptographic API and checked them against some defined rules.

Hoffmann et al. [14] have created a framework that disassembles Dalvik bytecode to a *smali* representation and with this code the framework performs static slicing for backtracking parameters.

2.1 Program Slicing

Program slicing is used in this work to create a smaller program of the original application that contains only those parts that affect the parameters that are used in cryptographic functions.

Computing program slices was discussed in several ways before. It was originally introduced by Weiser [26] as an interprocedural approach. The slices were computed using dataflow equations, which describe the dependencies between statements (such as control dependencies and data dependencies).

Ottenstein and Ottenstein [18] used a Program Dependence Graph (PDG) [9] for creating intraprocedural slices by defining the problem as a reachability problem. In the graph itself statements are represented as nodes and edges describe data dependencies and control dependencies. The slicing criterion in this case is a node and the corresponding slice contains all nodes that are reachable from the node of the slicing criterion. PDGs are generated only for single procedure and hence they can not be used to create intraprocedural-procedural slices. However they can be extended to System Dependence Graphs (SDG) [15] where dependencies between procedures are described as well and creating slices using a SDG works exactly as when using a PDG.

Agrawal et al. [1] presented a solution that creates static program slices using a PDG that handles pointers and arrays. However in their work only a intraprocedural-procedural solution was described.

Lyle and Binkley [17] extended Weiser's [26] method to work with pointers by using a variant of symbolic execution. Statements that assign a pointer value are identified and symbolic values are generated for those. These symbolic values are then propagated throughout the program and this information is used during program slicing to identify variables that are pointed to.

Jiang et al. [16] also used dataflow equations and included handling of pointer types. For addresses of variables and variables that are pointed to *dummy* variables are introduced and those new variables are included in the dataflow equations. However, in [23] it is shown that the produced slices are not correct in all cases and some relevant statements are not included in the slice, if values, which are pointed to, get modified. They use information provided by the alias analysis (similar to pointer analysis) of Weihl [25].

Binkley and Harman [5] have shown the impact of supporting analysis methods for program slicing. The most important supporting method is pointer analysis.

Shapiro and Horwitz [21] compared the influence of different pointer analysis methods [2, 22, 20]. Using a more precise pointer analysis not only produces a better slicing result but also needs less time for computing the slice. Of the three compared approaches Andersen's [2] was the most precise, but hast the worst performance in respect to time. Steensgard's [22] algorithm runs in almost linear time having worse results though. Shapiro's and Horwitz's [21] method in both precision and runtime between the other two.

2.2 Pointer Analysis

Andersen [2] defines both a context-sensitive and a context-insensitive algorithm (in his work he uses the term intra-procedural for context-insensitive and inter-procedural for context- sensitive). In general context-sensitive means that different calls to the same function are distinguished, which is not done in an context-insensitive algorithm. Andersen's approach provides very accurate result, but it has a runtime of $O(n^3)$.

The flow-insensitive and context-insensitive pointer analysis of Steensgaard[22] runs in almost linear time with an upper bound of $O(n * \varphi(n, n)^{-1})$ where φ^{-1} is the inverse Ackermann, which increases very slowly, thus the term almost linear time is used. It is based on a non-standard type system (these types have nothing to do with the types used in programming languages) where two pointers may point to the same location, if they have the same type. Aliases for each pointer are found using a type inference algorithm that is based on constraint solving.

The main difference between Andersen's and Steensgaard's pointer analyses is that Andersen uses inclusion relations and Steensgaard uses equality relations. This means that having an assignment $x = y$ Andersen says that anything that is in the points-to set of y has to be in the points-to set of x as well and in Steensgaards pointer analysis both points-to sets have to be equal. Point analysis methods that use inclusion relations are generally referred as *Andersen-style* analyses and *Steensgaard-style* when using equality relations.

As already mentioned Andersen's initial algorithm runs in $O(n^3)$ and as the programs analyzed in this work tend to have millions lines of code this upper bound would lead to a poor overall performance. There are various Andersen-style analysis methods [12, 19, 4, 10, 11] that improve the performance issue that comes up when using Andersen's pointer analysis. Hardekopf and Lin [10] improved this approach to a level where the analysis can be done in almost linear time like Steensgaards method while producing results similar to those delivered by Andersen's algorithm and they show that their method runs significantly faster than other Andersen style approaches [12, 19, 4]. However, the used optimizations in this work are those described by Hardekopf and Lin. In [11] a constraint optimization is described that merges the definitions of similar pointer variables so the input size for constraint solving get smaller and [10] targets the constraint solving process by detecting strongly connected components in a graph. These components get collapsed to a single node since they will form a cycle and each node in this cycle will point to the same locations.

Part I

Background

Chapter 3

Decompilation

Decompilation describes the task of translating low level machine code to a language of higher abstraction. The goal of decompilation in this work is getting a representation of a binary, which was compiled for iOS and the ARMv8 CPU architecture, in the LLVM Intermediate Representation[1]. Since the only purpose of this decompiled code is to perform static analysis on it, it does not need to be executable. Nevertheless, it has to represent the data flow of the program correctly to be able to identify the possible paths a value can take during execution.

Similar to the process of compiling programs, every architecture has to be handled separately and for iOS devices this means that both used architectures, ARMv7 and ARMv8, have to be handled independently to fully support decompilation of iOS binaries. However, the main focus in this work is set on performing static analysis on this decompiled code and not the decompilation itself and for this task decompiling a single architecture is sufficient. The newer ARMv8 architecture is, due to the simplification of the instruction set, a little easier to handle and thus it was chosen to be used in this work.

The input for decompiling is a binary that contains the machine instructions for the targeted CPU and information about external libraries as well as language specific information and other data that is required during runtime. Binaries are generated using a compiler that translates a program defined in a source language to machine instruction. During this process a lot of information gets thrown out like function headers, local variables or type information. By losing this basic information some other related data gets lost as well. It is not possible to generate a valid call graph without having knowledge about the types of local variables or function parameters. However, this information is crucial for performing static analysis and therefore needs somehow to be reconstructed.

The sections in this chapter discuss how the information of the compiled binaries is used to create a usable intermediate representation. Section 3.2 describes the already existing basic implementation of the decompiling framework *Dagger* that is extended to handle the binaries of iOS applications. The file format of the binaries, Mach-O, and how the necessary information is extracted from such a file so it can be used in static program analysis is described in Section 3.3.

The implementation details are discussed later in Section 7.1 where the details about the rest of the system are described as well.

3.1 The ARMv8 Architecture

All iOS devices use CPUs of the ARM family. Since 2013, new devices have a 64-bit ARMv8 CPU while older devices run on a 32-bit ARMv7 CPU. The newer ARMv8 architecture is fully backward compatible with ARMv7. So it is able to execute binaries compiled for the ARMv7 architecture. Still,

[1]`http://llvm.org/docs/LangRef.html`

there are differences that are relevant for decompiling binaries for these two architectures. However, as already mentioned above the implementation of the decompiler in this work is only made for the newer ARMv8 family.

This section briefly describes what has to be considered when a binary has to be decompiled for this architecture.

3.1.1 Instruction Sets

ARMv8 processors support the following instruction sets:

- A64 (32-bit fixed size instructions)
- A32 (32-bit fixed size instructions)
- T32, Thumb and Thumb-2 (32-bit and 16-bit instructions mixed)

The A32 and T32 instruction sets are also used by 32-bit ARM CPUs and they are supported by ARMv8 for compatibility reasons. A64 was designed to work with 64-bit ARM CPUs only and this will be the only instruction set supported by the framework implemented in this work.

For executing programs with the instruction set A64, the CPU has to be in a distinct execution mode called AArch64. Execution of A32 or T32 instructions happens in the execution mode AArch32. The execution mode itself can not be changed during execution, but the instruction set may change. For decompiling this means that binaries, which have to be executed in AArch32 mode, may contain instructions of both instruction sets A32 and T32, while the ones that have to be executed in AArch64 mode have only instructions of the A64 instruction set. This means that a binary with A64 instructions can only contain instruction of this instruction set so there is no need to support any other instruction set as it would be the case with A32 or T32.

3.1.2 Registers

An ARMv8 CPU has two groups of registers: the general purpose registers (GPR) and registers for floating point or vector values (SIMD/FP). Compared to ARMv7 CPUs the registers are organized in a simpler way that makes decompilation and a later analysis easier.

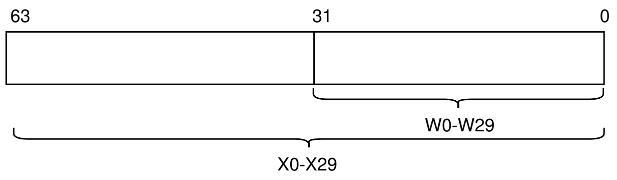
A single register has always multiple identifiers, no matter if it is a GPR or a SIMD/FP register. Those identifiers define the accessed region of a single register as shown in Figure 3.1. If a value is read from a register using an identifier, which accesses less than the full width of an register, the value only gets truncated to the required size. But in the other case, when a value is written to a register using this identifier, all bits of the register may change since the upper bits, which exceed the size of the value that gets stored there, are simply set to zero.

For the analysis this means that a register never keeps “old data”, if it gets set to a new value, no matter what size this new data has, so no additional data dependencies, referencing this previous data, will have to be computed.

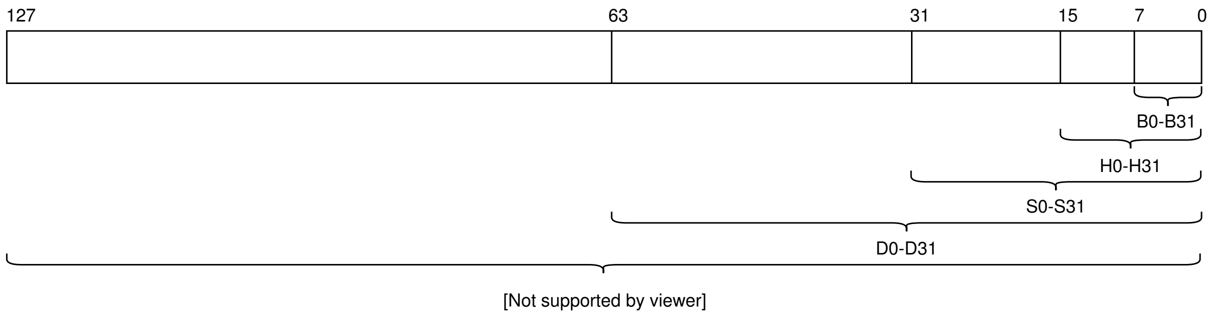
3.2 *Dagger*

The static analysis is performed on LLVM IR code for the reason of being capable of doing this on multiple architectures as well as the fact that LLVM IR code is simpler and better readable than plain assembly code and further, LLVM IR allows to use already implemented optimizations.

To generate a LLVM IR of an iOS binary *Dagger* is used. It extends the LLVM framework and tries to make use of the already defined backends and their description of registers and instructions. The main



(a) ARMv8 General Purpose Registers



(b) ARMv8 FP/SIMD Registers

Figure 3.1: The Figures 3.1a and 3.1b show that smaller parts of general purpose registers and FP/SIMD registers can be accessed via other identificators.

goal of decompilation in this work is to generate LLVM IR that correctly represents the control flow and data flow of an application and not to use this code again for compiling it and executing it. And this will indeed not be possible for decompiled iOS binaries since the contained data, which is referenced in the code, will not be included in the decompiled code. The references, however, will be translated into the LLVM IR and since these references are static addresses the data in the binary can still be accessed during analysis.

Section 3.2.1 describes what these definitions and in Section 3.2.2 the process of translating individual machine instructions to LLVM IR is discussed. The workflow of *Dagger* is sketched in Section 3.2.3. Representing a programs control flow is also crucial for decompiling and this is discussed in Section 3.2.3.1. The architectures supported by LLVM always use registers for storing and accessing data and *Dagger* uses a similar approach to reproduce this behavior as discussed in Section 3.2.4.

3.2.1 The LLVM Backend

The LLVM backend of a target provides the information that is needed to translate LLVM IR to target specific machine instructions during the process of compiling a binary. Parts of this target descriptions, namely the register and instruction descriptions, are used by *Dagger* to decompile machine instructions back to LLVM IR code.

The register description is used to model the processors register file. It describes the type of a register as well as its relation to other registers like an being an alias for another register or being a subregister of another register. *Dagger* uses this description to create a data structure that is able to simulate the same behavior as the CPUs register file. Section 3.2.4 discusses the way these descriptions are used by *Dagger*.

Instruction descriptions include the pattern definitions that are used to select the machine instructions for IR instructions during code generation[2]. The patterns are used by *Dagger* in the exact opposite direction. A given machine instruction is translated to a sequence of LLVM IR instructions that match the pattern described by this record in the LLVM backend. For most machine instructions these patterns are already defined, but not for all as it is not required by LLVM to do this for every instruction. Section 3.2.2

[2] `http://llvm.org/docs/CodeGenerator.html`

describes how these patterns are defined and used by *Dagger*.

Dagger uses so called *instruction semantics* to translate a single machine instruction to LLVM IR code and while for many instructions these semantics can be generated from the definitions that are used to generate a machine instruction from LLVM IR instructions during compilation by inverting this step. However, these definitions are not required to be available for all instructions and such missing definitions would prevent the decompilation of these instructions. To fill this gap of missing definitions they have to be specified so all instructions have a "LLVM IR counterpart".

3.2.2 Instruction Semantics

In *Dagger* instruction semantics describe how a machine instruction is translated to LLVM IR. During code generation LLVM tries to match sequences of instructions to a pattern that is defined for a machine instruction of the target machine. The decompiler gets the machine instructions as input uses these patterns to create a sequence of LLVM IR instructions that represents this machine instruction.

The TableGen[3] tool was extended to create instruction semantics using the patterns defined for machine instruction selection. The semantics describe the types of operands from the machine instruction and which operations have to be applied for getting an equivalent expression in LLVM IR and this is already the basic output of the decompiler.

How this is done is shown on the example of the instruction for adding an 64-bit integer and an immediate value as shown in Listing 3.1. The pattern that is used for selecting this instruction during code generation is shown in Listing 3.2 and means the following:

- **set** stores the second operand (the result of **add**) in the first operand (**$Rd**), which has to be a general purpose register, including the stack pointer, with an width of 64-bit
- **add** adds the value of the register **$Rn** and the operand **$imm**. The **$imm** operand corresponds to the last part of the ASM instruction in Listing 3.1: **#imm{, shift}**.

The generated semantics for this instruction in Listing 3.3 pretty similar, but this representation makes the semantics easily interpretable by the decompiler.

Line 1: Loads the value of type MVT::i64 from the register described in the operand with index 1 of the machine instruction and adds the value to the operand list.

Line 2: Tells the decompiler that the next operand is a custom operand that is of type MVT::i64 and its values are stored in the operands of the machine instruction starting at index 2

Line 3: Executes the addition of the first two elements in the operand list and adds the result to the operand list

Line 4: Stores the result of the addition (stored in the operand list at index 2) and stores it in the register declared by the operand in the machine instruction at index 0

Line 5: Tells the decompiler that this instruction is done

These instruction semantics are all stored in a single array and this array is accessed using a lookup table that matches machine instructions to an array element from which the decompilation of this instruction starts and all operations are executed until a `DCINS::END_OF_INSTRUCTION` identifier is reached.

[3] `http://llvm.org/docs/TableGen/`

```
ADD  Xd|SP, Xn|SP, #imm{, shift}
```

Listing 3.1: The A64 ASM instruction for adding an (shifted) immediate value to a 64-bit register and storing the result in an 64-bit register.
Xd|SP is the destination register
Xn|SP is the register of the first operand
#imm is the immediate value
shift is the shift operation that is applied to the second operand

```
[(set GPR64sp:$Rd, (add GPR64sp:$Rn, addsub_shifted_imm64:$imm))]
```

Listing 3.2: Pattern for the 64-bit ADD instruction

```
DCINS::GET_RC, MVT::i64, 1,
DCINS::CUSTOM_OP, MVT::i64, AArch64::OpTypes::addsub_shifted_imm64, 2,
ISD::ADD, MVT::i64, 0, 1,
DCINS::PUT_RC, MVT::isVoid, 0, 2,
DCINS::END_OF_INSTRUCTION,
```

Listing 3.3: The instruction semantics created by TableGen

3.2.3 *Dagger* Workflow

The decompilation itself consists of two stages as it is shown in Listing 3.4. First the binary gets disassembled and the machine instructions are grouped into basic blocks that represent the control flow (lines 1 to 22 in Listing 3.4). The second step iterates over the machine functions and its basic blocks of disassembled machine instructions, which get decompiled using the so called *instruction semantics* (see Section 3.2.2), into LLVM IR. This step basically fills the basic blocks with instructions and by defining the terminator instructions of basic blocks, the last instruction of a block that indicates which basic blocks are executed next, the edges of the CFG are specified. These two steps are independent of the architecture and only the decompilation of a single instruction has to be modified to support ARMv8 binaries.

```
1   for MI in MachineInstruction do
2       //Function starts are stored in the Mach-O binary
3       if MI.address in FunctionStarts then
4           createMachineFunction(MI.address)
5           //A function always has an entry basic block
6           createMachineBasicBlock(MI.address)
7       end
8
9       DIS = disassemble(MI)
10      if branchInstruction(DIS) then
11          //Only control flow changes inside a function are considered
12          if not callInstruction(DIS) then
13              if getTarget(DIS) then
14                  createMachineBasicBlock(getTarget(DIS))
15              end
16              //Control flow changes -> a terminator instruction
17              //in LLVM IR
18              createMachineBasicBlock(MI.address + InstructionSize)
19          end
20      end
```

```
21        addToMachineBasicBlock(DIS)
22    end
23
24    for MF in MachineFunctions do
25        switchToFunction(MF)
26        createAllBasicBlocks(MF)
27        for MBB in MF.MachineBB do
28            switchToBasicBlock(MBB)
29            decompileInstruction(MBB)
30        end
31    end
```

Listing 3.4: Rough workflow of decompiling a binary in *Dagger*

3.2.3.1 Control Flow Statements

Handling control flow statements is an important task while decompiling machine instructions. Control flow statements change the order in which statements are executed. LLVM IR puts all statements into basic blocks. That are sequences of statements with a single entry and exit point (all of them are executed or none). The exit point of a block is always a *terminator* instruction, which is a control flow statement that defines the succeeding basic blocks or returns from this function.

The A64 instruction set has various control flow statements. They can be either conditional or unconditional. Conditional statements change the control flow, if a condition is fulfilled and unconditional statements change the control flow always when they are executed. In the A64 instruction set has the following instruction that can change the control flow of an program:

- Conditional
  - **B.cond:** conditional branch
  - **CBNZ:** Compare and branch if not zero
  - **CBZ:** Compare and branch if zero
  - **TBNZ:** Test if bit is not zero and branch
  - **TBZ:** Test if bit is zero and branch
- Unconditional
  - **B:** Branch
  - **BL:** Branch and link (function call)
  - **BLR:** Branch and link register (function call using the value stored in a register)
  - **BR:** Branch register (branch using the value stored in a register)
  - **RET:** Return

All of the instructions above have to define a target address where the branch should end up. For most instructions this address is defined statically in the instruction, but some instructions (namely **BR, BLR** and **RET**) read the destination address from a register and this value can not be determined during decompilation.

For all possible targets of control flow statements a basic block has to be created. This is done by the decompiler when the machine instructions are disassembled. Every instruction is checked whether it is a control flow instruction and it is tried to determine the target (lines 10 to 20 in Listing 3.4). For instructions whose target depends on a register value this can not be done here. These values can be found later when a pointer analysis is done. The results of the pointer analysis for this register represent the possible targets for this instruction and they are limited to a small range of allowed values using the assumptions:

- **Branch address within the scope of the current function:** This assumption is made for the **BR** instruction. In LLVM IR a branch has to have a basic block in the same function as target.
- **A function pointer has to point to a function:** This may sound obvious, but it is possible to execute a **BLR** instruction to an arbitrary location. However, since BLR instructions are translated to *call* instructions in LLVM IR and they require this value to be a valid function pointer this assumption is made.

Results from the pointer analysis for function pointers do not need any particular treatment in the decompiling process, but branch targets have to be processed again. For the pointer values of branch targets a corresponding basic block has to be found. If no basic block is available for this address the decompiler has to split the block that contains the instruction of this address has to be divided into two blocks so a this new block can be targeted in a branch instruction. This is done by decompiling this function again because basic blocks are defined during disassembling this function and the LLVM IR instructions are inserted later into this blocks.

3.2.3.2 Function Calls and Parameter Passing

Function calls (`BL` and `BLR` instruction in ARMv8) are translated to `call` instruction in LLVM IR. The `call` instruction always passes a parameter, the register set, to the called function. This register set captures the current state of execution across function calls. In the scope of a function the register values are stored in local variables since this reduces the amount of `store` and `load` instructions that would be necessary, if the register set was updated after every modification of a register. However, to be able to share the register values between functions it is required to store the local values of the registers to the register set whenever it is used. For every function `call` statement three steps are necessary so register values are consistent across functions:

1. Store local register values to register set
2. Call function with the register set as parameter
3. Load values from the register set to local variables

and the called function has to do similar operations although they are executed in a different order:

1. Load registers to local variables
2. Execute function body
3. Store local variables to the register set right before returning to the callee

Storing the values right before returning to the callee also means that the returned values are stored at this point.

Passing parameters to functions happens through storing values in registers or on the stack (or both as defined in ARMv8 [3]). No additional work has to be done for parameter passing, since the called function retrieves the values from the registers and the stack using the register set.

This also means that we have no information about parameters or return values at all at this point, but static analysis requires this information to reconstruct data flow between functions. This implies that this information has to be reproduced in the actual static analysis where it is needed after all.

3.2.4 The Register Set

The CPU has a set of registers for storing operands and results of operations. *Dagger* creates a data structure, which is an array of elements with variable size, to store the register values that are used by the machine instructions. Data is shared across the program using this data structure. To achieve this a reference to this data structure is passed as argument of every function call in LLVM IR, so all functions basically access the same memory locations. It is the only parameter that is passed to a function in the decompiled code since the assembler instructions do not contain any information about function arguments. However, parameters are still passed to functions, but this happens using the registers that are represented by this data structure. To retrieve these parameters a called function has to extract them from the data structure of registers (Section 3.2.3.2 describes this in more detail).

The register description of a target also defines the hierarchical composition of registers via sub- and super-registers. There are two possibilities for the relation of a super-register to its sub-register(s):

- **1 : 1**: the super-register has a single sub-register.
- **1 : n**: a super-register consists of multiple sub-registers

Since the produced LLVM IR code is used for program analysis, it should be prepared so it can be used conveniently for doing this. Having super-registers of the form **1 : n** makes calculating data dependencies more difficult. Updating a register, whic has a super-register of the form **1 : n**, only the parts that store this particular register should change and the other parts have to keep their value (they depend on the previous value of the register). However, these dependencies do not display the actual data dependencies created by the machine instructions of the program, if the super-register is not directly accessible via machine instructions and this is the case when using the register descriptions for A64. The A64 instruction set has some instructions that require consecutive registers as operands. To ensure that only consecutive registers are for such instructions LLVM defines super-registers whose sub-registers are the actual registers that are accessed by such instructions. *Dagger* stores the current register values always in the largest available super-registers so all sub-registers can access the value from any super-register of this sub-register. This means that the value for one register is stored in multiple places and all those places have to hold the same value after updating a register. Figure 3.2 shows that for a single physical register all super-register pairs, triplets and quadruplets that contain this physical register have to be updated. For decompilation this means that simple register update has to be done multiple times so all superregisters contain the same value and this will increase the code length.

When a value has to be read from an register it is again fetched from one of the largest super-registers. In Figure 3.3 it is shown that multiple physical registers depend on a single super-register. For decompilation itself this does not cause any problems beside the increased code length, but this code is used later for static program slicing. For slicing it is crucial to find data dependencies and having super-registers creates additional, unwanted data dependencies. When a value for a physical register is loaded this value depends on the super- register from which it is loaded, but this super-register depends on all other values that were stored there as well. Due to these dependencies the slice will be bigger than necessary. To overcome this issue *Dagger* is modified to store only one physical register in a single super-register (one of the largest super-registers). Since a super-register stores only a single physical register now it only has a data dependency to this specific physical register. The super-register is still present and it still has the same size and identifier as before (see Figure 3.2), the value itself is only updated in the largest super-register and the remaining values, which were stored in this super-register before, are always zero (for example the super-register $\{Q_n, Q_{n+1}, Q_{n+2}, Q_{n+3}\}$ holds only the value for the physical register Q_n . $Q_{n+1}, Q_{n+2},$ and Q_{n+3} are set to zero)

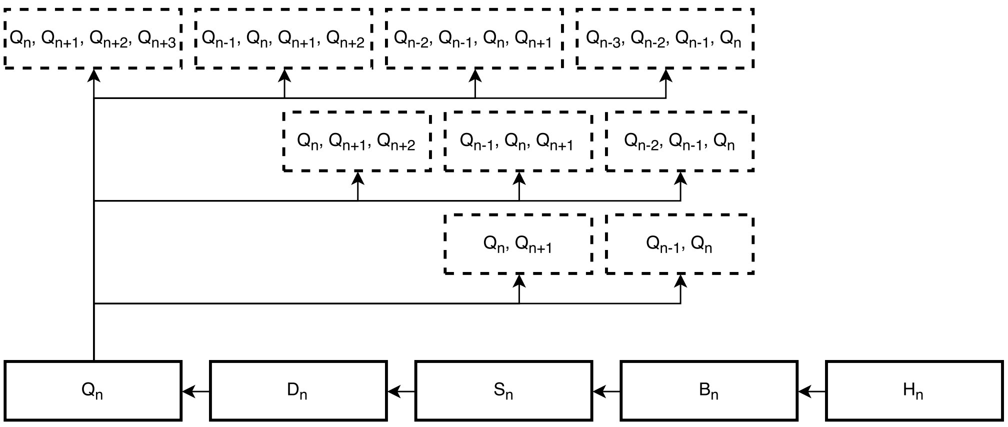

$Q_n, Q_{n+1}, Q_{n+2}, Q_{n+3}$
$Q_{n-1}, Q_n, Q_{n+1}, Q_{n+2}$
$Q_{n-2}, Q_{n-1}, Q_n, Q_{n+1}$
$Q_{n-3}, Q_{n-2}, Q_{n-1}, Q_n$
Q_n, Q_{n+1}, Q_{n+2}
Q_{n-1}, Q_n, Q_{n+1}
Q_{n-2}, Q_{n-1}, Q_n
Q_n, Q_{n+1}
Q_{n-1}, Q_n
Q_n
D_n
S_n
B_n
H_n

Figure 3.2: The value from a single SIMD physical register is written multiple super-registers

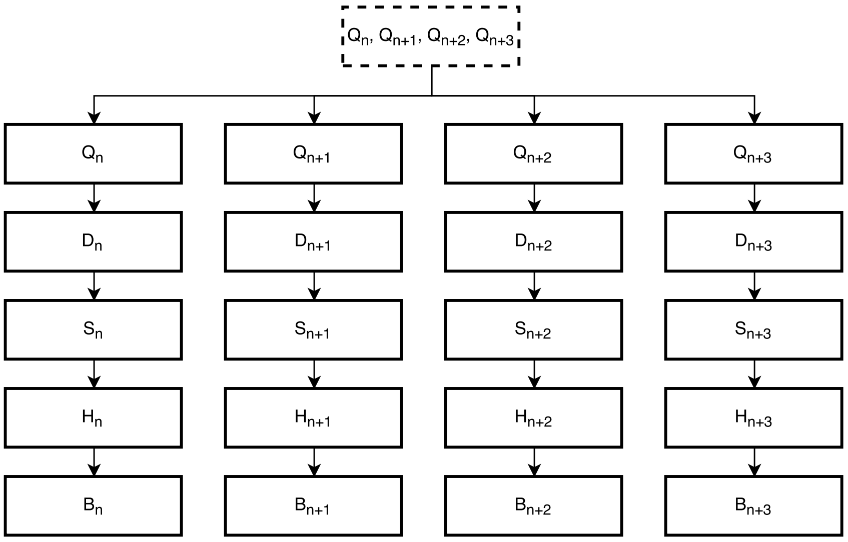


Figure 3.3: The largest super-register for the SIMD registers holds values of four physical registers

3.3 Mach-O File Format

All iOS binaries use the Mach-O file format[4]. It supports multi-architecture binaries, which means that a single binary file can contain multiple Mach-O files for different CPU architectures. In this work we will always use only the Mach-O file for the A64 instruction set.

A Mach-O file is divided into three main regions:

- **Header**: Identifies the file as a Mach-O file and includes the information about the target architecture.
- **Load Commands**: Specifies the layout of the file and where the different segments are located in memory when the binary is executed.
- **Data**: The data region has again various segments that have different sections, which are loaded into memory. This region also contains the *Dynamic Loader info*, which defines where external symbols, which are loaded dynamically, have to be stored when the application is executed.

For this work the *Load Commands* section is important to locate the data that is needed in the *Data* region. The *Data* region itself contains all data that is needed during execution including the machine

[4]`https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime`

instructions. Having the machine instructions only is not sufficient to decompile the binary for doing static program analysis.

3.3.1 Objective-C Class Information

When dealing with method calls in Objective-C it is crucial to know the type of the object and the name of the method that gets called. In Objective-C methods are never called directly using the address of the method in the binary, but they are called using the Objective-C runtime library and this library decides based on the information about the type of the object and the method name, which method actually gets called.

The sections that are relevant for this are:

- **__objc_classlist:** Is a list of pointers to all class descriptions of those classes that are defined in this binary. The pointers point to addresses of the section **__objc_data**.
- **__objc_data:** Contains a pointers to the superclass and a pointer to **__objc_const** for retrieving class infos.
- **__objc_const:** Has the details for every class in the binary out the implemented methods (their name and the location in the binary) as well as the details about implemented protocols, instance variables and defined properties.
- **Dynamic Loader Info:** The pointers in the entries of **__objc_data** to the super class are not set initially, if the class is not defined in this binary (for example classes of Objective-C runtime library). Using this section the actual values for those pointers can be determined and the full class hierarchy can be reconstructed.

3.3.2 External Functions

iOS applications will always contain calls to functions that are not defined inside the applications binary, but in some library that gets loaded when the application is executed. For a successful analysis these calls have to be detected and handled correctly since they may create or modify data used in the application. These calls are made using *indirect symbols*. However, the information that is needed to resolve these symbols is found in the binary. They can be either *non-lazy* symbols, which are resolved when the binary is loaded using the binding info stored in the *Dynamic Loader Info*, or *lazy* symbols that are resolved when they are first used using *stubs*.

The *non-lazy* symbols are resolved by executing commands of stored in *Binding Info* of the *Dynamic Loader Info* section in the binary. These commands calculate the address where the pointer should be stored and using a name that is also defined there, the symbol gets resolved. An example for a non-lazy symbol resolution is shown in Listing 3.5. There may be some additional commands, but the ones in this example have to be always present. The important values are defined in line 2 that sets the name of the symbol and using line 4 the address of the symbol can be retrieved.

```
1  BIND_OPCODE_SET_SYMBOL_TRAILING_FLAGS_IMM
2  string
3  BIND_OPCODE_ADD_ADDR_ULEB
4  uleb128
5  BIND_OPCODE_DO_BIND
```

Listing 3.5: Non-Lazy Symbol Resolution

Lazy symbol resolution is shown in Listing 3.6. When a call to a lazy symbol is executed the branch instruction redirects the execution to the function *stub* where the address of the actual function should be loaded from a *lazy_ptr_address*. If the symbol resolution has already been done this pointer holds the address of this function. However, if this was not done this pointer contains a *stub_helper_address* that tells the runtime environment, which symbol has to be resolved. The instructions at this address loads an ID to a register (in Listing 3.6 line 14 loads the ID stored at line 17) that can be matched to an entry in the symbol table where the name of this symbol can be retrieved.

```
1       ...
2       BL      stub_address
3       ...
4
5   stub_address:
6       NOP
7       LDR     X16, lazy_ptr_address
8       BR      X16
9
10  lazy_ptr_address:
11      stub_helper_address
12
13  stub_helper_address:
14      LDR    W16, stub_id
15      B      stub_helper
16  stub_id:
17      .long ID
18
19  stub_helper:
20      ;call to dyld_stub_binder
```

Listing 3.6: Lazy Symbol Resolution

3.3.3 Method Signatures and Protocol Definitions

The Mach-O file includes method signature definitions for all defined Objective-C methods. While for arbitrary methods the type definitions are stored in a "compressed" way where all Objective-C objects are represented by the universal identifier "@" the method signatures for protocol methods are stored with their complete signature. Having this information for protocol methods makes the call graph generation more precise since the parameters, which are passed to these methods, are probably not allocated anywhere in the code of the binary but in other libraries (for example UI protocol methods are called from the UIKit framework). Without these definitions there would be no possibility of getting the types of such parameters, if they are allocated externally and thus the call graph generation would lack of calls made to these objects.

3.3.4 Instance Variables

Instance variables are annotated with their type in the Mach-O binary similar to methods signatures. In most cases instance variables that store objects are allocated somewhere in the code of the binary. However, this does not hold for all instance variables. For example take UI elements. They can be created in two different ways: either directly in the code, which then leads to knowing their types, or using the *Interface Builder* that creates the UI based on information stored in files. The latter is handled by an external framework and thus the allocation of the objects happens there as well. For such objects the type information would be incomplete without having the information from the binary.

3.4 Recovering Lost Information

Event though information gets lost when compiling a binary the remaining information is sufficient to decompile the binary for static analysis purposes. Some of the previously lost data can be reconstructed using the data stored in the binary and the machine instructions of the binary. It may be possible to completely restore some parts, but in for others assumptions have to be made.

3.4.1 Intraprocedural Control Flow

Intraprocedural control flow describes the possible execution paths a program can take inside a single function. These paths are influenced by control flow statements (e.g. branch instructions) that do not leave the scope of the current function contrary to function call instructions, which will leave this scope.

Instructions can be grouped into so called *basic blocks*. These basic blocks contain instructions with a single entry and a single exit point, which means that either all instructions of this basic block are executed or none. Branch instructions always define an exit instruction of a basic block since they have multiple successor instructions. To identify all exit and entry instruction the branch instructions need to be analyzed. Branch instructions define the succeeding instructions that are executed after the branch instruction (if multiple successors are defined the decision, which successor is actually executed is made during runtime based on a condition, which is also defined by the branch instruction). These successors are described by the address of the instruction in the binary and using this address the entry points of basic blocks are found. The immediate predecessors of entry points can now be set as exit instructions of basic blocks, no matter if this is a branch instruction or not.

Basic blocks can now be used to describe the control flow inside a single function. They are represented as nodes in the control flow graph and the edges between those nodes describe the successors.

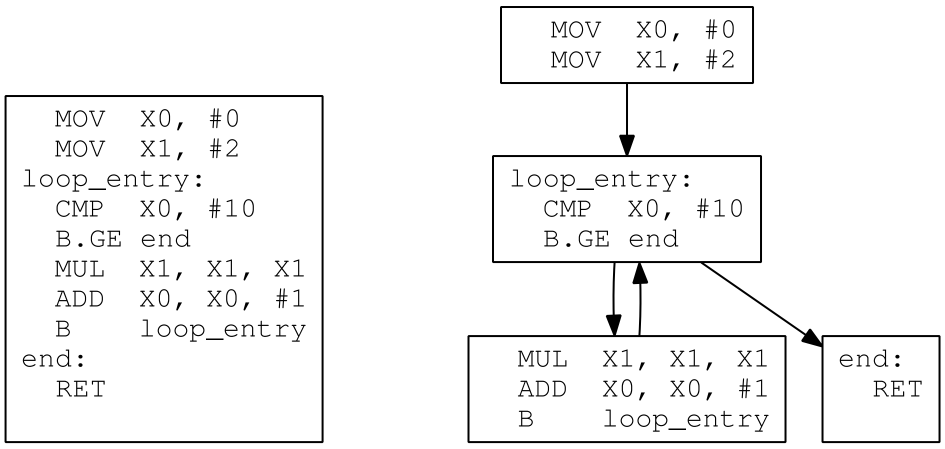
If a basic block contains a set of machine instructions it also contains the corresponding decompiled instructions since the decompiled code has to perform like the machine instructions. This means that the decompiled code has exactly the same control flow graph as the original binary even though the size of the basic blocks will differ since a single machine instruction may be represented with multiple instructions in intermediate representation. However, the crucial part is that we are able to construct a control flow graph during decompilation by analyzing the branch instruction of the binary.

The example in Figure 3.4 visualizes how grouping instructions into basic blocks helps understanding the control flow inside a function. Having only the machine instructions the successors and predecessors are not always clearly visible.

3.4.2 Function Parameters and Return Values

This task is more difficult (and less precise) than the previously described control flow graph generation. Function parameters are defined by function headers in the source program. However, these definitions are completely removed during compilation and only assumptions can be made. These assumptions are made based on the calling conventions defined by the CPU architecture [3] that describe where parameters and return values have to be stored. A function parameter is assumed when a value is read from a location where a parameter may be stored before it was written there. To identify these locations the CFG is traversed from the entry point to this load instruction and a parameter is assumed, if no instruction is found that stores something to this location. The condition for identifying return values is similar to the one of function parameter. The difference is that the CFG is not traversed from the function entry, but from the call instruction (to which will be returned after executing the function).

The identified parameters and return values are later used when it comes to data flow analysis (static program slicing in Chapter 4, which includes pointer analysis).



```
  MOV  X0, #0
  MOV  X1, #2
loop_entry:
  CMP  X0, #10
  B.GE end
  MUL  X1, X1, X1
  ADD  X0, X0, #1
  B    loop_entry
end:
  RET
```

```
  MOV  X0, #0
  MOV  X1, #2
```

```
loop_entry:
  CMP  X0, #10
  B.GE end
```

```
  MUL  X1, X1, X1
  ADD  X0, X0, #1
  B    loop_entry
```

```
end:
  RET
```

Figure 3.4: The control flow graph is generated from the code on the left that executes 10 iterations of a loop, which squares the value stored in the register X1.

3.5 External Symbols

These symbols represent values that are not defined inside the given binary, but in an external library that is loaded when the application is executed. To resolve these values we have to mock the behavior during application execution when the placeholders are set to the correct values. The information about the used external symbols is always present in the binary since it is needed for executing an application. This means that, unlike some other information that can only be assumed, this information is completely reproducible. The information about symbols consists of the symbol name and the library where the symbol is defined and this already sufficient for the purpose of static analysis and the placeholder value gets replaced by this information in the decompiled code.

Chapter 4

Static Slicing

Static slicing is the process of creating a subset of statements of a program that might influence a value at a specified point. The algorithm presented is based on Weisers [26] method of program slicing. With the modifications made it is possible to create program slices of binaries that were decompiled to LLVM IR as described in Chapter 3.

The goal of static slicing in this work is to identify parts of a program that influence a specified parameter, the slicing criterion, and further to use this information to find paths from the origin of a parameter to its use in a cryptographic function.

The slicing is first described as an intraprocedural method using dataflow equations in Section 4.1.0.3 and later this approach is extended to deal with function calls in Section 4.1.1. To be able to create slices when pointers are used a precise knowledge of pointer states is needed. This information is delivered by a supporting pointer analysis described in Section 4.2.

4.1 Computing Slices

Before being able to generate slices using a slicing criterion the *DEF* and *REF* sets need to be generated for each statement in the program. These sets have to be generated only once since they do not change during program slicing.

4.1.0.1 Modified Variables

$DEF(n)$ contains all variables $v \in V$ that are modified at statement n. This does not only include variables that get assigned at a statement n like in statements in the form of "$v = RHS$" where the variable v is obviously defined, but also also statements that write a value to a memory location. In LLVM IR this is solely done by `store` statements.

```
store v, ptr
```

writes a value v to a location l that is pointed to by ptr ($l \in pts(ptr)$). Consider that ptr may point to multiple locations. This means that such statements are able to modify multiple variables, even though a single execution of the statement only modifies the location l that is pointed to by ptr at the time of executing the statement.

$$DEF(n) = \begin{cases} v & \text{, if } n \text{ is an assignment: } v = RHS \\ \{l \mid l \in pts(ptr)\} & \text{, if } n \text{ is a } \texttt{store} \text{ instruction: } \texttt{store}\ v,\ ptr \end{cases} \tag{4.1}$$

4.1.0.2 Referenced Variables

$REF(n)$ is the set of variables that are referenced at statement n. For most statements the REF set is easily determinable by including all variables of the statement except the left-hand side of an assignment (if the variable is present in both the left-hand side and the right-hand side it is included in $REF(n)$ and $DEF(n)$). However, it is also possible to read from a memory location l using the `load` instruction. In this case the location l has to be included in $REF(n)$ as well.

$$REF(n) = \begin{cases} ptr \cup \{l \mid l \in pts(ptr)\} & \text{, if } n \text{ is a } \texttt{load} \text{ instruction: } v = \texttt{load}\, ptr \\ v \cup ptr & \text{, if } n \text{ is a } \texttt{store} \text{ instruction: } \texttt{store}\, v,\ ptr \\ v \in RHS & \text{, else } n \text{ is an assignment: } v = RHS \end{cases} \tag{4.2}$$

4.1.0.3 Dataflow Equations

The dataflow equations in Equation 4.3 and 4.4 were defined by Weiser [26] and they are used to compute relevant variables and statements for a slicing criterion $C = (n, V)$ where n represents the statement where the slicing algorithm starts and V is a set of variables for which the possible influences should be found.

$$\begin{aligned} R^0_C(i) &= \{v \mid v \in V \wedge i = n\} \cup \left\{v \mid v \in R^0_C(j), v \notin DEF(i)\right\} \cup \left\{v \mid v \in REF(i),\ DEF(i) \cap R^0_C(j) \neq \emptyset\right\} \\ S^0_C(i) &= \left\{i \mid (DEF(i) \cap R^0_C(j)) \neq \emptyset\right\} \end{aligned} \tag{4.3}$$

The equations above represent the directly relevant variables R^0_C and the directly relevant statements S^0_C. Direct relevance is denoted by the superscript 0 and the higher this superscript is the more indirectly relevant variables and statements are and they are defined as followed:

$$\begin{aligned} B^k_C &= \left\{b \mid \exists\, i \in S^k_C,\ i \in INFL(b)\right\} \\ R^{k+1}_C(i) &= R^k_C \cup \textstyle\bigcup_{b \in B^k_C} R^0_{(b,REF(b))}(i) \\ S^{k+1}_C(i) &= B^k_C \cup \left\{i \mid DEF(i) \cap R^{k+1}_C(j) \neq \emptyset\right\} \end{aligned} \tag{4.4}$$

The algorithm of Weiser computes relevant variables and relevant statements in an iterative fashion. The solution, namely the slice S, is the fixpoint of the set of relevant statements. As described by Tip [23] the algorithm of Weiser consists of two stages:

1. **Follow data dependencies:** this step is executed iteratively, if control dependencies were found.
2. **Follow control dependencies:** includes the relevant variables of control flow statements and step 1 is executed again for such variables.

As an example the algorithm is described based on the example in Listing 4.1 computed intermediate results shown in Table 4.1. First the sets of directly relevant variables R^0_C are computed and using those the set of directly relevant statements $S^0_C = \{1, 3, 5\}$ is generated. For the next iteration the set of statements with direct influence on S^0_C is needed, which is $B^0_C = \{4\}$. The statements of this set have to be considered when computing the indirectly relevant variables (in this case R^1_C). Now it is possible to determine the relevant statements $S^1_C = \{1, 2, 3, 4, 5\}$, which is already the solution and further iterations would not change anything (there is only one statement left that is not included in the slice). Note that without having the points-to information of $px \rightarrow x$ only the statement at line 1 would be in the slice, which means that having points-to information is crucial.

```
int x = 0;
int y;
int *px = &x;
if (y) {
  *px = 123;
NSLog(@"%d", x);
```

Listing 4.1: Example to show the computation of relevant variables and statements.

Line	DEF	REF	$INFL$	pts	R^0_C	$R^0_{(4,\ y)}$	R^1_C
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{x\}$	$\emptyset$	$\{x\}$
3	$\{px\}$	$\{x\}$	$\emptyset$	$\{x\}$	$\{x\}$	$\{y\}$	$\{x,\ y\}$
4	$\emptyset$	$\{y\}$	$\{5\}$	$\emptyset$	$\{x,\ px\}$	$\{y\}$	$\{x,\ y,\ px\}$
5	$\{x\}$	$\{px\}$	$\emptyset$	$\emptyset$	$\{px\}$	$\emptyset$	$\{px\}$
6	$\emptyset$	$\{x\}$	$\emptyset$	$\emptyset$	$\{x\}$	$\emptyset$	$\{x\}$

Table 4.1: Results of the relevant variables using the criterion $C = (6,\ x)$ for the example in Listing 4.1. Using R^0_C the relevant statements $S^0_C = \{1, 3, 5\}$ can be computed.

4.1.1 Interprocedural Slicing

The previously defined method is only capable of intraprocedural slicing. To be able to create slices that include function calls two steps are necessary: first an intraprocedural slice of the function P is computed using the method described earlier and after that a set of new slicing criterions is generated for every function that calls P or is called by P. In Weisers work the generation of new criterions is described as $DOWN(C)$ for the callers of P and $UP(C)$ for functions that are called by P based on the slicing criterion C. His method substitutes formal parameters with actual ones. This approach will not work using decompiled code due to the way how parameters are passed and values are returned. Every function has exactly one formal parameter that represents the register set and this register set is the same for all functions and that means no parameter can be substituted. The actual parameter passing is done with the registers of this register set by following the specifications of the target CPU. In the case of ARMv8 [3] both registers and the stack are used to pass parameters.

Stack parameters do not need to be handled because they are sliced using the points-to information and if a stack parameter is a relevant variable, the set of relevant variables R_C holds the variable representing the location of this parameter and this location is the same in both functions.

The method described below explains how new slicing criterions can be generated when decompiled code is used. Equation 4.5 defines how a new criterion is created when a function P calls a function Q at statement i. $ROUT(i)$ collects all relevant variables of the statements that can be executed after the call at statement i. n^Q_r is the return statement of the function Q for which the new slicing criterion is generated. The subscript $F \to A$ denotes the translation of formal parameters to actual parameters, but in this work parameters are passed using the register set and how this works is described later in this section.

$$(n^Q_r, ROUT(i)_{F\to A} \cap SCOPE_Q) \tag{4.5}$$

$$ROUT(i) = \bigcup_{j\in Successor(i)} R_C(j) \tag{4.6}$$

The following equation describes how a new criterion is created when a function R calls a function P at statement i where n_e^P is the entry statement of a function P. Again, the parameters passed to P need to be substituted by the variables available variables in R.

$$(i, R_C(n_e^P)_{A\to F} \cap SCOPE_R) \tag{4.7}$$

The substitution of parameters is used in both of the cases above. There are two directions such a substitution can be done. First, from formal to actual parameters $F \to A$ that is required in Equation 4.5 when a function P calls an other function Q and the formal parameters that are used in Q need to be mapped to those accessed by P.

$S(v, r)$ is a `store` instruction that stores a variable v to the register r and $L(v, r)$ is a `load` instruction that loads a value from the register r to the variable v. $STORE(i)$ returns the set of the last variables that were stored to a register before the instruction i and $LOAD(i)$ returns the set of the first variables, which were loaded from a register after the instruction i defined as followed:

$$STORE(i) = \left\{(v,\ r) \mid \exists\, S_i(v,\ r) \to^*_{CFG} i,\ \nexists\, L_i(v,\ r) \to^*_{CFG} S_j(v',\ r) \to^*_{CFG}\ i\right\} \tag{4.8}$$

$$LOAD(i) = \left\{(v,\ r) \mid \exists\, i \to^*_{CFG} L_i(v,\ r),\ \nexists\, i \to^*_{CFG} L_j(v',\ r) \to^*_{CFG} L_i(v,\ r)\right\} \tag{4.9}$$

Using these sets it is possible to define the substitution rule for both cases. $ROUT(I)_{F\to A}$ substitutes the variables of the relevant set after the call i with those used in the function Q. This is needed to be able to slice returned values defined in the function Q and later accessed in function P.

$$\begin{aligned} ROUT(i)_{F\to A} = \ & \{v \mid (v' \in ROUT(i),\ \exists(v, r) \in STORE(n_r^Q),\ \exists(v', r) \in LOAD(i))\ \textbf{xor} \\ & v \in ROUT(i)\} \end{aligned} \tag{4.10}$$

$R_C(n_e^P)_{A\to F}$ replaces the variables, that are in the relevant set at the function entry n_e^P of P, with those that were passed to P by R. This is simply done by taking the last value that was stored to a register in R that is later used in P.

$$\begin{aligned} R_C(n_e^P)_{A\to F} = \ & \{v \mid (v' \in R_C(n_e^P) \exists\, (v, r) \in LOAD(n_e^P),\ \exists\, (v', r) \in STORE(i))\ \textbf{xor} \\ & v \in R_C(n_e^P)\} \end{aligned} \tag{4.11}$$

Both of the equations above preserve the relevant values, which can not be substituted like the locations of stack parameters with the result that they are included in the new slicing criterion, if they are possibly modified.

4.2 Pointer Analysis

Programs written in languages that support pointers can be difficult to analyze. To be able to perform a static analysis on such programs, knowledge of pointer states is essential. Not knowing anything about the pointers used in the program will lead to wrong results during static analysis. The following example shows why static slicing does not deliver correct results without having points-to information:

```
int x = 1;
int *px1 = &x;
int *px2 = &x;
*px2 = 2;
int y = *px1;
```

Listing 4.2: Example code for showing the effects of a pointer analysis

If we want to create a slice of the variable `y` at the end of the example the statement `*px2 = 2` would not be included in the slice, but `*p2x` and `*px1` point to the same location, which means changing the value stored at this location affects both variables.

In this section a pointer analysis is described that is capable of computing points-to sets of binaries that were decompiled to LLVM IR. It is a flow- and context-insensitive analysis (the differences to flow- and context- sensitive methods are described in Section 4.2.1.2 and Section 4.2.1.1) that is able to be scaled to programs with millions lines of code. The language that was used to create the binary of which the pointer analysis has to be done might introduce additional challenges. The challenges in a pointer analysis for the widely used language Objective-C are discussed in Section 4.2.6.

4.2.1 Inclusion Based Pointer Analysis

This type of pointer analysis is also known as *Andersen style* pointer analysis as it was defined by Andersen [2]. The pointer analysis is formulated as a set-constraint problem where a constraint system C is created for a given program and by solving the constraint system it is possible to determine the locations a variable might point to during execution of the program. This constraint system only consists of constraints of the type $a \supseteq b$, which means that information flows uni-directional from b to a. There are only four different types of constraints that may be added to C (see Table 4.2) and they are added by identifying code patterns as described in Section 4.2.2.1.

Andersen has defined a context sensitive and a context insensitive version of his algorithm (even though they are called inter-procedural and intra-procedural in his work). However, in the scope of this work only the context insensitive method is used for computing points-to sets

4.2.1.1 Context Sensitivity

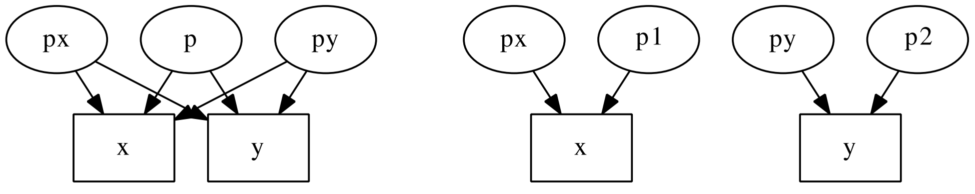
A context sensitive pointer analysis separates information that originates from different paths. The example in Listing 4.3 and the results in Figure 4.1 show how context sensitive pointer analysis works compared to a context insensitive one.

However, the possibility of getting more precise results in a more expensive analysis. The time complexity for such an analysis is exponential as shown by Wilson [27] and Emami et al. [8]. For this reason context sensitive pointer analysis is not very scalable for larger programs that may have a high number of calling contexts.

```
-(void) func {
  int x, y;
  int *px, *py;
  px = [self copyPointer: &x]₁;
  py = [self copyPointer: &y]₂;
}

-(int*)copyPointer:(int*)p {
  return p;
}
```

Listing 4.3: Simple example to show the effects of context sensitivity by assigning the return value of a function that is called twice with different parameters to different variables.



(a) Context insensitive **(b)** Context sensitive

Figure 4.1: The results for both a context insensitive and a context sensitive pointer analysis of the sample program shown in Listing 4.3 are shown in this figure. In Figure 4.1a all pointers (`px, py` and `p`) point to both `x` and `y`. The result of a context sensitive pointer analysis is shown in Figure 4.1b where p_1 and p_2 refer to the first and the second call in Listing 4.3.

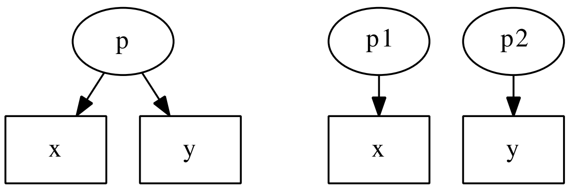
4.2.1.2 Flow Sensitivity

Flow sensitive pointer analysis takes the control flow of an program into account whereas flow insensitive pointer analysis ignores the control flow of a program. The flow sensitive analysis results in an individual points-to set for every instruction since the same variable might have different points-to sets, if it gets assigned different values at statements that are reachable via different paths while the flow insensitive analysis collects all information in a single points-to set (a “summary” of the whole program). The results of the example in Listing 4.4 displayed in Figure 4.2 show that for the flow insensitive case the pointer `p` may point to both `x` and `y` even though at the assignment instruction p_1 the variable `y` has never been assigned yet. A flow sensitive distinguishes these assignments and creates a points-to set for each assignment.

Hind and Pioli [13] have shown that a flow sensitive pointer analysis does not improve precision of the results, if a context insensitive algorithm is used, and further flow sensitive pointer analysis did not scale well for larger programs.

```
-(void) func {
  int x, y;
  int *p;
  p₁ = &x;
  p₂ = &y;
}
```

Listing 4.4: Example code to show the differences between flow-sensitive and flow-insensitive pointer analysis. The results are described in Figure 4.2.



(a) Flow insensitive **(b)** Flow sensitive

Figure 4.2: The results of a flow insensitive pointer analysis (Figure 4.2a) show that a single pointer that was assigned with different locations points to all locations while a flow sensitive distinguishes the assignments (Figure 4.2b) and creates an own points-to set for all statements (indicated by the subscript value).

4.2.2 Constraint Generation

The constraint generation defined in Andersens pointer analysis [2] covers only the C language and misses an important pattern that occurs in the decompiled code: all memory accesses are done by converting an integer to a pointer variable. This is done as a result of having only the target CPUs register set for storing values. In this register set everything is stored as an integer value and gets converted to other types, if needed. In [2] such integer to pointer casts will point to the abstract location *Unknown* and this leads to an unsuccessful analysis. However, in this work all memory access operations have a predecessor that converts a integer to a pointer variable (an *inttoptr* instruction) and an unmodified version of Andersens constraint generation method would produce a points-to set where all pointers point to *Unknown*.

The input for the `inttoptr` instruction is one of these three sources:

- **Binary:** has a constant integer as source operand for the `inttoptr` instruction
- **Heap:** requires a value that was returned by some memory allocation functions
- **Stack:** adds a constant integer to a value that references the stack pointer

Whenever data, that is stored in the binary, is accessed a statically defined address is converted to a pointer and using this pointer the data is accessible. This type of memory access is the simplest case to handle because the operand for the `inttoptr` instruction is already the address as an integer value. Heap addresses are always created by calling memory allocation function (e.g. malloc) and the return value describes an abstract location that is used for the pointer analysis. Stack accessed memory is usually by adding a static offset to the the stack pointer value (or frame pointer value), this value is converted to a pointer and is then used by a memory access instruction. If the offset is zero, the stack pointer value gets converted directly without having a binary instruction. However, different instructions that access the same location on the stack are not easily identifiable because the pointer used in each instruction is created separately for these instructions. The following example in assembler code contains only a store instruction followed by a load instruction for the same location on the stack:

```
STR  X0, [SP, #8]
LDR  X0, [SP, #8]
```

Listing 4.5: ASM example for pointer analysis of a stack variable

Is decompiled to:

```
%1 = add i64 SP_init, 8
%2 = inttoptr i64 %1 to i64*
store i64 X0_0, i64* %2
%3 = add i64 SP_init, 8
%4 = inttoptr i64 %3 to i64*
X0_1 = load i64, i64* %4
```

Listing 4.6: Decompiled code of the example in Listing 4.5

The pointers (%2 and %4) obviously point to the same location, but they are created from different variables that contain the same value. The essential task is to find the variables that refer to the stack and might contain the same value during execution. How these matching memory accesses are combined is discussed by describing the way stack parameters are identified since this is done in the same way (see Section 4.2.3.2).

Constraint type	Meaning
$v_i \supseteq \{v_j\}$	$loc(v_j) \in pts(v_i)$
$v_i \supseteq v_j$	$pts(v_i) \supseteq pts(v_j)$
$v_i \supseteq *v_j$	$\forall v \in pts(v_j) : pts(v_i) \supseteq pts(v)$
$*v_i \supseteq v_j$	$\forall v \in pts(v_i) : pts(v) \supseteq pts(v_j)$

Table 4.2: Constraints and their meaning

4.2.2.1 Pattern Descriptions

Constraint generation is done by identifying different patterns that occur in the decompiled code. Table 4.2 describes the constraint types that are used during constraint generation to generate the points to information. The constraints described there are based on those of Hardekopf and Lin [10].

Base pointer with constant offset: This pattern occurs when a value stored in a register is used as base pointer and an offset is added to this base pointer (for example accessing a local variable that is stored on the stack is handled by this pattern).

```
v_BaseRegPtr = getelementptr
v_Base       = load v_BaseRegPtr
v_Addr       = binop v_Base, ConstInt
v_Ptr        = inttoptr v_Addr
```

Constraint to add:

$$\begin{gathered} v_{\mathtt{Ptr}} \supseteq v_{\mathtt{Addr}} \\ loc(v_{\mathtt{Addr}}) = \emptyset : v_{\mathtt{Addr}} \supseteq \{v_{\mathtt{Addr}}\} \end{gathered} \tag{4.12}$$

If ConstInt is zero $v_{\mathtt{addr}}$ is equal to $v_{\mathtt{base}}$ and $v_{\mathtt{base}}$ might be used directly in the inttoptr instruction. However, in both cases the same type of constraint will be created.

Base pointer with variable offset: Variable offset means that the offset used in the binary operation is not a constant integer value but a variable value stored in a register.

```
v_BaseRegPtr = getelementptr
v_Base       = load v_BaseRegPtr
v_BaseAddr   = binop v_Base, ConstInt
v_Addr       = binop v_BaseAddr, v_Offset; {v_Offset ∉ ConstInt}
v_Ptr        = inttoptr v_Addr
```

Constraint to add:

$$\forall v \in loc(v_{\mathtt{Base}} + c) \neq \emptyset : v_{\mathtt{Base}} \supseteq v \qquad , c \in \mathbb{Z} \tag{4.13}$$

This type of memory access is harder to process than the previous one with constant offset because in theory the pointer might point anywhere. However, using the base pointers address the points-to information can be conservatively approximated. It is assumed that anything that anything that uses this particular base pointer for accessing memory is included in the points-to set of $v_{\mathtt{Addr}}$.

Loading from a constant address: This happens when something is read from the binary. Getting references to Objective-C related information is covered by this as well as accessing global variables.

```
v = load (inttoptr c) ; c ∈ℤ
```

Constraints to add:

$$\begin{aligned} \mathtt{v} &\supseteq *\mathtt{c} \\ \mathtt{c} &\supseteq \{\mathtt{c}\} \end{aligned} \tag{4.14}$$

Before adding the constraints of this instruction the value of `c` is verified to be an address located in the binary of this program since only those values are valid.

Loading from a non-constant address: All other `load` instructions are handled by this pattern. The difference to the case of constant addresses is that abstract locations have already been handled by the `inttoptr` instruction.

```
vPtr = inttoptr vAddr ; vAddr ∉ ℤ
v = load vPtr
```

Constraint to add:

$$\mathtt{v} \supseteq *\mathtt{v_{addr}} \tag{4.15}$$

Storing a value to a constant address: Similar to loading a value from a constant address the addresses should be contained in the binary of the program.

```
store v, (inttoptr c); c∈ℤ
```

Constraints to add:

$$\begin{aligned} *\mathtt{v} &\supseteq \mathtt{c} \\ \mathtt{c} &\supseteq \{\mathtt{c}\} \end{aligned} \tag{4.16}$$

Storing a value to a non-constant address: like in the case of loading from a non-constant address the difference to the case of storing a value to constant address is that the abstract locations have already been handled.

```
vPtr = inttoptr vAddr ; vAddr ∉ ℤ
store v, vPtr
```

Constraint to add:

$$*\mathtt{v} \supseteq \mathtt{v_{addr}} \tag{4.17}$$

4.2.3 Constraints for Function Calls

At every call instruction the points-to information needs to be propagated from the callee to the called function. As described in Section 3.2.3.2 all parameters are passed using the only actual parameter that represents the register set and the called function reads the parameters from this register set. Parameters are passed to functions either using registers or the stack (the same holds for return values). The target CPUs specification [3] defines when to use a register (and which register) or when to use the stack and where the parameters are stored on the stack when function calls are made.

4.2.3.1 Register Parameters

s_i and s_j represent `store` instructions in the calling function that store arbitrary values v_i and v_j to the same register in the register set. The same holds for the load instructions l_i and l_j, which load values v'_i and v'_j in the called function. i_{call} is the `call` instruction and i_{entry} is the function entry of the called function. For a proper handling of register parameters the last value in the calling function has to be copied to the first used value of the same register in the called function:

$$\begin{gathered}\exists\ s_i \rightarrow^*_{CFG} i_{call},\ \exists\ i_{entry} \rightarrow^*_{CFG} l_i, \\ \nexists\ s_i \rightarrow^*_{CFG} s_j \rightarrow^*_{CFG} i_{call},\ \nexists\ i_{entry} \rightarrow^*_{CFG} l_j \rightarrow^*_{CFG} l_i :\ v'_i \supseteq v_i\end{gathered} \tag{4.18}$$

4.2.3.2 Stack Parameters

Whenever a parameter is stored on the stack it can not be directly accessed. They are accessed using the stack pointer as a base address to which an offset gets added and the result represents the memory location, which is converted to a pointer. Using `load` and `store` instruction this parameter can be accessed and modified.

Two accesses with values that use the stack pointer as base address are matched together, if the added offsets are equal. Algorithm 1 shows how the offset is computed for values that reference the stack pointer. The value passed to the algorithm for the parameter I_{Load} is the value of the stack pointer loaded from the register set. As a result the algorithm return all value-offset pairs that might be used as addresses referencing the stack. It simply takes the stack pointer value and finds all operations that use the stack pointer in an `add` or `sub` operation with a constant integer. Such operations are used to create the stack address that is later used in an `inttoptr` instruction for creating a pointer. This means that the result is a set of all possible stack access addresses with the relative offsets to the stack pointer and finding the stack parameters is quite simple using this set. In the ARMv8 procedure call standard [3] it is defined that the stack parameters have an offset of ≥ 0 relative to the stack pointer value at the function entry and all local variables have offsets < 0. This set of stack accesses is calculated for both the calling function and the called function, but since the stack pointer may be modified between entering the calling function and the function call statement (the stack has to be increased to hold at least the parameters) the values of the two sets will not match. To be able to create matches between those two sets the offset of the stack pointer between the function entry and the call statement is needed (the stack size at the call statement) and this value is added to the offsets of the stack parameters resulting in offset values that have to be found in the stack access set of the calling function.

What Algorithm 1 is not able to handle are iterative incrementations or decrementations of a stack referencing value (for example the stack pointer gets incremented by a constant value in every iteration of a loop without being decremented by the same value before the next iteration). However such behavior is not expected anyway for accessing the stack since stack stored variables are accessed using a constant offset and by iteratively changing the stack pointer this would not be possible. Nevertheless, such constructs may be present in the code, even if they are not used for accessing the stack, but this does not get verified. The algorithm simply does not continue processing a value that has already been handled.

Having the value and offset pairs delivered by Algorithm 1 stored in $Results_{Caller}$ for the calling function and in $Results_{Callee}$ for the called function it is possible to add constraints that fulfill the following condition:

$$(v_i, Offset_i) \in Results,\ (v_j, Offset_j) \in Results,\ Offset_i = Offset_j,\ v_j \supseteq \{l\} \in C :\ v_i \supseteq v_j \tag{4.19}$$

By adding this constraint for both the calling and the called function all values that might be used in a memory access instruction it is ensured that they point to the same abstract location $\{l\}$, but for now only the constraints are only generated for the two functions and the parameters are not matched yet. To achieve this the stack size at the `call` instruction has to be determined. This is done in a similar way as showed in Algorithm 1 by evaluating `add` and `sub` instructions until the `call` instruction is reached. This value is represented by the value *StackSize*. To match the parameters the following condition has to hold, that detects accesses to the same location on the stack, to add new constraints:

$$(v_i, Offset_i) \in Results_{Caller},\ (v_j, Offset_j) \in Results_{Callee},\ Offset_i + StackSize = Offset_j,\ v_i \supseteq \{l\} \in C :\\ v_j \supseteq v_i \tag{4.20}$$

Algorithm 1 Computing offsets for base pointers

```
function getOffsets($I_{Load}$, $InitialOffset$, $Results$)
    $CurrentOffset \leftarrow InitialOffset$
    for all $I \in$ uses($I_{Load}$) do                ▷ uses returns all instruction where $I_{Load}$ is used
        if $I \in Results$ then                        ▷ Prevents an endless loop
            continue
        end if
        if $I \in$ store then
            $\mathcal{I}_{NextLoad} \leftarrow$ findNextLoads($I$)
            for all $I_{NextLoad} \in \mathcal{I}_{NextLoad}$ do
                getOffsets($I_{NextLoad}$, $CurrentOffset$, $Results$)
            end for
        end if
        if isAddInstruction($I$) and hasConstantOperand($I$) then
            $C \leftarrow$ getConstantOperand(I)
            $CurrentOffset \leftarrow CurrentOffset + C$
        end if
        if isSubInstruction($I$) and hasConstantOperand($I$) then
            $C \leftarrow$ getConstantOperand($I$)
            $CurrentOffset \leftarrow CurrentOffset - C$
        end if
        AddResult($Results$, $I$, $CurrentOffset$)  ▷ Adds a tuple with the value and the offset as constant
integer to the result set
    end for
end function
```

4.2.3.3 Return Value

This works similar like passing parameters using registers, but in this case the `store` instruction s_i and s_j (storing the values v_i and v_j) are defined in the called function and the `load` instruction l_i and l_j (loading the values v'_i and v'_j) are defined in the calling function. The called function has only a single exit point (the Instruction i_{return}) and i_{call} is again the `call` instruction. The constraint generation is done the other way round as in passing parameters using registers. Now the last value in the called function is copied to the first value used after the call for the same register:

$$\exists\ s_i \rightarrow^*_{CFG} i_{return},\ \exists\ i_{call} \rightarrow^*_{CFG} l_i,\\ \nexists\ s_i \rightarrow^*_{CFG} s_j \rightarrow^*_{CFG} i_{return}, \nexists\ i_{call} \rightarrow^*_{CFG} l_j \rightarrow^*_{CFG} l_i :\ v'_i \supseteq v_i \tag{4.21}$$

4.2.4 Iterative Constraint Updates

The algorithm presented by Andersen [2] generates constraints by a single pass over every statement in the program. This approach needs to be modified to fit the needs of the programs used in this work. By having uncertain knowledge about which functions get called by call instructions when it comes to Objective-C calls (described in Section 4.2.6) or function pointers such call instructions need to be handled multiple times because points-to knowledge is already needed before being able to process these calls.

Algorithm 2 describes how constraint generation is done in this work. The first loop iterates over all instructions in the given program. This is also done in [2], but Andersen did not differentiate between call instructions and other instructions and the task of constraint generation was finished after this loop. However in this work the call instruction are handled separately after this pass over all instructions. The next loop is executed until no new constraints are added and this means that no new edges are added to the call graph since the constraints that are generated at call instruction handle parameter passing and return values only. Constraint solving is executed in every iteration of this loop for propagating points-to information through the program to be able to update the call graph as described later in Algorithm 3.

Algorithm 2 Constraint generation

```
Require: Program $\mathcal{P}$
  for all Instruction $\mathcal{I} \in \mathcal{P}$ do
    if isCallInstruction($\mathcal{I}$) then
      AddInstruction($\mathcal{I}$, $CallInstructions$)
    else
      generateConstraints($\mathcal{I}$)
    end if
  end for
  repeat
    solveConstraints
    for all Instruction $\mathcal{I} \in CallInstructions$ do
      updateConstraints($\mathcal{I}$)
    end for
  until no new constraints added
```

4.2.5 Constraint Solving

The points to set of a variable v is defined as $\mathcal{S}_v = \{v_l : v \supseteq \{v_l\}\}$. It contains all abstract locations the variable might point to during execution of the program. Solving the constraint system C is done by rewriting the constraints described in Table 4.2 to constraints of the form $v_1 \supseteq \{v_2\}$. The rules defined in the constraints system C are rewritten to the constraint system C' ($C \Rightarrow C'$) using the rewrite rules from Table 4.3 until the system stabilizes and no rewrite rules can be applied.

Normalization		
$C \equiv C' \cup \{v_1 \supseteq *\{v_2\}\}$	$\Rightarrow$	$C \cup \{v_1 \supseteq v_2\}$
$C \equiv C' \cup \{*\{v_1\} \supseteq v_2\}$	$\Rightarrow$	$C \cup \{v_1 \supseteq v_2\}$
Propagation		
$C \equiv C' \cup \{v_1 \supseteq v_2\}$	$\Rightarrow$	$C \cup \bigcup_{v \in Collect(v_2,C)} \{v_1 \supseteq \{v\}\}$
$C \equiv C' \cup \{v_1 \supseteq *v_2\}$	$\Rightarrow$	$C \cup \bigcup_{v \in Collect(v_2,C)} \{v_1 \supseteq *\{v\}\}$
$C \equiv C' \cup \{*v_1 \supseteq v_2\}$	$\Rightarrow$	$C \cup \bigcup_{v \in Collect(v_2,C)} \{*\{v_1\} \supseteq \{v\}\}$

Table 4.3: Constraint rewriting rules defined by Andersen [2]. Rewriting rules for constraints that are never defined in this work are omitted. The function $Collect(v, C)$ returns all abstract locations a variable v points to using the constraint system C.

4.2.6 Objective-C

Having programs that were compiled from Objective-C code introduces some problems that are not present when doing Andersens pointer analysis on C code. Especially method calls (similar to the problem of generating a call graph) are difficult to handle and this is discussed in Section 4.2.6.1. Getting the types of objects and propagating them through the program correlates heavily with creating a call graph and is described in Section 4.2.6.2.

While the already mentioned problems are always present when Objective-C is used other language specific problems arise from specifically using those features in an Objective-C application. Two very commonly used features are *Blocks*, discussed in Section 4.2.6.3, and *Fast Enumeration* for collections as described in Section 4.2.6.4.

4.2.6.1 Call Graph

Pointer analysis is also used to create a correct call graph. The results of the pointer analysis is required when the original code contains calls to function pointers and for all calls to Objective-C methods.

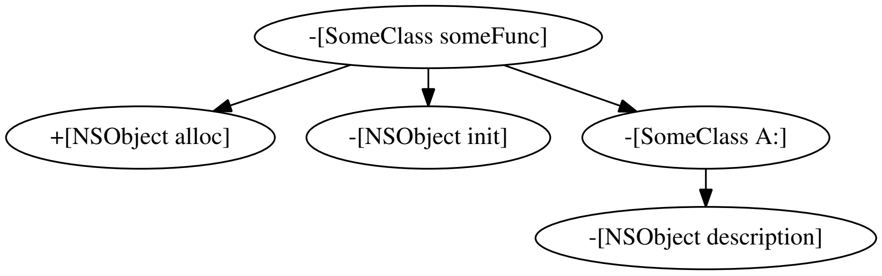
The example presented in Figure 4.3 and Figure 4.4 displays the problem of message sending in Objective-C and call graph generation. Generating the call graph from the original source code of Figure 4.3a is easily possible since the variables and their types are defined clearly. The decompiled code in Listing 4.7 does not allow a simple solution for generating the call graph since neither the type information nor the selector names are defined in the source code. Only calls to the function *objc_msgSend()* are present. This function takes the information about the class or object and the name of the method to call as the first two parameters. In the case of ARMv8 they are stored in the registers X0 and X1, but the values stored there are only references to addresses located in the binary from where the actual information is retrieved. By knowing these locations where the parameters may point to during execution it is possible to restore the original call graph.

Algorithm 3 describes how points-to information is used to restore the original call graph. The algorithm iterates over the values of the first two parameters, that are passed to the Objective-C runtime library, and based on the locations they point to it can be determined which method can be called. The first parameter has to point either to a class info location or to a dynamic allocated location. If the value points to a class info location a class method will be called and if it points to a location that can be associated with dynamic memory allocation and is annotated with type information a instance method is called.

```
-(void) someFunc {
    id o = [[NSObject alloc] init];
    [self A:o];
}

-(void) A:(id)someObject {
    [someObject description];
}
```

(a) The Objective-C source code



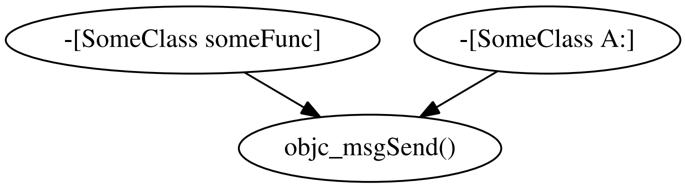
(b) The correct call graph

Figure 4.3: An example for creating the call graph from Objective-C code

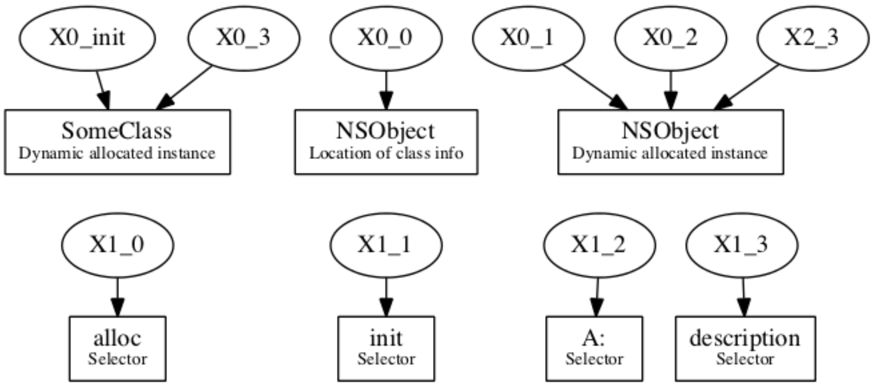
```
define void @"-[SomeClass A:]"(%regset*) {
  ...
  %X0_init = ... ; Contains the 'self' reference
  %X0_0 = load i64, i64* inttoptr (i64 SomeClassAddress to i64*)
  %X1_0 = load i64, i64* inttoptr (i64 allocSelectorAddress to i64*)
  ; store X0_0 and X1_1 to registers X0 and X1
  call void @objc_msgSend(%regset* %0)
  %X0_1 = load i64, i64* X0_ptr
  %X1_1 = load i64, i64* inttoptr (i64 initSelectorAddress to i64*)
  ; store X0_1 and X1_1 to registers X0 and X1
  call void @objc_msgSend(%regset* %0)
  %X0_2 = load i64, i64* X0_ptr
  %X1_2 = load i64, i64* inttoptr (i64 ASelectorAddress to i64*)
  ; store X0_2 to register X2 as the first parameter
  ; store X0_init and X1_2 to registers X0 and X1
  call void @objc_msgSend(%regset* %0)
  ...
}

define void @"-[SomeClass A:]"(%regset*) {
  ...
  ;the first parameter is stored in X2 and is moved to X0 before calling
     the function
  %X0_3 = load i64, i64* %X0_ptr
  %X2_3 = load i64, i64* %X0_ptr
  %X1_3 = load i64, i64* inttoptr (i64 descriptionSelectorAddress to i64*)
  ; store X2_3 to register X0 and X1_3 to register X1
  call void @objc_msgSend(%regset* %0)
  ...
}
```

Listing 4.7: Decompiled code of Figure 4.3a with the relevant variables and function calls.



(a) The faulty call graph created using the decompiled code from Figure 4.7.



(b) The call graph created using the decompiled code

Figure 4.4: Figure 4.4a shows the wrong call graph using only the code of Listing 4.7 and no points-to information. The points-to graph in Figure 4.4b calculated from Listing 4.7 is used to reproduce the original call graph of Figure 4.3b.

Algorithm 3 Using points-to information to restore the call graph

```
Require: CallInstruction, PtsTo_a, PtsTo_b      ▹ Points to set of the first two parameter a and b
  for all loc_a ∈ PtsTo_a do
      ClassMethod ← false
      if PointsToClassInfo(loc_a) then
          ClassMethod ← true
      end if
      Type ← GetTypeName(loc_a)
      for all loc_b ∈ PtsTo_b do
          if not PointsToSelector(loc_b) then
              continue
          end if
          Selector ← getSelectorName(loc_b)
                                  ▹ If the method is defined in the binary or it is a known library
          if not KnownMethod(Type, Selector, ClassMethod) then
              continue
          end if
          if containsEdge(CallInstruction, Type, Selector, ClassMethod) then
              continue                                                  ▹ Add edges only once
          end if
          addEdge(CallInstruction, Type, Selector, ClassMethod)
      end for
  end for
```

4.2.6.2 Types

Type information is collected by identifying calls made to *alloc* functions. Whenever an edge gets added to the call graph the new edge gets checked, if it is a call to an *alloc* method. In the case of having such a call an abstract location is created for the returned value and gets annotated with type information or if the return value already points to an abstract location, the new type information is added to the existing information. Even if every abstract location is associated with a single call instruction, it may have multiple types. The message, that is passed to the Objective-C runtime library for an *alloc* call, contains the reference to the class information as first parameter and if this parameter points to different class informations in the binary the abstract locations has multiple types.

Propagating the types of object through the program needs no additional actions. The constraint system is solved in every iteration of the call graph generation (Algorithm 2) and since the abstract locations are annotated with a type all variables that point to this particular abstract location have this type.

4.2.6.3 Blocks

This is a feature that was added by Apple to the languages C, C++ and Objective-C. Blocks are similar to *closures* or *lambdas* in other languages. They allow also to access values from the enclosing scope (the surrounding function) and variables that use the `__block` type modifier can be modified inside a block and these modifications are shared with all locations from which the variable can be accessed. Additionally to sharing variables parameters can be passed to blocks like in ordinary functions as well.

While passing parameters is handled in the same way as in function calls, variables that are shared with the surrounding scope need to be treated separately. To be able to modify a variable in different places all modifiers need to access the same location in memory. Since a block can be scheduled for later execution (for example the block may be executed in a separate thread) the stack frame of the function where the block was defined may be not valid anymore when the block is executed. For this reason the variable needs to be stored in a separate location as long as the block is valid. The allocation of this memory and further steps that allow to access this memory from multiple locations are handled by the runtime environment and not by functions defined in the analyzed binary. Nevertheless it is possible to determine the statements that access this variables. This is again done by identifying a pattern in the block when a variable is accessed. The information that is known initially is defined in the block variable (see Figure 4.5a) and the value stored there is only a reference to data structure that holds another reference to the actual data.

How this works in detail is described based on the example in Listing 4.8. First a pattern has to be found that exists whenever a block variable is accessed (the pattern uses the names used in Figure 4.5a for simplicity reasons):

```
block_y2_Addr = load i64, i64* block_y2_Ptr
block_y2_Shared_Ptr_Addr = add i64 block_y2_Addr, 8
block_y2_Shared_Ptr = inttoptr i64 block_y2_Shared_Ptr_Addr to i64*
sharedPtr1 = load i64, i64* block_y2_Shared_Ptr
yShared_Addr = add i64 sharedPtr1, Offset2; Offset2 ∈ ℤ
yShared = inttoptr i64 yShared_Addr to i64*
```

The variable `block_y`$_{2\_\text{Ptr}}$ can be easily matched to `block_y`$_{1\_\text{Ptr}}$ using the method of matching stack stored parameters like it is done with variables that are passed by value. This variable holds the reference to `block_y`$_1$ (`block_y`$_{1\_\text{Addr}}$), which is stored on the surrounding functions stack. Somewhere in this function the (stack relative) address of `block_y`$_1$ is stored there and by finding `store` instruction that store a value to `block_y`$_1$`_ptr` it is possible to find the address of `block_y`$_1$. The variable `block_y`$_1$ is a struct that contains some flags, that are not needed here, of 8 bytes size and `shared`$_{\text{Ptr1}}$ for address of the actual shared variable $\mathtt{y}_{\mathtt{Shared}}$. Whenever a memory operation is found that uses $\mathtt{shared}_{\mathtt{Ptr1}}$ and adds an offset of Offset_1, while fulfilling the condition $\mathit{Offset}_1 = \mathit{Offset}_2$, the assumption that this location refers to $\mathtt{y}_{\mathtt{Shared}}$ is made and a constraint is added for this instruction. The result of the operation of adding Offset_1 to $\mathtt{shared}_{\mathtt{Ptr1}}$ is called again $\mathtt{y}_{\mathtt{Shared}}$ and the following constraint handles both the block itself and the surrounding function where the block was defined:

$$\mathtt{y}_{\mathtt{Shared}} \supseteq \mathtt{y}_{\mathtt{Local}} \tag{4.22}$$

```
-(void)blockExample {
  int x = 123;
  __block y = 0;
  void (^block)() = {
    y = x;
    x = 321;
  }
  block();
}
```

Listing 4.8: An example how blocks can access variables that are defined by the surrounding function. Variables without the `__block` modifier are passed by value to a block. In this example `x` has still the value 123 after executing the block. The variable `y` changes its value to 123 since it is defined as `__block` variable.

4.2.6.4 Fast Enumeration

Objective-C provides collection classes to handle groups of Objective-C objects. They can be accessed either by sending messages to the collection object or by enumerating over this collection. For enumerating a concept called *Fast Enumeration* was introduced that increased performance by reducing the number of messages needed to be sent to the collection. This is done by copying the pointers of the containing objects to a separate, contiguous memory location, which can be easily iterated through without sending any additional messages. In the case of an array this reduces the message sending overhead from one message per element to a single message that initialized this memory location with the object pointers. For analyzing code that uses this concept, instead of the simple method of retrieving each element solely, makes this task more difficult. It is easy to detect the use of fast enumeration, but the difficult part is to detect where the pointers of the objects get copied to and the instructions that access these pointers.

Lets take the collection class *NSArray* as an example.

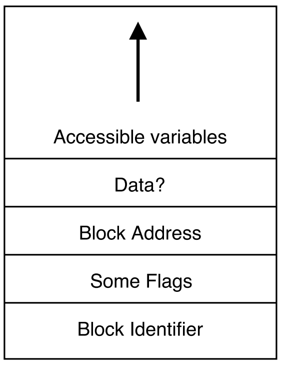
```
    - (NSUInteger)countByEnumeratingWithState:(NSFastEnumerationState *)
       state objects:(id [])stackbuf count:(NSUInteger)len
```

and the parameter *state* is a reference to the struct:

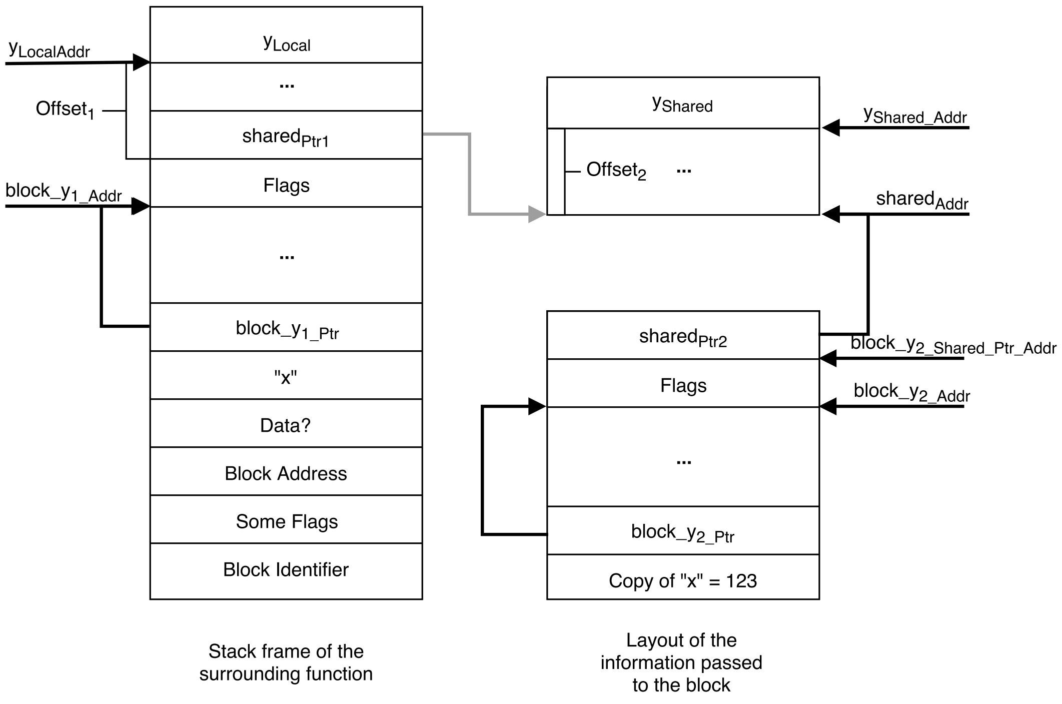
```
typedef struct {
    unsigned long state;
    id *itemsPtr;
    unsigned long *mutationsPtr;
    unsigned long extra[5];
} NSFastEnumerationState;
```

The parameter *stackbuf* will contain the pointers to the objects after calling this method. However, the variable *itemsPtr* of the *state* parameter will also point to this location and iterating over the objects this value will be accessed and not the buffer itself. This means that the buffer can not be used directly to identify those instructions that access it.

For pointer analysis, and further for program slicing, this means that the code does not contain a instruction that writes to this location when we apply fast enumeration and therefore such calls have to be handled separately during pointer analysis. This is done by taking the passed *state* parameter and check, if it points to a stack location or if it is a dynamically allocated location. For the case of a stack references it is possible to determine its offset in the current function by evaluating the arithmetic operations made to the stack pointer before reaching the call instruction for fast enumeration. Having this offset the size of the *state* variable in the *state* struct has to be added and the result is the location of the array items. Finding the instructions that access this computed stack offset results in identifying those instructions that access the arrays elements and allows further analysis of those elements. In case of having a dynamic allocated *state* parameter the pointer to this variable is handled similar to the case of having a stack variable where the register, that contains the location of the dynamic allocated memory, is treated like the stack pointer with offset of zero relative to the function entry and to identify those instructions that access the *itemsPtr* a register containing the address to this location has to be used as base pointer with an added offset of the size of the *struct* variable.



(a) The general layout of a block variable.



(b) This image shows how variables of the surrounding scope are accessed from the example of Listing 4.8. The grey arrow indicates that the value in `Placeholder` is set by the runtime environment before a block is used and not by the surrounding function itself.

Figure 4.5: Layout of a block variable and an example how they are stored in an actual example

Chapter 5

Parameter Backtracking

Program slicing as it was described in the previous chapter is extended to provide the feature of parameter backtracking for the used programs. With backtracking it is possible to find the origin of a parameter and this step is needed before being able to check, if a parameter fulfills the defined rules. The difference to program slicing, or the additional features that are added to it, are mainly related to the fact that a program slice contains all relevant statements and variables that influence slicing criterion while backtracking focuses only on a specific variable and provides a data flow path that originates at the location where the parameter was first set and all its further modifications until it was used. Backtracking is needed to pinpoint those instructions that actually modify or define a value and to allow validation of these instructions via defined rules as it is done later during analysis.

5.1 Finding Predecessors

The static single assignment form (SSA) of the LLVM intermediate representation provides already a simple way of backtracking for instructions, which do not use pointers for accessing memory. SSA ensures that a variable is defined a single point in the program only and if this variable is referenced anywhere it is immediately possible to find its definition. However, for pointers this is more complex because, if a value is read from a location referenced by a pointer, it is not possible to determine the statement where a value was written to this location by solely inspecting the statement. This is very similar to the requirements of slicing programs in the presence of pointers, but in the case of program slicing it is only needed to add the location to the relevant set and traverse the CFG backwards to find a modification of this location. For backtracking we need to specify more than just the relevant location. Whenever a location l is added to the relevant set the statement s, which induced this location, is added to a separate set $R_{Sources}(i, l)$ that contains all statements introduced the relevant variable l at statement i and this allows to match modifications with the corresponding read instructions.

How these sets are defined for each instruction and the way the sources of relevant variables are propagated through the program is defined below. Similar as in the definitions for static slicing the instruction j is an immediate successor of the instruction i ($i \rightarrow_{CFG} j$):

$$\begin{aligned} l &\in Loc \\ R_{Sources}(i,\ l) &= \{i \mid i \in S_C,\ l \in REF(i)\} \cup \{i \mid i \in R_{Sources}(j,\ l),\ i \notin S_C\} \end{aligned} \tag{5.1}$$

With this information it is now possible to determine the predecessing modifications of a location, if a value is accessed using a pointer. If the statement r reads a value from a location l the predecessors for backtracking the read value are defined as:

$$Pred'(r) = \{s \mid r \in R_{Sources}(s,\ l),\ l \in DEF(s)\} \tag{5.2}$$

And the set of predecessors that includes all instructions, which modify a referenced variable is defined as:

$$Pred(i) = \{s \mid r \in R_{Sources}(s, l),\ l \in DEF(s),\ l \in Loc\} \cup \{op \mid op \in Operators(i),\ op \in Instructions\} \tag{5.3}$$

For function calls there is no need to handle them separately since relevant variables that represent locations are valid for all functions and parameters or return values that are stored in registers are handled implicitly using the *REF* sets as described in Section 4.1.1.

5.2 Generating Paths

Using the *Pred*(*i*) set of an instruction *i* we get all predecessors of this instruction. This may be multiple instructions for two reasons. First, an instruction *i* may reference multiple different values an for each of these values has a different predecessor where it is defined. Second, the statement *i* can be reached via different paths in the CFG and in each path a referenced value be defined differently as well.

All possible for a value *v* to its origin can be defined as graph $G = (V,\ E)$. The set of vertices *V* contains all instructions of the program and the edges *E* are defined by adding all reachable predecessors recursively:

$$\begin{aligned} E^{(0)} &= \{(v,\ j) \mid j \in Pred(v)\} \\ E^{(k)} &= \{(i,\ j) \mid (j,\ k) \in E^{(k-1)}, i \in Pred(j)\} \end{aligned} \tag{5.4}$$

This graph will contain edges that lead to values, which are not relevant for a analysis that tries to find the origin of a specific value. This holds especially for values that are accessed using pointers. This instruction will not only reference the location itself, but also the pointer variable itself. To get rid of these values of no interest no edges are added for such values and this eliminates automatically this whole path and only the actual relevant variables are used to add new edges.

The example in Figure 5.1 shows how the backtrack graph of the example shown in Listing 5.1 looks like, if the value *%val* is backtracked. The example includes a pointer to show that not all referenced variables are fully backtracked. The actual graph only contains two paths and both end at the constant definitions of *%a* and *%b*. Even though in this example only a few statements are not present in any of the backtracking paths it can be seen that this approach only includes the actual relevant statements and, as it is the case with having decompiled binaries as described earlier, there will be a vast amount of pointer variables, which are usually not relevant for backtracking and can be discarded (like *%ptr* in this example).

```
  %a = i32 0
  %b = i32 1
  %ptr = alloca i32
  br i1 %cond, label %path1, label %path2

path1:
  store i32 %a, i32* %ptr
  br label %end

path2:
  store i32 %b, i32* %ptr
  br label %end

end:
  %val = load i32, i32* %ptr
```

Listing 5.1: The example code for the backtracking paths shown in Figure 5.1. The dashed red edges indicate paths that are not pursued.

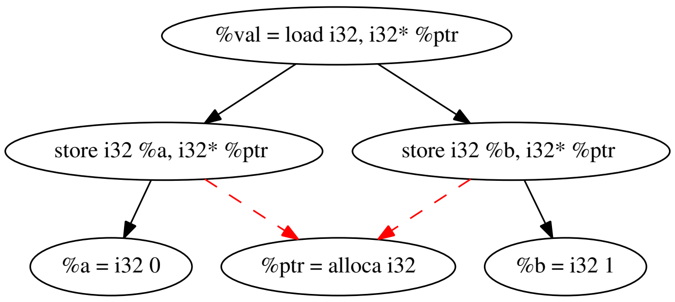


Figure 5.1: A value is loaded using a pointer to witch's location is written on different locations in the program.

5.2.1 Cycles

Whenever parts of code are executed inside a loop cyclic dependencies may occur. If these dependencies are not handled properly, they will lead to infinite loop during analysis.

The example in Listing 5.2 shows an (endless) loop where the value *X0.0* references itself, creating a cycle, and a value *X0_1*, which is defined outside the loop. Figure 5.2 shows the backtracking path for this example including the cyclic dependency that would cause an endless path.

```
blockA:
  %X0.0 = phi i64 [ %X0.0, %blockA ], [ %X0_1, %blockB ]
  br label blockA

blockB:
  %X0_1 = or i64 %X0_0, 0
  br label blockA
```

Listing 5.2: The variable *X0.0* causes a cyclic dependency by referring to itself.

To be able to handle cyclic dependencies the branch of a path is discarded immediately if a instruction is encountered that is already contained in the path. This means that branches without cyclic dependencies are still continued and will lead to the initial definition of this value. For the example in Listing 5.2 this means that the path to *X0_1* will be continued, but *X0.0* appears only once on this path.

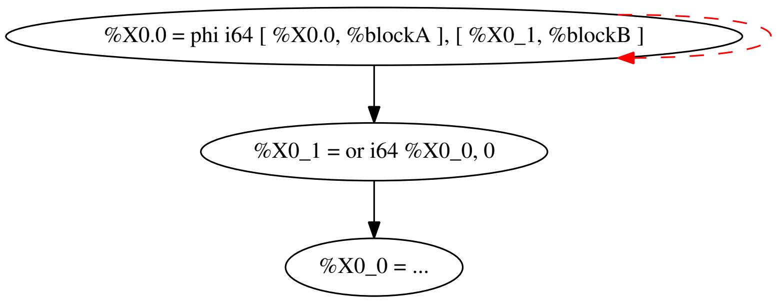


Figure 5.2: At the initial instruction *X0.0* there are two edges that set the next node, but only the one to *X0_1* will be followed since the path to *X0.0* can be added infinitely often.

Chapter 6

Cryptographic Misuse

With the methods discussed in the previous chapters it is now possible to take a decompiled iOS binary and find paths to the origin based on slicing criterions by backtracking parameters. This allows to check, if cryptographic functions are used correctly. The missing chapter for the complete analysis describes when cryptographic misuse is given in an application. For the purpose of this work the rules are only defined for cryptographic functions that are already provided by the iOS system (described in Section 6.2) and not for any third party libraries that provide similar functionality.

These rules will be used by the implementation and violations to them will be interpreted in the evaluation part of this work.

6.1 Definitions

This section gives some basic definitions for terms used later when the rules to detect cryptographic misuse are described.

Block Cipher A block cipher is a function that takes a message of fixed length and transforms this message to another message of the same length by applying a symmetric key. The encryption for a block cipher is defined as:

$$\begin{aligned} E_K :\ \{0,1\}^n &\rightarrow \{0,1\}^n \\ E_K(M) &= C \end{aligned} \tag{6.1}$$

Respectively, the decryption function of a block cipher

$$\begin{aligned} D_K :\ \{0,1\}^n &\rightarrow \{0,1\}^n \\ D_K(C) = D_K(E_K(M)) &= M \end{aligned} \tag{6.2}$$

Symmetric encryption scheme With a symmetric encryption scheme messages of arbitrary length can be encrypted and decrypted using a symmetric key. A symmetric encryption scheme $\mathcal{SE}$ is defined by the following triple:

$$\mathcal{SE} = \{\mathcal{K}, \mathcal{E}, \mathcal{D}\} \tag{6.3}$$

Where $\mathcal{K}$ is the random key generation algorithm for generating a key K. $\mathcal{E}$ and $\mathcal{D}$ define the encryption and decryption algorithms, which use the symmetric key K.

Electronic Codebook Mode In ECB mode $\mathcal{E}$ and $\mathcal{D}$ of the encryption scheme $\mathcal{SE}$ are defined as:

$$\begin{aligned} \mathcal{E} : \forall i \in \mathbb{N}^+ :\ & C_i = E_K(M_i) \\ \mathcal{D} : \forall i \in \mathbb{N}^+ :\ & M_i = D_K(C_i) \end{aligned} \tag{6.4}$$

All blocks of data are encrypted and decrypted independently of other blocks so equal plaintext blocks encrypted with the same key always lead to equal ciphertext blocks.

Cipher Block Chaining Mode In CBC mode the input to the encryption function is always depending on the output of the last encryption operation, or on the initialization vector for the first block of data:

$$\begin{aligned} & C_0 = IV \\ \mathcal{E} : \forall i \in \mathbb{N}^+ :\ & C_i = E_K(M_i \oplus C_{i-1}) \\ \mathcal{D} : \forall i \in \mathbb{N}^+ :\ & C_i = D_K(C_i) \oplus C_{i-1} \end{aligned} \tag{6.5}$$

Using a different initialization vector for the same plaintext message that is encrypted with the same key leads to a different ciphertext message.

IND-CPA This stands for *indistinguishability under chosen plaintext attack*. Consider the following scenario:

- An attacker has given a ciphertext C_x that was generated from one of the plaintexts M_1 or M_2, but the attacker does not know which one
- The used key K is unknown to the attacker
- The attacker is able to perform additional encryption operation with a chosen plaintext

IND-CPA security is now given, if the attacker has no, or negligible advantage over guessing which message M was used for C_x.

6.2 The Cryptographic System in iOS

Apple already provides APIs for cryptographic functions that are included in iOS[1]. These APIs are distributed across two libraries, namely *CommonCrypto* and *Security*. The first contains all symmetric cipher functions as well as hashing operations and a key derivation function. The latter provides functionality for asymmetric ciphers and all related operations like certificate handling or keychain access. All rules that are described later in this chapter have their origin (the point where the slicing criterion is defined) in function of *CommonCrypto*, but for detecting termination points of paths, the instructions in a program where a path does not need to be pursued further, the *Security* library is considered as well. For random numbers it is even necessary to consider both since secure random number generators are defined in both.

There are now three types functions in the *CommonCrypto* library that need to be handled for defining slicing criterions:

- **CCCryptorCreate**: this is a whole family of similar functions that initialize a *CCCryptorRef* reference with all its needed parameters. These parameters include everything we need to check. The *CCCryptorRef* is then used to perform encryption or decryption on any data.
  - *op*: defines if the data has to be encrypted or if it gets decrypted

[1] `https://developer.apple.com/library/mac/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html`

  - *options*: specifies the cipher mode (ECB or CBC)
  - *key*: a pointer to the key data
  - *iv*: a pointer to the IV data

- **CCCrypt**: similar to *CCCryptorCreate* functions, but this function is a "one-shot-function" where all options and all data for the cipher are already passed to the function and the result is returned immediately. The passed parameters of interest are exactly the same as for *CCCryptorCreate* functions.

- **CCKeyDerivationPBKDF**: derives a key as defined by *PBKDF2* using a password, salt and a number of iterations.

  - *password*: the password as C-string
  - *salt*: a pointer to the salt data
  - *rounds*: the number of rounds

6.3 Description of the Rules

In this work we check the same rules as Egele et al. in [7], except for the rule that defines that the seed for a random number generator can not be constant because this does not have to be done in iOS (it uses `/dev/random`).

The remainder of this chapter gives a short description why it is necessary to check for each rule and how this is done with the cryptographic functions provided by iOS.

No ECB mode for encryption The basic problem about data encrypted with ECB mode is that every block of data is encrypted independent and thus same plaintext blocks result in same ciphertext blocks. This allows to obtain some knowledge very easy:

- If ciphertexts match they encrypt the same plaintext data
- Common prefixes can be found
- Long sequences of the same data are exposed (for example many null-bytes)

Consider an application where the attacker knows that the set of messages $M \in \mathcal{M}$ is small, where small means that computing the ciphertexts for all messages M is feasible. For example there are only the messages M_1 and M_2 that signal the outcome of a coin flip. If the attacker is able to encrypt both messages (chose plaintext) he immediately knows the input message M for any ciphertext C. Since such an attack is possible, the operation is not *IND-CPA* secure.

To detect a violation of this rule calls to *CCCrypt* or *CCCryptorCreate* functions have to fail the condition:

$$op = kCCEncrypt \wedge (options \mid kCCOptionECBMode) = kCCOptionECBMode$$

No non-random IV for CBC encryption If a initialization vector is either constant or predictable it is possible to obtain similar knowledge as in the case of using ECB mode.

While in EBC mode any block can be compared to another block this is not possible in the case of CBC mode. CBC data can only be compared from the first block on to find matching prefixes if the same key and the same IV was used. A common prefix, of course, also allows to check for equal ciphertext-plaintext pairs since the prefix may be the whole data.

The description above holds, if a static initialization vector was used, however, if it is only predictable comparing ciphertexts will not disclose that much information, but using a simple attack will do so.

If an attacker is able to perform a chosen plaintext attack and the initialization vector is predictable the encryption will again not be *IND-CPA* secure. The ciphertexts C and C' were generated from the messages M and M' using the distinct initialization vectors IV and IV'. Clearly, the ciphertexts will not be equal, but it is still possible that $M = M'$ and the goal of this attack is to detect this.

Consider C'_1, the first block of the ciphertext C', which is computed by $C'_1 = E_K(M'_1 \oplus IV')$, but since this is a chosen plaintext attack this can be changed to $C''_1 = E_K(M'_1 \oplus IV' \oplus IV' \oplus IV) = E_K(M'_1 \oplus IV)$ and if $M = M'$ holds, this means $C''_1 = E_K(M'_1 \oplus IV) = E_K(M_1 \oplus IV) = C_1$. M' is chosen freely. So using this approach it is possible to find out, if M was the original plaintext.

To be sure that always a random IV is used the following condition has to hold:

$$iv \in Random \wedge (options \,\&\, kCCOptionECBMode) \neq kCCOptionECBMode$$

Random is data generated by a cryptographic secure random number generator, which is defined in both libraries (*CCRandomGenerateBytes* in *CommonCrypto* and *SecRandomCopyBytes* in *Security*).

No constant encryption keys Encrypting data with a constant key means that this key has to be hardcoded in the binary and thus it is no secret and as a result the encrypted data is not private at all. We also consider a constant password that is passed to the *CCKeyDerivationPBKDF* function as constant key:

$$key \notin Constant \ \wedge \ password \notin Constant$$

No constant salts for PBE Adding random salt to passwords prevents from generating the same key for same passwords. If the salt value is constant the key will always be the same.

A possible attack is a so-called *pre-computed dictionary attack* where keys are computed based on passwords. Having a constant salt value means that this list of keys has to be computed only once. The result of this computation is a rainbow table, which can be used easily to match encryption keys with their password counterparts. Using random salts requires the computation of this rainbow table to be done for each salt value, which leads to an infeasible attack.

To prevent this constants for salt values must not be used:

$$salt \notin Constant$$

Not less than 1000 iterations for PBE The number of iterations in password based encryption has the purpose of extending the time needed for deriving a single key from a password. Having a large number of iterations prevents the fast computation of the previously mentioned rainbow tables and further, a brute force attack, which uses a dictionary of possible password as input, should also be made infeasible by increasing the time spent computing a single key.

The NIST recommendation published 2010 [24], which is the latest recommendation at the moment, states that at least 1000 iterations should be used for password based encryption. However, it also says that 100000 iterations may be appropriate, if the performance of the application is not critical. The reason for a high iteration count is to make brute force attacks harder and with the increase of computation power this number also need to be increased permanently. Nevertheless, for a rule we still use the lower bound of the recommended values for parameters that are passed to *CCKeyDerivationPBKDF*:

$$rounds \geq 1000$$

6.4 Constant Data

The first use of constant data is very easy to detect since this means that a pointer is used that points to a location in the binary that stores constant data. However, it might not be the case that this pointer is passed directly to a cryptographic function but via a Objective-C *NSData* object that contains this pointer. This means that at some point this *NSData* gets allocated using the usual way of allocating objects in Objective-C and now this object does not have a constant address anymore, but still the containing data is in fact constant. In a case like this it is necessary to backtrack this object until the initialization point is reached and if constant data is used for initializing the object, the initialized object is considered constant as well.

The same rules apply for other Objective-C classes as well as for strings, which can be constant C-string or constant NSString objects. In both cases there is a pointer to constant data in the binary at some point.

Part II

Implementation Details

Chapter 7

System Overview

The system implements the methods described in the previous chapters to detect cryptographic misuse in an iOS binary by performing static analysis on it. In Figure 7.1 it is shown how the different stages, that are needed for getting this information for a given binary, interact with each other.

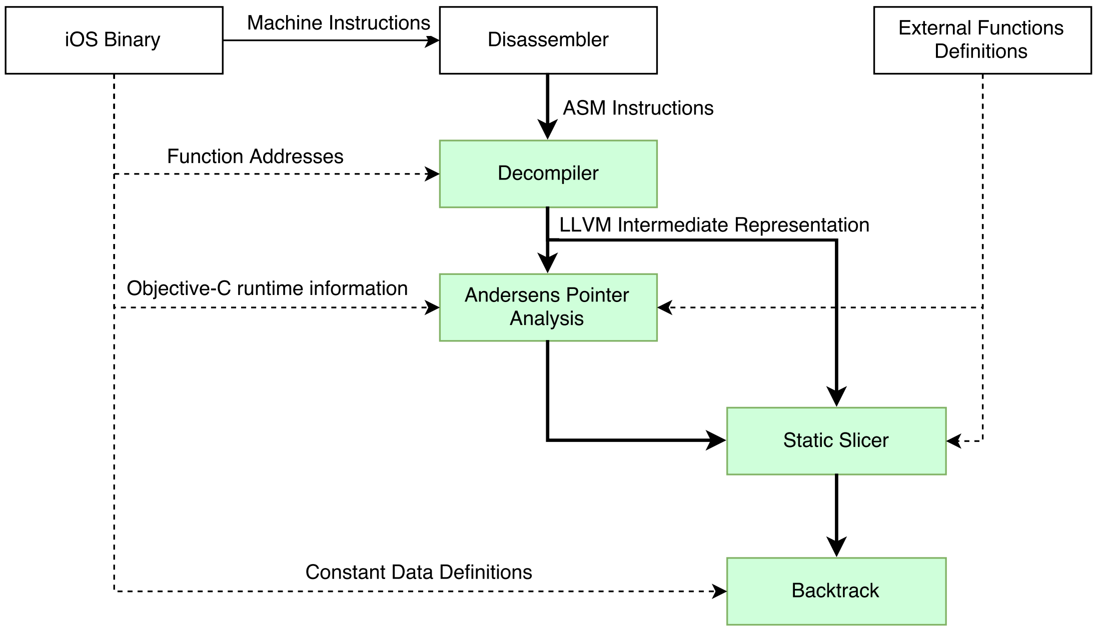

iOS Binary
Machine Instructions
Disassembler
External Functions Definitions
ASM Instructions
Function Addresses
Decompiler
LLVM Intermediate Representation
Objective-C runtime information
Andersens Pointer Analysis
Static Slicer
Constant Data Definitions
Backtrack

Figure 7.1: The workflow of the system

The main tasks for the static analysis in this work are:

1. **Disassemble**: generating ARMv8 assembly code from a Mach-O binary. This is already supported by the LLVM framework and *Dagger* uses this implementation.
2. **Decompile**: translating ARMv8 assembly code into LLVM IR with the extended *Dagger* framework.
3. **Pointer Analysis**: the supporting analysis for static program slicing to handle pointers.
4. **Static Slicing**: identifying the relevant code in a program based on slicing criterions defined by rules that describe when cryptographic misuse is given.
5. **Backtracking**: using the information generated during program slicing to determine the paths a parameter can take through the program to end up at the slicing criterion.

Each of these steps uses information from a previous task or global available information. Global information is accessible by all steps during the process of static analysis. It is either the binary itself, which remains unmodified during the whole process, or definitions for external functions. This are functions which are not defined inside the binary, so no code is available, but they still might be called using an external library. For getting precise results this informations is crucial since the generated intermediate code does not contain function headers, parameters or similar information that is lost during compilation and without these definitions it is not possible to make any assumptions about a used external function.

The first step, disassembling a binary, is already provided by the LLVM framework and will not be discussed further in this work. For the remaining steps either a new solution had to be provided or an existing framework had to be modified. With the result of the described process it is possible to perform static analysis on iOS binaries and apply the defined rules to detect cryptographic misuse.

The following sections describe those solutions and modifications in detail. Section 7.1 describes the decompilation task that takes assembly code and produces LLVM IR code using the *Dagger*[1] framework as a base that is extended to be able to handle ARMv8 instructions provided by an Mach-O binary. Followed by Section 7.2 which describes the modifications made to LLVMSlicer[2] which includes the pointer analysis and backtracking of parameters.

7.1 Extending *Dagger*

Dagger is currently not capable of handling binaries compiled for ARMv8, as required in this work, and needs to be extended to support this. What is missing for the support of ARMv8 is the complete definition of instruction semantics. These definitions were already described in Section 3.2.2 and defining these semantics for a single instruction is straight forward as described there, thus it is not discussed further.

However, changes to the way how registers are handled and simplifying the output using passes have to made as well and these changes are described in this section, as well as the optimizations made to the output code for simplifying it for the analysis.

7.1.1 Registers

Dagger tries to represent the CPUs internal storage, the register file, in a similar fashion by creating a datastructure that represents this register file in LLVM IR. As described in Section 3.2.4 the registers of an ARMv8 CPU are overlapping using the LLVM definitions and storing multiple sub-registers in a single super-register will lead to difficulties in the analysis because data dependencies can not be computed easily with these overlaps.

This model is modified in a way that now a single super-register will only store the value of exactly one physical register as shown in Figure 7.2. The largest super-registers originally contains the values of four physical registers and every physical register is stored in exactly four different largest super-registers. This means that three values of every of the largest super-register can be removed and still no data is lost.

If an instruction accesses multiple physical registers using a super-register, which now contains only the data of a single physical register, multiple super-registers need to be merged into a single value which is then used by the translated instructions.

Modifying *Daggers* approach to this solution still keeps the data stored in registers consistent while allowing an easier computation of data dependencies of a super-register.

[1] `http://dagger.repzret.org/`

[2] `https://github.com/jirislaby/LLVMSlicer`

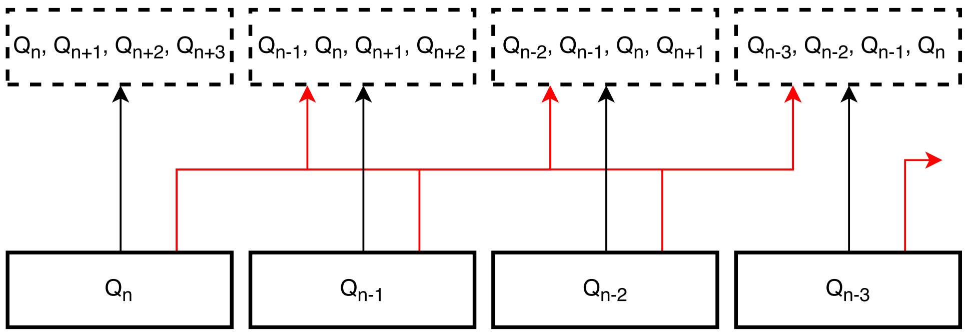

Q_n, Q_{n+1}, Q_{n+2}, Q_{n+3}
Q_{n-1}, Q_n, Q_{n+1}, Q_{n+2}
Q_{n-2}, Q_{n-1}, Q_n, Q_{n+1}
Q_{n-3}, Q_{n-2}, Q_{n-1}, Q_n
Q_n
Q_{n-1}
Q_{n-2}
Q_{n-3}

Figure 7.2: A single super-register stores only the value of a single physical register. The dependencies marked in red are removed.

7.1.2 Non-Volatile Registers

The content of a *non-volatile* register must be preserved across function calls, so the register contains the same value before and after the function call. These registers are used to handle local values without storing them on the stack. In the Procedure Call Standard [3] they are referred as *callee-saved registers* and, as this name already suggests, these registers are saved by the called function, so the register can be used in this function, and right before returning to the calling function the value gets restored so the registers value always corresponds to the value that was set by the calling function.

This means that the callee accesses these values only for saving and restoring them. Still, there will be data dependencies between the caller and callee caused by storing these values and these data dependencies are not necessary since the values are not used for anything else. The goal of this pass is to remove those unnecessary data dependencies to simplify the code for analysis usage.

The following example shows the changes made with this pass by replacing the value loaded from the register set after calling a function with the value a register had before calling the function.

```
%X19_0 = ...
store i64 %X19_0, i64* %X19_ptr, align 4
call void @someFunction(%regset* %0)
%X19_1 = load i64, i64* %X19_ptr, align 4
...
store i64 %X19_1, i64* %ptr, align 4
```

Listing 7.1: Non-volatile registers are handled via the register file prior to running the pass

The value of *X19_0* would be saved to the stack in *someFunction()* and restored to the register *X19* before returning so *X19_1* will have the same value as *X19_0*. This code can be simplified to the code showed below where the *store* and *load* instructions before and after the *call* statement are removed and references to the variable *X19_1* are replaced by the variable *X19_0*.

```
%X19_0 = ...
call void @someFunction(%regset* %0)
...
store i64 %X19_1, i64* %ptr, align 4 // %ptr can point to anything but
    to a register
```

Listing 7.2: Load and store instructions of non-volatile registers are removed

Additionally to keeping the values of *non-volatile* registers local to a function the code size is also decreased since the number of *load* and *store* instructions is reduced as they are not needed anymore and get removed from the code.

7.1.3 Tail Calls

A function call is called *tail call*, if it is the final instruction of a function, which means that no instruction is executed after the function call in this function before returning to the calling function. In ARMv8 this is done by replacing a branch and link (*BL*) instruction with a branch (*B*) instruction. This is possible because the *RET* instruction uses only the link register to determine where the function has to return to and the function with a tail call just skips the step of setting the link register so the previous value is used for returning.

In *Dagger* the code at the location of the branch target is always added to the current function since branch instruction can not leave the scope of a function in LLVM IR (only *call* instruction can). This means that, if a tail call is present in a function, the target functions code gets added to the function containing the tail call to include it in the current function. The code of the target function of a tail call is then decompiled multiple times because one time this is done for the function itself, like it is done for every other function as well, and the additional decompilations happen once for every function that has this function as target in a tail call. This leads to a larger output due to duplicate code, but the bigger issue is that the resulting call graph is incorrect because no edge gets added for the tail call since this code is inside the calling function now.

A simple program, like the one in Listing 7.3, will lead to an optimized code that contains a tail call as shown in Listing 7.4. Figure 7.3 shows the execution path of this short example and the problem is clearly visible there as *bar* directly returns to *foobar*.

```
void foobar()
{
  ...
  foo();
  ...
}

int foo()
{
  return bar();
}

int bar()
{
  return ...;
}
```

Listing 7.3: Compiling this example results in a tail call when the function *bar()* is called.

```
foo:
  B   bar

bar:
  ... # Some computations
  RET
```

Listing 7.4: The ASM code of the function *foo* and *bar* of the example in Listing 7.3

The expected behaviour Without tail call optimization, and the assembly code of *foo*

```
foo:
  BL  bar
```

```
    RET
```

Listing 7.5: The function *foo* without a tail call

And this is exactly the wanted behaviour of the decompiled code. Since the tail call consists only of a single branch instruction and the unoptimized solution has two instructions it is not possible to simply replace the branch instruction with the *BL* and *RET* instructions.

This issue is solved by checking the target address of each branch instruction. If the target address is now outside of the function body the branch instruction is considered being a tail call and is replaced by a call instruction to this function followed by a return instruction.

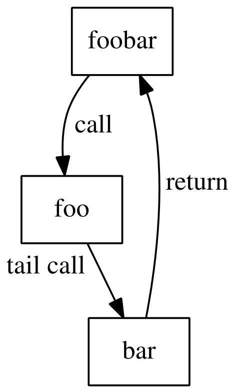


Figure 7.3: *foo* gets called by *foobar* and then calls *bar* at the end. *bar* never returns to *foo* but directly to *foobar*.

7.2 LLVMSlicer

This chapter describes the implementation details for creating slices and the backtracking of parameters using static slicing. The input for this task is the produced LLVM IR of the decompiler described in the previous chapter. This code gets sliced, based on the slicing criterions defined by the rules described in Chapter 6 and during the process of program slicing the information for backtracking (described in Chapter 5) is created. This information is then used to create paths from the slicing criterion to its origin and, additionally to the program slice itself, this is the output generated here and required for the last step of verifying the rules of Chapter 6.

For these goals an existing implementation of an static slicer, namely the LLVMSlicer[3], which was modified to fit the needs described in Chapter 4. This includes creating and using points-to information and further the method of backtracking as described in Chapter 5 is also added to this slicer.

7.2.1 Andersen's Pointer Analysis

Even though LLVMSlicer already includes a pointer analysis implementation we use the implementation of Chen[4] since it is easier to modify for fulfilling the requirements as defined in Chapter 4.2 and LLVM-Slicer also provides a simple interface that accesses points-to information for which a wrapper was made without touching the remaining parts of the implementation of LLVMSlicer. An additional reason why to use Chens implementation is that it implements the constraints solving algorithm of Hardekopf and Lin [10] that performs well on large programs, with which has to be dealt due to decompiling binaries in this simple way.

7.2.2 Restoring Missing Type Information from the Binary File

Missing type information refers to objects that are accessed in the binary, but are allocated elsewhere (in an external function). The considered cases are objects that get passed as parameters to protocol methods and instance variables. In the case of instance variables it may be the case that they will be allocated by an function of an external library so no type information can be found by analyzing the code only. Nevertheless, as described in Section 3.3.3 and Section 3.3.4 those type informations are usually available in the binary and can be used. Retrieving this information it is possible to annotate these instance variables and parameters with their types without having the code that allocates these objects. Consider that the type is only the most general type an object can have: the common superclass of all objects that may be passed. Objects of a subclass may be used during execution, but they will not be recognized.

Instance variables are always accessed by loading an offset from an static address in the binary, where every instance variable has its offset stored at an own address, and by this way of handling this it is possible to get all instructions that access a particular instance variable. Further, this means for the pointer analysis that a single abstract location can be created for every instance variable.

Protocol methods, however, are different since we can not be sure, which parameters may point to the same location, and thus for each parameter, which is an Objective-C object, an own abstract location is created. For example take two methods of the *UITextFieldDelegate*:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
- (BOOL)textFieldShouldClear:(UITextField *)textField
```

In both of the methods above the parameter *textField* may point to the same object, but it can not be determined for sure. This is handled by creating an abstract location for each object that is passed as

[3]https://github.com/jirislaby/LLVMSlicer

[4]https://github.com/grievejia/andersen

an parameter and for each method. For the example above the *textField* parameters point to two distinct abstract locations even if the method gets triggered by the same object. Even if this prohibits a precise pointer analysis of these particular parameters, it makes the analysis in general more usable since after including this information it is possible to find calls made using these objects, which would not be possible, if no type information at all is available for those parameters.

7.2.3 Calls to External Functions

Having calls to external functions on a path from the source of an parameter to its use in a cryptographic functions means that the information is needed how this function processes this parameter. This holds for both the pointer analysis and program slicing. In the case of pointer analysis it is needed to know, if something gets allocated, written to a pointer location or loaded from it, or if a pointer is returned that was passed to the function. For program slicing all modified values(the *DEF* set) and all referenced values (the *REF*) need to be defined.

To get a complete coverage for external functions called in iOS applications these properties need to be defined for all functions, which includes the methods of all classes derived from NSObject. However, this results in an enormous list of functions, of which only few are necessary to be defined for covering the functions in the scope of cryptographic functions.

7.2.3.1 Points-to Informations for External Functions

For every external function that needs to be handled constraints defining the following properties need to be defined:

- $\mathtt{v}_i \supseteq \{\mathtt{v}_j\}$: if $\mathtt{v}_i$ is a newly allocated object
- $\mathtt{v}_i \supseteq \mathtt{v}_j$: if $\mathtt{v}_i$ is passed to the function and $\mathtt{v}_j$ gets returned and points to the same location
- $\mathtt{v}_i \supseteq *\mathtt{v}_j$: if a pointer $\mathtt{v}_i$ is written to the location of $\mathtt{v}_j$
- $*\mathtt{v}_i \supseteq \mathtt{v}_j$: if $\mathtt{v}_i$ is loaded from a location $\mathtt{v}_j$ points to

7.2.3.2 Program Slicing

Similar to the information about external functions that is needed for pointer analysis the used external functions need to be considered for static slicing as well. The required information is about the referenced and modified variables in the scope of an external function. This means that all parameters that are passed to a function and the return values need to be added to the *DEF* and *REF* sets of an call instruction. And further all referenced values that have to be considered for backtracking need to be defined as well.

7.2.3.3 Extending LLVM Tablegen for External Functions

LLVM Tablegen[5] is a tool that helps developing and maintaining domain specific knowledge. For specifying information about external functions the Tablegen tool was extended to help achieving this. All functions are defined as records that contain the information about both points-to information and the information for program slicing. A single function may also be defined in multiple records. This allows the definition of records for multiple functions with the same behavior, regarding points-to information or program slicing, to be grouped together in the Tablegen definition and if the record of members of this group needs to be extended, another record may be added for this function (an example for this is shown in Listing 7.6).

[5] `http://llvm.org/docs/TableGen/`

The most important added Tablegen classes that are provided for defining external functions are:

- **Function<*FunctionName*, list<*PtsTo*>, list<*Slicing*> >**: creates a record for the function with the name *FunctionName* and the elements of the passed lists have to contain elements of the classes defined below
- **Pre<*Reg*>**: the set of values a register can have before the call instruction
- **Post<*Reg*>**: the set of values a register can have after the call instruction
- **Pts<Pre/Post<*Reg*> >**: all locations the values of the specified register can point to
- Classes for points-to records
  - **Copy<*From*, *To*>**: adds the constraint $v_{From} \supseteq v_{To}$
  - **Load<*From*, *To*>**: adds the constraint $*v_{From} \supseteq v_{To}$
  - **Store<*From*, *To*>**: adds the constraint $v_{From} \supseteq *v_{To}$
  - **Alloc<*Reg*, *Type*>**: adds the constraint $v_{Reg} \supseteq \{v_{Reg}\}$ and annotates the location $\{v_{Reg}\}$ with the type *Type*
- Classes for slicing records
  - **Def<*Reg*>**: adds the specified register to the *DEF* set
  - **Ref<*Reg*>**: adds the specified register to the *REF* set
  - **Ref1<*Reg*>**: adds the specified register to the *REF* set and further this class defines that a value should be backtracked, if this statement is included in the slice

Tablegen uses the records and generates C++ code, which is called while points-to information in calculated, and later when the *DEF* and *REF* sets are computed.

```
foreach f = ["objc_retain", "objc_release"] in {
    def : Function<f, [], [Ref<Pts<Pre<"X0">>]>;
}

def : Function<"objc_retain", [Copy<Pre<"X0">, Post<"X0">>], []>;
```

Listing 7.6: This example shows the definitions for *objc_retain* and *objc_release* where both get a reference to the object to retain or release passed as the parameter in register X0. While *objc_release* does not return anything *objc_retain* returns the passed object again and this, of course, points to the same location.

7.2.4 Backtracking Parameters and Rule Checking

Parameter backtracking and rule checking, as described in Chapter 5 and Chapter 6, are integrated into LLVMSlicer too since the information needed here is generated during program slicing. Each path starts at a single variable of a slicing criterion. With the information about the predecessors, which describe the predecessors that modify this parameter, created during slicing it is now possible to create paths to a parameters origin by simply traversing these predecessors.

- **No predecessor**: the path ends here
- **A single predecessor**: this predecessor gets added to the path and the path generation continues there

- **Multiple predecessors**: the current path has to be split up into multiple paths where to each one of the predecessors is added and each path is traversed separately

To verify, if a path meets the condition of the rules that have to be checked, each element of this path is tested against certain conditions defined by this rule. These conditions define either if a rule is fulfilled or if it is violated somehow. In both cases the path traversal does not need to be continued for this path. The rules themselves are provided using JSON definitions to allow an easy extensibility. It is also possible to group similar functions, as it is the case with functions of the *CCCryptorCreate()* functions as described in Section 6.2. An example for defining a rule is shown in Listing 7.7 that makes use of this groups of functions too.

```
  {
    "name": "No non random IV for encryption",
    "criterion": "IV-Cipher",
    "conditions": [
        {
            "type": "PRE",
            "criterion": "OP-Cipher",
            "name": "Encrypt",
            "conditions": [
                {
                    "type": "OK",
                    "conditionType": "ConstInt",
                    "equal": 0
                }
            ]
        },
        {
            "type": "OK",
            "calls": "SecureRandom"
        }

    ]
},

{
    "name": "IV-Cipher",
    "functions": [
        {
            "name": "CCCryptorCreate",
            "parameter": "X5"
        },
        {
            "name": "CCCryptorCreateFromData",
            "parameter": "X5"
        },
        {
            "name": "CCCrypt",
            "parameter": "X5"
        },
        {
            "name": "CCCryptorCreateWithMode",
            "parameter": "X4"
        }
    ]
}
```

```
{
    "name": "SecureRandom",
    "functions": [
        {
            "name": "CCRandomGenerateBytes",
            "parameter": "X0"
        },
        {
            "name": "SecRandomCopyBytes",
            "parameter": "X2"
        }
    ]
}
```

Listing 7.7: The rules for detecting cryptographic misuse are provided to LLVMSlicer using JSON. The described rule is, as the name states, the one that requires the IV to be random for all calls to the functions defined in *IV-Cipher* and shows how to define multiple conditions for a single rule. First it requires the first parameter, stored in register *X0*, to be zero, which tells the function to perform encryption. Second, the path needs to contain a call to one of the functions that provide a secure random number, which are described by the identifier *SecureRandom*.

Chapter 8

Limitations

The presented framework has also some limitations that may prevent a successful analysis of an iOS application or produce incomplete results.

8.1 Swift

Swift[1] is a programming language developed by Apple, which can be used for developing iOS applications. The biggest difference, concerning the analysis done in this work, is that methods are not only called by sending messages, but by using dynamic dispatching with virtual method tables and direct calls as well. While messages and direct calls are supported by the analysis framework, virtual method tables are not. Without supporting this the generated call graph will be incomplete and analyzing dataflow between different function would not be possible at all.

Handling these calls would be the initial step for supporting the Swift language, but similar to Objective-C other language specific features have to be handled separately. Adding all those features for being able to perform an analysis on many applications written in Swift would increase the workload that exceeds the time available for this work.

8.2 Polymorphism

In most cases polymorphism will not cause any issues, but in certain scenarios the computed call graph will contain spurious edges. The following example sketches such a scenario where the framework follows impossible function calls.

```
@class BaseClass
{
  - (void) foo;
  - (void) foobar;
}

@class ClassA : BaseClass
{}

@class ClassB : BaseClass
{
  - (void) bar;
```

[1] https://swift.org/

```
}
```

Listing 8.1: Class definitions for the polymorphism example

```
void someFunction()
{
  BaseClass *object = nil;
  if (someCondition) {
    object = [[ClassA alloc] init]; // $\{loc_A\}$
  } else {
    object = [[ClassB alloc] init]; // $\{loc_B\}$
  }
  [object foo]; // $\{loc_A, loc_B\}$
}
```

Listing 8.2: A scenario where the pointer analysis fails due to polymorphism

Clearly, *object* is either of type *ClassA* or *ClassB* depending on *someCondition*. Knowing the correct type is only possible inside the branches after the allocation statement. After this if-else block the variable *object* has a points-to set that contains both abstract location loc_A and loc_B since the used pointer analysis is a flow insensitive one and therefore it can not be differentiated, which path was taken to reach this statement.

Having the points-to set of *object* with both abstract locations is not a wrong points-to set, but if an instance method is called using this variable both types have to be considered. This means for the example code above that the call to *foo* has to be handled for both classes *ClassA* and *ClassB*.

The methods of *ClassA* and *ClassB* are implemented as described below.

```
@implementation ClassA {
  - (void) foo
  {
      [self foo]; // $\{loc_A, loc_B\}$
  }
}

@implementation ClassB
{
  - (void) foo
  {
      [self bar]; // $\{loc_A, loc_B\}$
  }

  - (void) bar
  {
      [self foobar]; // $\{loc_A, loc_B\}$
  }
}
```

Listing 8.3: Implementations of the classes defined in Listing 8.1

Since *self* is always represents the instance object of which the method was called and this variable is passed as a function parameter (this happens implicitly and does not have to be stated anywhere). This results in having a constraint $self \supseteq object$, which means that *self* will point to all elements of the points-to set of *object*. These points-to sets will then contain both abstract location with the types *ClassA* and *ClassB*, which can not be correct in this case, since *self* will only point to an instance of the current methods class or one of its subclasses, but never to a sibling like it is the case in this example. In Figure 8.1 it is shown the generated call graph for this example. There are three wrong edges in this graph where the edge $-[ClassB\ foo] \rightarrow -[ClassA\ bar]$ is actually not really added to the call graph since the function *-[ClassA bar]* does not exist. Since this edge can not be added, it will not cause any issues later. However, the other two wrong edges have an impact on later analysis as the falsely called methods exist.

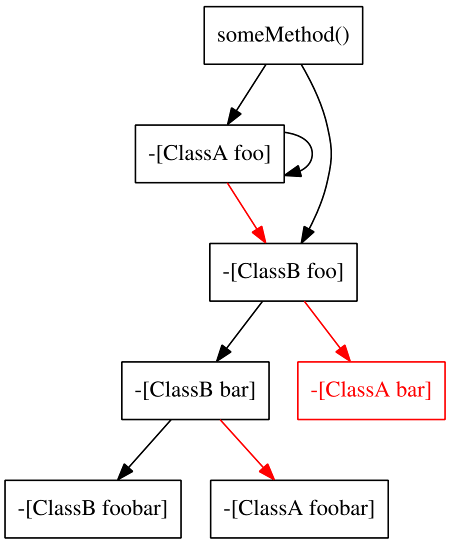


Figure 8.1: The call graph is created from the example code in Section 8.2. Edges displayed in red to *-[ClassA foobar]* and *-[ClassA bar]* are the ones that were added falsely. Not only the edge to *-[ClassA bar]* must not be added but also this method does not exist in this class.

Solving this issue requires a flow sensitive pointer analysis, which is expensive for large programs that have to be processes in this work and the existing basic implementation of the pointer analysis is a flow insensitive. Also, this problem does not occur in many cases since both allocations have to be assigned to the same variable as shown in example. And even if wrong edges exist in the call graph, during backtracking only paths are considered where the parameters of interest are modified. However, it may happen that such constructs are present in the relevant parts of the analysis and this will produce wrong results.

8.3 C Arrays

The pointer analysis, described in Section 4.2, is a field insensitive analysis. This means that different fields of an array or of a struct can not be distinguished by the pointer analysis.

Having something simple like the following produces an imprecise analysis result:

```
char *array = ...;
array[0] = 'a';
array[1] = 'b';
```

While the first statement, which accesses the array, refers directly to the address where the pointer points to the second statement has to add an offset to the pointer address before accessing the location. Using the base pointer in the second case and set the points to set to the same set as the base pointer has does not work either since this array or struct may be stored on the stack as well. All stack stored variables are accessed using the stack pointer and the stack pointer is actually a base pointer to which an offset is added. If now an array or struct is stored on the stack and the stack pointer is taken as base pointer for all elements of this array or struct all other local stack variables would be included in this points-to set as well. This happens because local variables are also accessed by adding an offset to the stack pointer and they can only be distinguished from each other by the offset value. Omitting this offset value by only considering the stack pointer would result in a single points-to set for the whole stack frame of a function instead of an own points-to set for each local variable.

To solve this issue the pointer analysis has to be changed to a field sensitive approach, but in this work the implementation of the pointer analysis is based on an already existing solution[2], which uses Andersen's [2] field insensitive approach. However, for the goal of backtracking parameters it is usually not necessary to distinguish between the single element of arrays since they are passed to functions using only the base pointer.

8.4 User Interface Elements

When it comes to creating an user interface two options are supported: either it is created directly in code or the Interface Builder[3] is used, which creates *.nib* files that define the user interface and get parsed by the application. The user interface also requires to define how events are handled and this is the part where the analysis is not able to succeed.

Event handling is again possible via two different ways. First, it can be done by implementing and setting delegates. A delegate has to implement a specific protocol, depending on the user interface element, and these protocol methods are triggered by user interface events. The second way is done by setting event listeners for selected events. Both can be done directly in the source code of the application or the Interface Builder. However, if the interface builder is used to define actions, without using delegates, these informations are retrieved from *.nib* files, but the analysis framework uses only information stored in the binary file and nothing more. This causes the problem that the triggering user interface element is not known to the analysis framework.

A scenario where this is an issue is if, for example a password field defines an action for an *UIControlEventEditingChanged* event that triggers the function *textFieldChanged:* in Listing 8.4. In this case the type of *sender* is unknown since the function is called by an external source, which passes this parameter to the function. Further, if this type is unknown, the call *[sender text]* is not recognized by the analysis framework, thus the variable *password* has no type as well (i.e. it points nowhere) and none subsequent uses of the variable *password* can be backtracked to this statement. Having a password and taking it as input to the function *CCKeyDerivationPBKDF()* for generating an encryption key will not lead to the initial statement where the text is retrieved from the *UITextField*.

If the delegate method of Listing 8.4 was called for the same task, the backtracked path will end at the statement where the text is fetched from the *UITextField* since this method is defined in a protocol the function header with its types is known by using the information about protocol methods stored in the binary as described in Section 3.3.3.

```
//Set by the Interface Builder
- (IBAction)textFieldChanged:(UITextField *)sender
{
  NSString *password = [sender text];
  ...
}

//Delegate method
- (void)textFieldDidEndEditing:(UITextField *)textField
```

[2]https://github.com/grievejia/andersen

[3]https://developer.apple.com/xcode/interface-builder/

```
{
  NSString *password = [textField text];
  ...
}
```

Listing 8.4: Two implementations of methods that have the same task, but one is triggered as a delegate function and the other one is set in the Interface Builder

8.5 *nil* Objects

In Objective-C *nil* represents an object pointer, which points to nothing. While other languages, like C++, would cause an error, if a method was called on an object that points to nothing, Objective-C allows this and does cause an error.

```
NSData *nilData = nil;
const void *bytes = [nilData bytes];
NSData *nilCopy = [NSData dataWithBytesNoCopy: bytes length:[nilData
   length]];
```

Listing 8.5: A valid Objective-C example of calling methods of a *nil* object

The code in Listing 8.5 is executed without error and the intended behaviour is that *bytes* points to NULL. The variable *nilCopy* will actually be an valid object (it will not point to *nil*), but the buffer of this object will again point to NULL since this buffer does not get copied or modified.

The problems are introduced by having an incomplete call graph. In the code snippet above the call *[nilData bytes]* is the one that is missing in the analysis since *nilData* is never allocated as *NSData* object and thus it has no type, which is needed for adding call graph edges. It is now unknown that the variable *bytes* points to NULL. Further, if *bytes* is used somewhere, it can not be backtracked to its origin since this call graph edge is missing.

Part III

Evaluation

Chapter 9

Evaluation

In this chapter the implemented framework was tested on different applications to show that cryptographic misuse was found in most cases, if present. However, due to some limitations, which are discussed in Chapter 8, it was not possible to get correct results for all available applications.

First, in Section 9.1 the method how this evaluation was done is described. Section 9.2 discusses the outcome of running the analysis framework on open source applications and these results are verified by comparing it to the original source code. In Section 9.3, the results of running the analysis on applications, which were published on the App Store, are presented.

9.1 Method

The evaluation method was carried out in two steps: first, the developed framework was executed with open source applications as input and the delivered results were verified using the original Objective-C source code. The second part was done by executing the framework with applications downloaded from the App Store as input to the analysis.

Test Set The test set of applications consisted of ten applications that have been found on github[1] where five of them were password managers and the others were all from different categories.

Each of the applications was first used as input for the analysis framework to generate a report, which contained all paths that had been found for each rule specified in Section 6.3. These paths contained only those statements that either modified a specific parameter or call statements to functions that modified this value.

To verify correctness of a performed analysis by the framework presented in this work the original source code of the application was required for keeping the error-proneness in the manual verification process low compared to verifying the result based on the assembly code directly. For this manual inspection of the original source code Xcode provides a quite useful tool that generates call hierarchies for selected functions. Such an hierarchy is basically a subgraph of the call graph containing only the nodes from which the targeted function was reached.

Backtracking the individual parameters in the original source code was then done manually without information generated by Xcode or other tools. Usually the parameters of our interest were modified rarely after they had been initialized and had been passed to a cryptographic function so this task was rather simple.

[1] `http://github.com`

Apple Store Applications After the analysis was performed on the test set and the results were verified, the framework was then used to gain insight on a larger number of applications by taking “real world” applications that were published on the App Store. Those applications are usually closed source, which means for this analysis that the results could not be verified with the original source code. A manual verification of a larger number of applications was not feasible in the limited time of this work.

The generated reports, however, had to be checked manually since their content was only a list of paths, where each path was annotated as valid, if no rule had been violated, or invalid, if a clear rule violation had been found or if the analysis had found an incomplete path (i.e. this happened when functions never got called and the backtracked value was a parameter that has to be passed to this function).

Retrieving applications All of the applications that had been examined in this step were retrieved from the App Store. The downloaded binaries were always encrypted and the key required for decrypting was stored in a secure enclave on the iOS device. However, there are tools available (e.g. Clutch[2], which was used in this work) that are capable of decrypting iOS binaries on an jailbroken device. Since the App Store app on an iOS device does not offer any API, which allows doing this in an automated fashion, this had to be done manually.

Not every application requires encryption, so we had to focus on categories where the assumption could be made that such functions were called. The main group, of which applications had been considered, were password managers since there were numerous options available, and each of them should have used encryption at some point.

Additionally, not all applications could be handled by the analysis framework since we had the restriction of only being able to perform the analysis using binaries that had been compiled from Objective-C code as explained in Section 8.1.

9.2 Test Set - Open Source Applications

To verify the results delivered by the implemented framework open source applications, that used the *CommonCrypto* library, were checked, if the generated paths matched with those that could be found in the original source code. *Xcode*[3] is capable of generating call hierarchies that show all functions that call a specified function. However, the important part was not to verify the call hierarchies, but the parameters that reached a given function and their origin.

9.2.1 Damn Vulnerable iOS App

The Damn Vulnerable iOS App (DVIA) is actually no app for the average user since it was only made to provide an application for penetration testing and for this purpose some common mistakes were made intentionally. DVIA implemented several examples of common insecure implementations and an example for *broken cryptography* was also included there. Obviously, this means that something had to be found.

The application also used the *Realm* database, which used encryption too, but parts of this framework were closed source and analysis results for these parts of the code are not discussed since they could not be verified. All symmetric cryptography related work was done by the framework *RNCryptor*[4].

Initialization vector and salt value The analysis reported multiple paths that violated rules regarding the initialization vector and salt value. Both of these values should have been generated by a

[2]https://github.com/KJCracks/Clutch

[3]`https://developer.apple.com/xcode/`

[4]`https://github.com/RNCryptor/RNCryptor`

secure random number generator and *RNCryptor* provided the helper function *+[RNCryptor randomDataOfLength:]* to do this. For each invocation of this function our analysis reported three separate paths, of which only one was automatically considered secure since *SecRandomCopyBytes()* was used:

```
NSMutableData *data = [NSMutableData dataWithLength:length];
if (SecRandomCopyBytes != NULL) {
  result = SecRandomCopyBytes(NULL, length, data.mutableBytes);
}
else {
  result = RN_SecRandomCopyBytes(NULL, length, data.mutableBytes);
}
```

Listing 9.1: The helper method of RNCryptor to generate random data

+[RNCryptor randomDataOfLength:] tried to call the function *SecRandomCopyBytes()*, which always works on iOS devices. The *else* condition therefore was unnecessary and was never executed, but it was defined and therefore it had been found by the analysis. Why *RN_SecRandomCopyBytes()* caused an error can be seen from its implementation as shown below.

```
dispatch_once(&onceToken, ^{
  kSecRandomFD = open("/dev/random", O_RDONLY);
});
if (kSecRandomFD < 0)
  return -1;
while (count) {
  size_t bytes_read = read(kSecRandomFD, bytes, count);
  ...
}
```

Listing 9.2: Implementation of RN_SecRandomCopyBytes()

It used */dev/random* as source for random data and the analysis found the initialization of this file descriptor. There were now two possible paths where *read* was not called and if this did not happen, the data returned from *+[RNCryptor randomDataOfLength:]* would be the initial value of the *data* variable, which only contained zero values. The first of these paths was taken, if the file descriptor of */dev/random* had not been successfully initialized and the second path was followed, if *count* was zero.

For analysis results this means that whenever random data was required and had been provided by this function, two errors, for every path that reaches this function, were reported:

1. */dev/random* was not set as a secure origin of random data
2. the *data* variable was initialized with zeros and may be returned without having been modified

Password based encryption This was the part where the weakness had been introduced by defining a hard coded password for encryption. The analysis reported paths to two different origins where both were found in the original code. The first one ended at a call to *-[UITextField text]*, which did not cause any rule violation.

However, the other instruction, where the paths ended, corresponded to the following instruction in the original source code:

```
NSData *encryptedData = [RNEncryptor encryptData:data
                                    withSettings:
                                       kRNCryptorAES256Settings
                                        password:@"Secret-Key"
```

```
                                                   error:&error];
```

Listing 9.3: Hardcoded password in the Damn Vulnerable iOS App

The hard coded password had been passed directly to the *RNCryptor* framework and this was also recognized by the analysis. In this instruction was also a the parameter *kRNCryptorAES256Settings*, which represented the default settings for *RNCryptor*. It contained a setting of 10000 rounds for PBKDF and this was recognized correctly by the analysis. The salt value was generated by *RNCryptor* using the earlier described function *+[RNCryptor randomDataOfLength:]* with the same analysis results.

Cipher Mode The encryption mode was set by the settings passed to the encryption function and the default settings did not use ECB mode. And since this was again a simple integer value like the iteration count it was found correctly by the analysis.

-[RNOpenSSLEncryptor initWithSettings:password:handler:] This was a function that was never called within the app, but violations were reported by the analysis since it was defined within the binary. This was caused by the way how the encryption key and initialization vector were generated after this method was called. In both cases the helper method *+[RNCryptor randomDataOfLength:]* was involved and, as mentioned earlier, each of these values induced an error in the report of the analysis.

The encryption key was also derived using a password, as the name of this function implied, but since it never was called in the application this parameter had no origin that could be found.

Spurious paths caused by context insensitivity There were multiple spurious paths reported that were caused by the context insensitivity of the underlying pointer analysis and polymorphism. The cipher operations were all handled by a single class *RNCryptor* that had multiple child classes, which were responsible for configuring the cipher to perform the cryptographic operations. In Figure 9.1 two possible paths are shown by the callgraph how this *CCCryptorCreate()* function may had been reached. The one that used functions of the class *RNDecryptor* was responsible for decrypting data and on the contrary the one with functions of *RNEncryptor* encrypts data. The spurious path was created when the origin of the decryption key should be found. This key was stored in an instance variable *encryptionKey* and the path showed where this instance variable was set. In Figure 9.1 the green arrows show the correct path, which was found by the analysis too, but on the bottom in the function *+[RNCryptor synchronousResultForCryptor:data:error:]* the current path was split into two paths of which one ended up in the wrong location. This was caused by the fact that both *+[RNEncryptor encryptData:withSettings:password:error:]* and *+[RNDecryptor decryptData:withPassword:error:]* called the function *+[RNCryptor synchronousResultForCryptor:data:error:]* and without having the calling context it was not possible to determine the correct callee thus both paths had to be followed since both of these functions had set the instance variable *encryptionKey*.

Since this path was only taken for a decrypt operation it was redundant anyway, but it is included to show a weakness of the general approach taken, by using a context insensitive pointer analysis.

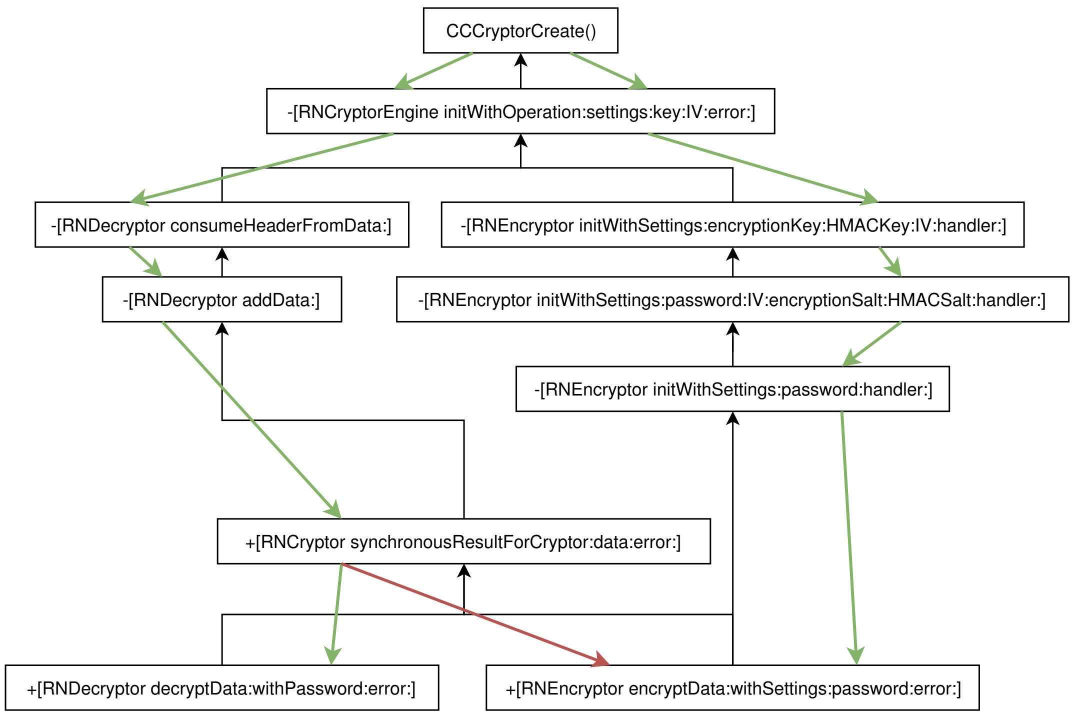

CCCryptorCreate()
-[RNCryptorEngine initWithOperation:settings:key:IV:error:]
-[RNDecryptor consumeHeaderFromData:]
-[RNEncryptor initWithSettings:encryptionKey:HMACKey:IV:handler:]
-[RNDecryptor addData:]
-[RNEncryptor initWithSettings:password:IV:encryptionSalt:HMACSalt:handler:]
-[RNEncryptor initWithSettings:password:handler:]
+[RNCryptor synchronousResultForCryptor:data:error:]
+[RNDecryptor decryptData:withPassword:error:]
+[RNEncryptor encryptData:withSettings:password:error:]

Figure 9.1: A small abstract of the call graph to show paths that may had been taken to initialize a cipher. The downward arrows show the paths that were taken during backtracking, where the edge from *+[RNCryptor synchronousResultForCryptor:data:error]* to *+[RNEncryptor encryptData:withSettings:password:error:]* was an invalid one caused by context insensitivity.

9.2.2 PA55 NYAPS

PA55 NYAPS[5] is a password manager that stored user credentials using a "master password". Clearly there should have been encryption involved in this process.

Random IV for encryption The analysis reported two different paths, where one of them had not violated the rule since it used data generated by *SecRandomCopyBytes()* as initialization vector and this was verified by the manual inspection of the code.

However, the other reported path stated that a *NSMutableData* object was initialized with a pointer to data that was set to zero. This matched to the following code in the original source code and hence this path was a correctly found one:

```
char zerobytes [kCCKeySizeAES128];
bzero(zerobytes, sizeof(zerobytes));
_iv = [NSMutableData dataWithBytes:zerobytes length:kCCKeySizeAES128];
CCCryptorStatus  create = CCCryptorCreateWithMode(kCCEncrypt,
                           kCCModeCTR, kCCAlgorithmAES, ccNoPadding,
                           _iv.bytes, _aesKey.bytes, _aesKey.length,
                           NULL, 0, 0, kCCModeOptionCTR_BE, &_cryptor)
                              ;
```

Listing 9.4: Initializing the initialization vector with all zeros

Calling *bzero()* sets all bytes of *zerobytes* to zero so no other values were used as initialization vector.

[5]https://github.com/pa55/pa55nyaps-ios

Creating an object was actually not even necessary in this case. A NULL pointer could be passed directly to this function and the result would have been the same.

Constant encryption keys The analysis reported several paths that originated in accessing constant data and passing it to an encryption function.

First there was a path with constant data to the function that was used to encrypt data. In detail this part of the code was reported:

```
NSString *phrase = @"my test master secret sentence";
...
printf("Password (%d chars): %s\n", i,
  [[NYAPSCore generateAESDRBGPasswordWithPhrase:phrase
                                          hint:hint
                                        length:i
                               userPreferences:userPreferences
                                   userCharset:nil] UTF8String]);
```

Listing 9.5: Constant string as password in PA55 NYAPS

The key was generated using *CCKeyDerivationPBKDF()*, but the input to this function was constant.

The other reported paths that caused an error corresponded to instructions of the form:

```
[[AESDRBG alloc] initWithRawSeed:[@"0123456789abcdef" dataUsingEncoding
   :NSUTF8StringEncoding]];
```

Listing 9.6: Constant data as encryption key in PA55 NYAPS

The code for these errors was located in a function called *+[NYAPSCore testNYAPSCore]*, which implied that this was only test code. But still it was present in the binary and it had been found by the analysis.

And a correct path was found as well that originated in a function that retrieved the text from an *UITextField*, which then was passed to the *CCKeyDerivationPBKDF()* function.

ECB Mode In this binary no cipher with ECB mode was used. The parameter, which defined the mode, was always passed directly to a function for creating a cipher and hence that there was no possibility for false results of the analysis.

PBKDF iteration count There was a call to *CCKeyDerivationPBKDF()* at two locations in the code. Both were wrapper functions that had a parameter *rounds* that represented the iteration count.

While one function was called from another function that passed a constant integer with a value of 25000, the other function did not get called from any function and thus this value was never set, which led to an error.

Constant salt for PBE The found path of the salt value to its origin did not violate any rule and ended in a call to *SecRandomCopyBytes()* in the same helper function as the correct path of the initialization vector. And this path was also verified using the original source code.

9.2.3 OpenSafe

This is another password manager that used password based encryption to secure credentials. For the analysis the source of the password for deriving the key caused problems, which are described in detail later in this section. Nevertheless, for all remaining rules all paths were found by the analysis and could be verified.

Initialization vector and encryption mode The analysis reported two paths, of which one violated the rule regarding the initialization vector as it was set to NULL directly when *CCCrypt()* was called. The original call to *CCCrypt()* validated this result since the initialization vector was directly set there. Further, the options only specified *kCCOptionPKCS7Padding* so the default CBC mode was used and the analysis found this path as well. These two paths are described together because they were both defined by the same instruction in the original code:

```
CCCrypt(kCCEncrypt,
  kCCAlgorithmAES128,
  kCCOptionPKCS7Padding,
  key.bytes,
  key.length,
  NULL
  data.bytes,
  data.length,
  encData.mutableBytes,
  encData.length,
  &numBytesEncrypted);
```

Listing 9.7: Setting the initialization vector to NULL in OpenSafe

Salt The rule for checking the salt value returned multiple paths where some ended at calling *SecRandomCopyBytes()*, but others had their final instruction in the function *+[Utilities base64Decode:]*. This was caused by the following code:

```
NSData* salt = [Utilities base64Decode:[saltKC objectForKey:(__bridge
   id)kSecValueData]];
if ([salt length] == 0) {
  salt = [self generateRandomData];
  ...
}
```

Listing 9.8: OpenSafe only creates a random salt value once and reuses it afterwards

A salt value was generated randomly using *SecRandomCopyBytes()*, but only once and all further encryptions reused this value.

Password for PBE - missed path The path for finding the password that got passed to *CCKeyDerivationPBKDF()* ended in *-[CCCryptHelper PBKDF2DeriveKeyFromPassword:withSalt:]*, but did not show where the password really comes from.

In the original code this function was called with the statement below, which also passed the password. However, the password was retrieved from an *UIAlertView* and this call was not recognized by the analysis.

```
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(
   NSInteger)buttonIndex {
  ...
  [self PBKDF2DeriveKeyFromPassword:[[alertView textFieldAtIndex:0] text]
                          withSalt:salt];
```

```
    ...
}
```

Listing 9.9: Retrieving a password from an UIAlertView in OpenSafe

The method *alertView:clickedButtonAtIndex:* is defined by the *UIAlertViewDelegate* protocol, but the issue was that the class *CCCryptHelper* did not conform to this protocol, it only implemented this particular method with the same name and signature as it is defined in the protocol *UIAlertViewDelegate*. Usually a delegate has to implement the corresponding delegate protocol, but in the case of *UIAlertView* this is not required and was not done (for other user interface classes this is usually not allowed). If the protocol was implemented by this class, the type of the parameter *alertView* would have been known, since it is defined by the *UIAlertViewDelegate*.

9.2.4 MiniKeePass

This is another open source password manager[6] that used password based encryption to protect a users credentials. The most different aspect in this application, compared to the other password managers presented, was that the key derivation function[7] was not provided by the *CommonCrypto* library. Here the key was derived by a hash function and symmetric encryption for multiple rounds.

Encryption Key As already mentioned, MiniKeePass used its own key derivation function and this was not supported by the analysis framework. The only information we got was that calls to *CC_SHA256()* and *CCCryptorUpdate()* were made, but not how often or how a salt value was added. This means that we did not get a usable result and such custom implementations will always return similar results.

Initialization vector The encryption function was called twice, once for deriving an encryption key and the second one was made when actual data was encrypted. For deriving a key no IV was used, hence this parameter was set to NULL and this was also reported by the analysis.

For the actual encryption an IV was used. This path ended in the function *+[Utils randomBytes:]* by calling *arc4random()* and since this function also accessed */dev/random*, like *SecRandomCopyBytes()* does, this was considered to be secure.

Encryption Mode The use of ECB mode was not reported by the analysis and the manual inspection verified this result as the default options were used.

Missed Paths - Creating a new database file When a new database was created, a object of type *KdbPassword* was initialized with a password string and this object was later used to derive a encryption key using the method *createFinalKeyForVersion:masterSeed:transformSeed:rounds:*, which accessed this string password. The path that should have been found was the one from the key derivation method to the origin of the string password, a call to the *text* method of a *UITextField* in *-[FilesViewController createNewDatabase:]*.

```
    NSString *password1 = newKdbViewController.passwordTextField1.text;
```

Listing 9.10: Retrieving the password from an UITextField in MiniKeePass

The call *newKdbViewController.passwordTextField1* was not found in the analysis and the reason was located in the callee of this method:

[6] https://github.com/MiniKeePass/MiniKeePass

[7] http://keepass.info/help/base/security.html

```
newKdbViewController.donePressed = ^(FormViewController *
    formViewController) {
  [self createNewDatabase:(NewKdbViewController *)formViewController];
};
```

Listing 9.11: MiniKeePass did set an instance variable to a block value, which was not detected by the analysis

This block was called when a new database should have been created. As described in Section 4.2.6.3 blocks need to be identified as such by calling a block related function of the Objective-C runtime library, otherwise it could not be said, if this was a block or not. Such a call was missing caused by the circumstance that this was an instance variable, which appeared to be handled different to ordinary block variables.

9.2.5 Account Manager

This password manager[8] yielded in a quite brief analysis report since all paths originated from a single instruction where all the encryption of this application was handled. This statement corresponded to the call to *CCCrypt()* in the snippet below. The paths for the initialization vector and encryption mode were rather short since these values were directly set at the function call and thus they were easily verified.

The paths to the encryption keys originated always contained a call to *-[NSString getCString:maxLength:encoding:]* and the result of this call was then passed to *CCCrypt()* and this was validated by the code below. Having this call on all paths already hinted that the encryption key was no random data, but a string.

```
[key getCString:keyPtr maxLength:sizeof(keyPtr) encoding:
    NSUTF8StringEncoding];
...
CCCryptorStatus cryptStatus = CCCrypt(kCCEncrypt, kCCAlgorithmAES128,
                                      kCCOptionPKCS7Padding,
                                      keyPtr, kCCKeySizeAES256,
                                      NULL,
                                      [self bytes], dataLength,
                                      buffer, bufferSize,
                                      &numBytesEncrypted);
```

Listing 9.12: A pointer to a C-string was used as encryption key

Both paths ended at similar statements in different functions, where both of these statements called the function *-[NSUserDefaults objectForKey:]*, which corresponded to this statement in the original code:

```
[data aes256EncryptWithKey:[[NSUserDefaults standardUserDefaults]
    objectForKey:@"aes256key"]];
```

Listing 9.13: Retrieving the password from NSUserDefaults

With this statement the key was retrieved from the *NSUserDefaults* system. The idea of *NSUserDefaults* is to provide a simple interface for storing application settings, but it is not suited for sensitive data at all. It is saved in plaintext inside the container of an application and may be retrieved from there.

While the automatic analysis was not able to go further since *NSUserDefaults* works like a *NSDictionary* and this was not supported by the framework, we continued manually to find the ori-

[8]`https://github.com/BaronNeron/Account-Manager`

gin of the value stored for the key in *NSUserDefaults*. This value was created in the *application:didFinishLaunchingWithOptions:* function of the application:

```
[[NSUserDefaults standardUserDefaults] setObject:[RandomStringHelper
    randomStringWithLength:32] forKey:@"aes256key"];
```

This statement was executed exactly once after the applications was installed and first started. The function *-[RandomStringHelper randomStringWithLength:]* generated a random string consisting of upper- and lower-case characters and numbers. A key generated with this function had obviously less entropy than one generated using a secure random number generator and thus was less secure. Nevertheless, even if the key was truly random, the encrypted data would have been easily decryptable by extracting the key from *NSUserDefaults*.

Additionally to the password stored in *NSUserDefaults* the initialization vector was set to NULL as shown in the statement above where *CCCrypt()* was called. This rule violation was reported by the analysis without any spurious paths as it was directly set at calling the function.

The encryption mode was also set directly at the call statement and the default mode CBC was used, which was also correctly reported by the analysis.

The remaining rules would only have been checked, if password based encryption was used and this was not the case for this application.

9.2.6 Encrypted Chat

Encrypted Chat[9] is a messaging application that stated to use AES-256 encryption for transmitting messages. While the messages indeed were encrypted, the following analysis results showed that this encryption was ineffective.

Static password for PBKDF The analysis reported the use of a constant string as input for *CCKeyDerivationPBKDF()*. As the application used the *RNCryptor* framework it only called the function *+[RNEncryptor encryptData:withSettings:password:error:]* and everything else was done by the framework.

```
[RNEncryptor encryptData:data
            withSettings:kRNCryptorAES256Settings
                password:PasswordGet(groupId)
                   error:&error];
```

Listing 9.14: Encrypted Chat called a RNCryptor function for encrypting data with a password

The *password* parameter was set by calling a the function *PasswordGet()* using a *groupId*, but this function was implemented in the following way:

```
NSString* PasswordGet(NSString *groupId)
{
  return @"0123456789";
}
```

Listing 9.15: The *PasswordGet()* of Encrypted Chat function returned a constant password.

Only a constant string was returned and used as a password for all encryption operations and the passed *groupId* was never accessed so the password was the same for all calls to this function.

[9] https://github.com/relatedcode/EncryptedChat

Remaining parts Since the *RNCryptor* framework was used the parameters for encryption were set using a settings struct object. This application again used the provided default settings, which were already mentioned in Section 9.2.1. The initialization vector was also generated by *RNCryptor* and the same results were generated in this case as in Section 9.2.1.

9.2.7 Tiny Password

The application Tiny Password[10] is again a password manager that used the *CommonCrypto* library for performing cryptographic operations. The main difference, regarding the analysis, compared to other applications discussed in this chapter, was that the analysis generated a massive amount of erroneous paths for Tiny Password caused by a large number of statements that ended up at an cryptographic function.

Encryption Key Running the analysis on this app resulted in reporting more than 7000 paths for the *key* parameter of the *CCCryptorCreate()* function (this implied both encryption keys and decryption keys since the same function had to be called for both operations). This vast amount of paths was not caused by a flawed analysis, but by the heavy use of the function *+[AESCrypt data_encrypt:password:]*, which then called the *CCCryptorCreate()* function. For none of these paths a key generated by a random number generator or by the provided PBKDF function was found, so all paths were considered to be insecure. Validating all these paths would have been very tedious and it even may not have been necessary since the paths were very similar. Similar means that many paths have their origin and destination at the same statements. We considered these similar paths, with same origin and destination, to be equal and discarded duplicate paths, which ended up in having 11 paths.

All those paths reached the function *CCCryptorCreate()* via the earlier mentioned function *+[AE-SCrypt data_encrypt:password:]* and, as the name already suggested, a password and no random key was applied for encrypting data. Since there was no call to *CCKeyDerivationPBKDF()* in this application, a custom key derivation function had to perform this task. The analysis method of the framework in this work was only capable of checking the key derivation function provided by *CommonCrypto* and not custom functions for this task. However, in this case the analysis results showed that a single call of *CC_SHA256()* was used to derive the key that was passed to the encrypting function. The manual inspection of the source code verified this result in the following statement:

```
NSData *encryptedData = [message AES256EncryptedDataUsingKey:
                        [[password dataUsingEncoding:
                           NSUTF8StringEncoding]
                        SHA256Hash] error:nil];
```

Listing 9.16: The key derivation of Tiny Password

With this statement the encryption key was created without performing multiple iterations of calling a hash function or adding a salt value.

Additionally to the way of generating keys constant password values were found in this application. The class *DropboxSyncHelper* used an instance variable of the *AppDelegate* called *privatePassword*, which was set only once after starting the application and this initial value was a constant string:

```
#define DropboxPassword @"*xiu@yue#1-(blue9tags/-/drop8box)-%&9-*"
...
privatePassword = DropboxPassword;
```

Listing 9.17: Setting a password to a constant value in Tiny Password

[10] https://github.com/HuiWang002/TinyPassword

The other password values were retrieved via instance variables as well, but in this cases they were not set only once when starting the application and this means that we could not be sure which value was then the actual password for the encryption. These values were set by retrieving text from UI elements in two different ways, where only one of these methods was detected by the framework. The first and detectable way was doing this by implementing a delegate protocol and accessing the UI elements using the parameters passed to these protocol methods. The types of these parameters were well known by parsing the class information stored in the binary as discussed in Section 3.3.3.

The other approach taken was setting event handlers for UI elements with the Interface Builder[11], which caused the problem of not knowing the type of the UI element that triggered the event as described in Section 8.4. The code snippet below shows how the text was retrieved from the UI element and this function was triggered by an UI event that was defined using the Interface Builder and there were no instructions present in the binary that defined this behaviour:

```
- (IBAction)textFieldEditingChanged:(UITextField *)sender
{
  self.inputStr = sender.text;
  [self next];
}
```

Listing 9.18: Accessing a *UITextFields* text value in a method set by the Interface Builder

As explained in Section 8.4 this type of events was not supported, thus the type of the parameter *sender* could not be determined and the instance variable *inputStr*, which was later part of the password, had no value.

Initialization Vector The analysis report for the initialization vector did not contain a usable result since a message was sent to an object with an unknown type.

```
    CCCryptorCreate( kCCEncrypt, algorithm, options,
                     [keyData bytes], [keyData length],
                     [ivData bytes], &cryptor );
```

Listing 9.19: Call to the encryption function where the variable *ivData* points to *nil*

The initialization vector was defined by the variable *ivData*, which was passed as a parameter to the function. However, the function call to this function was the following:

```
[self dataEncryptedUsingAlgorithm: algorithm
                              key: key
             initializationVector: nil
                          options: options
                            error: error]
```

This led to the problem that the variable *ivData* was always NULL and therefore the type *NSData*, which should have been the type of *ivData* defined by the function header, was not found by the analysis since it never had been allocated. Having no type for *ivData* means that the call of the *bytes* function of this object was not found and prevented further backtracking.

Encryption Mode The encryption mode was reported to be always set to CBC mode, which was the default mode, and this result was confirmed by the manual inspection as well.

[11] https://developer.apple.com/xcode/interface-builder/

9.2.8 M - Mynigma

Mynigma[12] is a email client that encrypted and decrypted emails. The only used function, of the ones that are checked by the analysis, was *CCCrypt()* and it was called only by a single statement..

Encryption key The analysis did not report an error for this rule and the inspection verified this result. Keys for encrypting data were generated by *SecRandomCopyBytes()* in the function *+[AppleEncryptionWrapper generateNewAESSessionKeyData]*.

Initialization vector The reported path for the initialization vector ended at a call to *+[NSFileHandle fileHandleForReadingAtPath:]* and the path contained also the call for reading from this file *-[NSFileHandle readDataOfLength:]*. This obviously means that *SecRandomCopyBytes()* was not used, but the path for the file was */dev/random* and the original source code validated this path by this statement for generating this data:

```
NSData* initialVector = [[NSFileHandle fileHandleForReadingAtPath:@"/
    dev/random"] readDataOfLength:16];
```

Listing 9.20: Initializing a NSFileHandle object with a random number generator

This way of generating a random number actually provided the same level of randomness as *SecRandomCopyBytes()* does since it read data from */dev/random* as well.

Encryption Mode The path returned states that the default CBC mode was set and the manual inspection verified this as well.

9.2.9 SecurePhotos

This app claimed to encrypt and store photos securely. Like previously discussed applications, this one used again the *RNCryptor* framework to perform cryptographic operations.

Constant keys for PBKDF The reported paths ended at the call instruction that corresponded to the following instruction in the original code:

```
[RNEncryptor encryptData:imageData
              withSettings:kRNCryptorAES256Settings
                  password:@"A_SECRET_PASSWORD"
                     error:nil];
```

Listing 9.21: SecurePhotos passes a hard coded password to RNCryptor

Similar to previously found problems the constant password was again passed directly to the *RNCryptor* framework. The values for the initialization vector and salt were generated by the *RNCryptor* framework using the *+[RNCryptor randomDataOfLength:]* function, which caused the analysis to report the same problems as already described in Section 9.2.1.

The parameters for the iteration count of the PBKDF function and the encryption mode were again defined by the provided default settings stored in *kRNCryptorAES256Settings*, which did not violate any rules, and the analysis found those valid values as well.

[12] https://github.com/Mynigma/M

9.2.10 tiqr client

The *tiqr client*[13] is a sample application that implements the *tiqr*[14] authentication method, which requires AES-256 encryption during this authentication process.

Encryption key The analysis report stated that key buffer points to NULL. This was caused by the following statements in the source code:

```
[key getCString:keyBuffer maxLength:sizeof(keyBuffer) encoding:
   NSASCIIStringEncoding];
keyBuffer[0] = 0;
...
CCCryptorStatus result = CCCrypt(kCCEncrypt,
                                 kCCAlgorithmAES128,
                                 0,
                                 keyBuffer,
                                 kChosenCipherKeySize,
                                 initializationVector ?
                                   [initializationVector bytes] : NULL,
                                 [data bytes],
                                 [data length],
                                 buffer,
                                 bufferSize,
                                 &numBytesEncrypted);
```

Listing 9.22: *tiqr* did set the first byte of the key buffer to zero

The variable *keyBuffer* served as encryption key, but the first byte of this data was set to zero and since the points to information refers to this exact location, this modification was considered to affect all data that was accessed with this pointer. On the contrary, if any other part than the first one would have been modified, this modification would not be considered during the analysis.

The analysis failed to find the origin of the *key* variable, whichs data was mostly used as encryption key. If this origin would have been found no rule violation would be found since it was derived using the *CCKeyDerivationPBKDF()* function with correct parameters.

Initialization Vector The snippet above shows the only statement where an initialization vector was required for encryption. Clearly, at least two paths had to be found for both values in the ternary operator. The NULL value caused, of course, a warning and this was found by the analysis framework as well as the paths for the other case of the ternary operator. There were multiple different paths that all ended up in the same function *-[SecretService generateSecret]*, which called *SecRandomCopyBytes()* to create random data for the initialization vector.

ECB Mode Again, the snippet shown above shows the only call to *CCCrypt()* where encryption was done and the third parameter there, which was set to zero, configured the operation with its default options, which included using CBC mode. Such paths were straight forward to be found by the analysis framework since no branches or calls were performed.

[13]https://github.com/SURFnet/tiqr-client-ios
[14]https://tiqr.org/

9.2.11 Summary

Analyzing open source applications with the implemented framework helped to show the weaknesses of the approach in this work. While most of the actual rule violations were found, some cases could not be handled correctly. First there was the problem of having a context insensitive pointer analysis, which means that the points-to set of a pointer is independent of its calling context, and may point to wrong locations and this leads to spurious paths during backtracking, but the important point is that among these false values, or values of an other calling context, the correct value will also be present, so the correct path will be found too.

The bigger issue was caused by missing type information, which led to an incomplete call graph, and further the data flow between functions, where the call graph was incomplete, could not be restored. This missing information was mainly caused by calls to functions of the binary from external source, e.g. user interface events, and if the function signatures were not available for these functions, the information about the parameter types was also missing and this did prevent the backtracking of these parameters.

9.3 Closed Source Applications

These applications were downloaded from the App Store by searching for free applications containing the keywords *password*, *encrypted* or *secure* and a description that suggested that encryption was used. Not all of these applications could be processed by the analysis framework for the following reasons:

- Some applications were not available for ARMv8 CPUs, especially older applications
- Applications developed with the *Swift* language
- No calls to *CommonCrypto* functions

In total 179 applications were downloaded, but 88 of these applications could not be included due to the fact that no relevant function of the *CommonCrypto* library was called. This is a quite high number, considering that this was already a set of applications where the occurrence of such function calls had been assumed. This issue was either caused by handling this functionality with a third party library, or the applications did not use such function at all and sensitive data was probably stored or transferred in plaintext.

Additionally 25 applications were only available for the, by the analysis framework, unsupported CPU ARMv7 and 15 applications were developed in *Swift*. For the remaining 51 applications a successful analysis was possible where only two of those applications did not violate any of the rules defined in Chapter 6.

The results of this analysis are shown in Figure 9.2 by giving additional information than only the rule title by giving a short description about the error. As described there the most common rule violation was a non random initialization vector for encryption by constant data, which was in most cases the NULL initialization vector that consisted only of zero valued bytes.

The more surprising result was the high number of constant encryption keys present in those applications. These constant encryption keys were most times plain C-strings that had been directly used without applying any form of key derivation function. Additionally there were also constant passwords from which keys were derived using *CCKeyDerivationPBKDF()*. Most applications, which derived a key with *CCKeyDerivationPBKDF()*, used the previously described framework *RNCryptor* for handling this. *RNCryptor* generated per default a random initialization vector and salt value for encryption so these applications had random values, but there was still the issue of having a constant password.

The iteration count was also defined by *RNCryptor*, if it was used and in all of those applications this value was not modified from its default value of 10000. For the remaining applications the lowest iteration count was 1000 iterations, which is quite low, but did not violate the rule definition.

ECB mode was also rarely set and this was probably caused by the fact that the default mode is always CBC in *CommonCrypto*.

Non random encryption key	**Total violations: 32**
Constant string used as encryption key	22
Retrieved from keychain	3
Retrieved from NSUserDefaults	2
Single pass hash value	3
Constant key data	1
Constant password for PBKDF	10

(a)

Non random salt values for PBE	**Total violations: 1**
Hash value of constant string	1

(b)

Non random initialization vector	**Total violations: 34**
Constant data / NULL	31
Hash value of constant string	3

(c)

ECB mode for encryption	**Total violations: 2**

(d)

Figure 9.2: The results described above were found by the successful analysis of 51 closed source applications. In only two cases no rule violation was found using the reports generated by the analysis framework. Each of the 49 remaining applications had at least one rule violation. The subcategories of non random encryption keys add up to a larger number as the total number states since some applications violated this rule in different ways.

9.3.1 Common Mistakes

In this section some common mistakes, which were found in the closed source applications, are briefly described.

9.3.1.1 Using a String as Encryption Key

An often observed path had the same behavior as the following code snippet, which was copied from an project found on github[15]. The name of the method may differ, but the behavior of it was often the same as in this example:

```
- (NSData *)AES256EncryptWithKey:(NSString *)key
{
  char keyPtr[kCCKeySizeAES256 + 1];
  bzero( keyPtr, sizeof( keyPtr ) );

  [key getCString:keyPtr maxLength:sizeof( keyPtr ) encoding:
     NSUTF8StringEncoding];
```

[15]https://github.com/zenOSmosis/NSData-AESCrypt

```
    NSUInteger dataLength = [self length];

    size_t bufferSize = dataLength + kCCBlockSizeAES128;
    void *buffer = malloc( bufferSize );

    size_t numBytesEncrypted = 0;
    CCCryptorStatus cryptStatus = CCCrypt( kCCEncrypt, kCCAlgorithmAES128
        ,
                                           kCCOptionPKCS7Padding, keyPtr,
                                           kCCKeySizeAES256, NULL,
                                           [self bytes], dataLength,
                                           buffer, bufferSize,
                                           &numBytesEncrypted );
      ...
}
```

Listing 9.23: A code sample where a C-string serves as encryption key. This was a common pattern found in the evaluation of closed source applications.

The main problem was that a password, represented by the *key* parameter, was falsely used as encryption key without having any form of a secure key derivation function in place. Since passwords could have arbitrary length, this method either appended zero value bytes to the short password or truncated a password, if it exceeded the key size.

9.3.1.2 NULL or *nil* Initialization Vector

The use of a NULL initialization vector was the most common rule violation of the applications downloaded from the App Store. There were two different ways of passing a NULL initialization vector to a function of the *CommonCrypto* library. First, the pointer was directly set to NULL and this was the way it was done in most cases. The second way was to use a *nil* object and this way of passing a initialization vector was the more difficult to find during backtracking. The values of NULL and *nil* are both zero, but their semantic meaning is different in Objective-C. While NULL represents a pointer, *nil* describes an Objective-C object, which points to NULL. In LLVM IR they are hard to differentiate since this semantic difference holds only for Objective-C. However, *CommonCrypto* functions always expects pointers and not objects, which means, if the initialization vector is represented by an Objective-C object, it is required to get the pointer to the data buffer (usually by calling the *bytes* method).

If now a *nil* object was used and *bytes* was called the return value is NULL as it is defined in Objective-C. But *nil* has no type and therefore the call to *bytes* could not be handled by the analysis framework. Whenever these calls could not be recognized it was assumed that the passed object pointed to *nil* and following this a NULL initialization vector was used.

Usually, if this rule was violated, the initialization vector was set directly to NULL as shown in the code example in previous section, where the Objective-C wrapper method does not even have a parameter that could possibly be used for passing a non NULL initialization vector.

9.3.1.3 Keychain Stored Keys or Passwords

Some applications stored encryption keys or passwords in the systems keychain and they were retrieving it from there whenever needed. Since dumping the keychain is possible under certain circumstances (e.g. on a jailbroken device) this would leak the used encryption key.

9.3.1.4 Storing Sensitive Data in *NSUserDefaults*

This flaw was already found in one of the open source applications (see Section 9.2.5) and since values stored there are written to the filesystem in plaintext it is at no point suitable for storing encryption keys, passwords or any kind of sensitive data, but its very simple interface makes it attractive for being used in all kinds of scenarios.

Similar to the scenario of storing those values in the keychain, these values are easily retrievable from a jailbroken device and in this case *NSUserDefaults* stores the values directly inside the application container in plaintext.

9.3.2 Summary

Analyzing closed source applications led to the unexpected problems of having very few applications that were suitable to be processed by the framework since many application either do not use cryptographic functions or handled this via a third party library. Additionally there were repeatedly paths in the report that did not lead to a correct origin of a value, i.e. referenced values could not be backtracked completely. Nevertheless, the analysis framework was able to identify violations of the defined rules at some point in the applications.

The analysis reported errors in almost every application from the set of closed source applications downloaded from the App Store. These errors were mainly caused either by having a NULL initialization vector or by setting a hard coded string as password or directly as an encryption key. Especially the case of using hard coded "passwords" directly, without any form of key derivation function, was quite surprising, but since these methods have often the same name and signature it is likely that these methods were simply copied from a common source base.

Chapter 10

Conclusion

The goal of this thesis was to describe a method that is capable of detecting cryptographic misuse in applications for the iOS operating system. This is necessary since a variety of publicly available applications use cryptographic functions to ensure security of the processed data. However, the exact parameters defined for these functions are normally no public knowledge and therefore users usually have to trust developers for using such functions in a correct way.

In this thesis, we showed how iOS applications, which were compiled for an ARMv8 CPU, can be analyzed to detect cryptographic misuse, if functions of the *CommonCrypto* library were called. We implemented a framework that takes Mach-O binaries as input and performs a static analysis method to find the initial definitions of these parameters passed to cryptographic functions, where the checked parameters were defined by a set of rules described in Chapter 6.

Our implementation first decompiles the binary to LLVM IR code for decoupling the further analysis steps from being restricted to the ARMv8 instruction set and further to allow the inclusion and extension of already implemented tools for this language. The analysis itself uses static program slicing to identify the relevant statements that influence a specific variable, which is in our case a parameter passed to a cryptographic function. To get this set of relevant statements, information about data flow dependencies and control flow dependencies is required. The decompiled code of an application always contains pointer operations and this needs to be considered for computing the data dependencies since knowing the values a pointer references is essential for this analysis. We modified existing implementation of Andersen's [2] pointer analysis, which was initially designed for programs written in C, to work with the assembly-like code and its way of storing values in a register related data structure, produced by the decompiler.

Our method additionally extends program slicing to be able to extract a subset of the relevant instructions, which form an execution path consisting only of those instructions that modify the parameter of interest.

The evaluation of our implementation was done with two sets of applications. First, a set of open source applications was used to generate reports that could be verified by comparing the generated paths with those of the original source code by a manual inspection. Although in most cases the correct paths to the parameters origins were found, this part of the evaluation pointed out some weaknesses of our approach. While in some cases incorrect additional paths were found, other paths were missing in the analysis report due to an incomplete information about the types of objects. Without knowing the correct types our implementation is not able to continue and this leads to incomplete paths or missing paths.

After evaluating open source applications another set of applications was downloaded from the official App Store by searching for keywords where it could be assumed that cryptographic functions were called in the applications. For a major amount of these applications it was not possible to run our implementation and get results since no calls to functions of the *CommonCrypto* library were found, which means that either a third party library was used there, or these applications falsely advertised security of sensitive data. However, performing the analysis on the remaining applications yielded in finding violations of at least one rule in almost every application. Surprisingly, many of these violations were caused by having a constant encryption key, which makes an encryption pretty useless.

10.1 Future Work

The method presented in this work could be improved in various ways by either making the implementation runnable for more applications or by optimizing our approach for delivering fewer false results.

Decompilation The task of decompiling was done in a very simple fashion by reproducing the CPUs registers to store and load values instead of variables, which not only caused an overhead in code size, but also made the static analysis more difficult as the variables and parameters are not accessed using an identifier, but their memory location, which is retrieved from the register set.

Improving this by removing the register set from the decompiled code and define variables and parameters in the usual way, and not through a memory location stored in the register set, would reduce the code size and the static analysis would also benefit from this.

Context sensitivity The evaluation has shown that the context insensitive approach of the pointer analysis in our work causes the creation of wrong paths by the unavailable information about which function call defines a certain element of the points-to set. A context sensitive analysis would take the calling context into account and elements of the points-to sets could be matched to a specific function call and during backtracking only matching calls have to be considered.

Interface Builder definitions If the user interface was developed with the Interface Builder, the definitions about layout and elements of the user interface are not stored in the binary, but in files exclusively for this purpose. This includes the types of the interface elements and this leads to the issue of unknown types for those objects during analysis and further the values retrieved from those objects can not be backtracked.

Swift In our evaluation the number of applications, which were developed in the Swift language, was quite low, but this will probably change by apps released in the future. The main difference between Objective-C and Swift is that Swift does not use the dynamic way of sending messages for calling methods, but by dynamic dispatching with virtual method tables.

Bibliography

[1] Hiralal Agrawal, R.a. DeMillo, and E.H. Spafford. "Dynamic slicing in the presence of unconstrained pointers". In: *Proceedings of the symposium on Testing, analysis, and verification* (1991), pages 60–73. http://dl.acm.org/citation.cfm?id=120813 (cited on page 5).

[2] Lars Ole Andersen. "Program analysis and specialization for the C programming language". In: *Writing* May (1994), pages 111–. http://www-ti.informatik.uni-tuebingen.de/%7B~%7Dbehrend/PaperSeminar/Program%20Analysis%20and%20SpecializationPhD.pdf (cited on pages 6, 27, 29, 33, 34, 68, 91).

[3] ARM. *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf. May 2013 (cited on pages 15, 20, 25, 31, 32, 57).

[4] Marc Berndl et al. "Points-to analysis using BDDs". In: *ACM SIGPLAN Notices* 38.5 (2003), page 103. ISSN 03621340. doi:10.1145/780822.781144 (cited on page 6).

[5] David Binkley and Mark Harman. "A Survey of Empirical Results on Program Slicing". In: 2458.October (2004), pages 105–178. doi:10.1016/S0065-2458(03)62003-6. http://linkinghub.elsevier.com/retrieve/pii/S0065245803620036 (cited on page 6).

[6] Manuel Egele et al. "PiOS Detecting privacy leaks in iOS applications". In: *Proceedings of the 18th Annual Network & Distributed System Security Symposium, NDSS 2011* (2011), page 11 (cited on page 5).

[7] Manuel Egele et al. "An empirical study of cryptographic misuse in android applications". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (2013), pages 73–84. ISSN 15437221. doi:10.1145/2508859.2516693 (cited on pages 5, 49).

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-sensitive interprocedural points-to analysis in the presence of function pointers". In: *ACM SIGPLAN Notices* 29.6 (1994), pages 242–256. ISSN 03621340. doi:10.1145/773473.178264 (cited on page 27).

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), pages 319–349. ISSN 01640925. doi:10.1145/24039.24041 (cited on page 5).

[10] Ben Hardekopf and C. Lin. "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code". In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* 42.6 (2007), pages 290–299. doi:10.1145/1250734.1250767. http://portal.acm.org/citation.cfm?id=1250767 (cited on pages 6, 30, 60).

[11] Ben Hardekopf and Calvin Lin. "Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis". In: () (cited on page 6).

[12] Nevin Heintze et al. “Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second”. In: *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation* (2001), pages 254–263 (cited on page 6).

[13] Michael Hind and Anthony Pioli. “Evaluating the effectiveness of pointer alias analyses”. In: *Science of Computer Programming* 39.1 (2001), pages 31–55. ISSN 01676423. doi:10.1016/S0167-6423(00)00014-9. http://linkinghub.elsevier.com/retrieve/pii/S0167642300000149 (cited on page 28).

[14] Johannes Hoffmann et al. “Slicing droids: program slicing for smali code”. In: *Symposium on Applied Computing* (2013), pages 1844–1851. doi:10.1145/2480362.2480706. http://dl.acm.org/citation.cfm?id=2480706 (cited on page 5).

[15] Susan Horwitz, Thomas Reps, and David Binkley. *Interprocedural slicing using dependence graphs*. 2004. doi:10.1145/989393.989419 (cited on page 5).

[16] J. Jiang, X. Zhou, and D.J. Robson. “Program slicing for C-the problems in implementation”. In: *Proceedings. Conference on Software Maintenance 1991* September (1991), pages 182–190. doi:10.1109/ICSM.1991.160328. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=160328 (cited on page 6).

[17] James R Lyle and David Binkley. *Program Slicing in the Presence of Pointers*. 1993. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.7122%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf (cited on page 6).

[18] Karl J. Ottenstein and Linda M. Ottenstein. “The program dependence graph in a software development environment”. In: *ACM SIGPLAN Notices* 19.5 (1984), pages 177–184. ISSN 03621340. doi:10.1145/390011.808263 (cited on page 5).

[19] David J Pearce, Paul H J Kelly, and Chris Hankin. “Field-sensitive pointer analysis for C”. In: 11 (2004) (cited on page 6).

[20] Marc Shapiro and Susan Horwitz. “Fast and Accurate Flow-Insensitive Ponts-To Analysis”. In: *In Symposium on Principles of Programming Languages* (1997) (cited on page 6).

[21] Marc Shapiro and Susan Horwitz. “The effects of the precision of pointer analysis”. In: *Proceedings of the International Symposium on Static Analysis* (1997), pages 16–34. ISSN 0302-9743 (cited on page 6).

[22] B Steensgaard. “Points-to analysis in almost linear time”. In: *Popl* (1996), pages 32–41. http://dl.acm.org/citation.cfm?id=237727 (cited on page 6).

[23] Frank Tip. “A Survey of Program Slicing Techniques”. In: *Journal of Programming Languages* 5399.3 (1995), pages 1–65. ISSN 0963-9306. doi:10.1.1.43.3782 (cited on pages 6, 24).

[24] Meltem Sönmez Turan et al. “Recommendation for Password-Based Key Derivation Part 1 : Storage Applications”. In: December (2010) (cited on page 50).

[25] William E. Weihl. *Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables*. 1980. doi:10.1145/567446.567455. http://portal.acm.org/citation.cfm?doid=567446.567455 (cited on page 6).

[26] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pages 352–357. ISSN 0098-5589. doi:10.1109/TSE.1984.5010248 (cited on pages 5, 6, 23, 24).

[27] Robert P. Wilson and Monica S. Lam. “Efficient context-sensitive pointer analysis for C programs”. In: *ACM SIGPLAN Notices* 30.6 (1995), pages 1–12. ISSN 03621340. doi:10.1145/223428.207111 (cited on page 27).