Automated Binary Analysis on iOS – A Case Study on Cryptographic Misuse in iOS Applications

Johannes Feichtner
Graz University of Technology

David Missmann
Graz University of Technology

Raphael Spreitzer
Graz University of Technology

ABSTRACT

A wide range of mobile applications for Apple's iOS platform process sensitive data and, therefore, rely on protective mechanisms natively provided by the operating system. A wrong application of cryptography or security-critical APIs, however, exposes secrets to unrelated parties and undermines the overall security.

We introduce an approach for uncovering cryptographic misuse in iOS applications. We present a way to decompile 64-bit ARM binaries to their LLVM intermediate representation (IR). Based on the reverse-engineered code, static program slicing is applied to determine the data flow in relevant code segments. For this analysis to be most accurate, we propose an adapted version of Andersen's pointer analysis, capable of handling decompiled LLVM IR code with type information recovered from the binary. To finally highlight the improper usage of cryptographic APIs, a set of predefined security rules is checked against the extracted execution paths. As a result, we are not only able to confirm the existence of problematic statements in iOS applications but can also pinpoint their origin.

To evaluate the applicability of our solution and to disclose possible weaknesses, we conducted a manual and automated inspection on a set of iOS applications that include cryptographic functionality. We found that 343 out of 417 applications (82%) are subject to at least one security misconception. Among the most common flaws are the usage of non-random initialization vectors and constant encryption keys as input to cryptographic primitives.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**;

KEYWORDS

iOS, Reverse Engineering, Program Analysis, Cryptographic Misuse

ACM Reference Format:
Johannes Feichtner, David Missmann, and Raphael Spreitzer. 2018. Automated Binary Analysis on iOS – A Case Study on Cryptographic Misuse in iOS Applications. In *WiSec '18: Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, June 18–20, 2018, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3212480.3212487

1 INTRODUCTION

Smartphones are ubiquitous in our daily lives and facilitate mobile working. Extensive technological capabilities of these devices enable consumers to carry out tasks that would have required a personal computer some years ago. Since the Apple iOS operating system is among the most popular mobile platforms, many software developers are also deploying programs that process sensitive user data. A common method to protect this information is the use of security APIs and cryptographic functionality, provided by the platform. For this to be effective, essential rules need to be obeyed, as otherwise the attainable level of security would be weakened or, in the worst case, entirely defeated. Among these essential rules are, for example, (1) the need to prevent the electronic code book (ECB) mode for block ciphers, (2) the need to prevent static keys or credentials, which are in the worst-case hard-coded into the application, or (3) the need to prevent static seeds for generating random numbers. Irrespective of whether an erroneous implementation is a result of a developer's ignorance, lack of knowledge, or a too complex documentation of the crypto API, the identification of crypto API misuse is of utmost importance to protect sensitive data and to provide the intended functionality, e.g., in case of password managers. Since developers usually do not provide detailed information and the source code of mobile applications is not made available, the correct implementation of cryptographic functionality can only be verified by reverse engineering the final application.

Analysis frameworks for the *Android* platform are already available and studies [12, 16] have confirmed that cryptographic misuse represents a significant problem on Android. Although one would expect that cryptographic misuse is also present on iOS platforms, an analysis on iOS has not been performed so far. Hence, the motivating questions in this paper are: *(1) What is needed to automatically analyze iOS apps for possible crypto misuse*, and *(2) do iOS apps actually violate common cryptographic principles?* In contrast to the Android platform [12, 20], such an analysis represents a challenging task on the iOS platform. For instance, Android applications are provided in a reversible bytecode format, whereas iOS applications are compiled to machine code that is tailored to a particular CPU architecture. Manually inspecting the disassembled code of an iOS binary can be a challenging endeavor. The increasing complexity and size of today's applications impede a conclusive analysis of security-critical programs. Automated binary analysis tools, in contrast, are typically not aligned to the characteristics of the iOS platform and, thus, fail to perform a thorough data flow analysis. Among these characteristics are, for example, dynamic control-flow decisions and the use of a pointer-aware language, where pointers may point to different memory locations and memory locations may be referenced from different pointer variables (*aliasing*). Especially the use of a pointer-aware language requires particular attention in order to focus on code parts that are essential for a particular computation by means of program slicing [55].

In this paper, we introduce a solution that enables such an analysis on iOS applications by addressing the following challenges.[1]

(1) *Decompiling and simplifying machine code*: The analysis of 64-bit binaries compiled for the ARMv8 architecture is error-prone and tedious. Therefore, we transform the binary into higher-level LLVM intermediate representation (IR) [37], where all low-level CPU instructions have to be modeled appropriately. This allows to re-use existing LLVM-based tools, such as KLEE [10], PAGAI [31], and LLBMC [41].
(2) *Language peculiarities*: iOS applications are developed in runtime-oriented languages such as Objective-C and Swift, and the majority of control-flow decisions are made during runtime. Instead of calling methods of objects directly or through virtual method tables (*vtables*), this task is delegated to a dynamic dispatch function in the Objective-C runtime library. To recover a semantically correct control flow from the binary, we reconstruct the hierarchies of classes, methods, and types from binaries. This information allows to resolve the target of a function call through the dispatch routine.
(3) *Pointer analysis:* Computing control flow and data dependencies, as well as the identification of instructions and variables that have an impact on a particular program statement, requires information about where different variables (and CPU registers) point to during execution. Since computing points-to sets is an undecidable problem [3], we propose a solution addressing the trade-off between the accuracy of program slices and the runtime overhead.

By solving these challenges, we developed a framework that allows inspecting 64-bit iOS binaries by reconstructing the corresponding control flows and data flows. Based on the *Common-Crypto* library for iOS, we formulate and specify rules for the use of cryptographic APIs, and we automatically test several hundred closed-source apps regarding their compliance to these rules. Our results indicate that the majority of apps use cryptographic APIs in unintended ways and, thus, violate essential rules.

Contributions. The contributions are as follows:

(1) We introduce and develop the necessary building blocks for a framework to analyze 64-bit iOS application binaries.
(2) We formulate the low-level properties required to identify cryptographic misuse of security APIs on iOS.
(3) Based on several open-source applications, we iteratively refine the framework to reduce the number of false positives.
(4) We apply our framework on more than 400 iOS applications that rely on cryptographic APIs and identify security-critical misconceptions in the majority of these applications.

Outline. In Section 2, we discusses related work. In Section 3, we discuss the structure of iOS applications, program slicing, and pointer analysis. Section 4 introduces our analysis workflow. In Section 5, we explain binary decompilation. In Section 6, we describe how to resolve pointer states and how to obtain an accurate call graph. Section 7 elaborates on how we leverage this information for program slicing and parameter backtracking. In Section 8, we present security rules to detect common misconceptions, followed by an evaluation in Section 9. Finally, we conclude in Section 10.

[1]The framework is available at: https://github.com/IAIK/ios-analysis

2 RELATED WORK

The analysis of security aspects on mobile platforms has attracted a lot of attention in the past years. A majority of publications in this field focus on the Android ecosystem where the openness of the platform promotes program inspection. Supported by the fact that Dalvik bytecode in Android applications can be decompiled to Java code, existing tools for static analysis are easily applicable [5, 6, 54]. Likewise, approaches for dynamic analysis [19, 50] enable live information-flow tracking while an application is executed by the Dalvik virtual machine. Tailored to their individual use-case, most analysis-related works aim to disclose possible leaks of private data [11, 13, 25, 57], identify malware [23, 24, 26, 44], or uncover security deficiencies in applications [8, 14, 21, 22].

Among all related research, the work by Egele et al. [16] comes closest to this paper. They evaluated Android applications regarding their use of cryptographic APIs in unintended ways. Based upon the Android reverse-engineering framework Androguard [4], a control flow graph over all functions is derived. Subsequently, static program slicing [55] is employed to inspect the parameters passed to cryptographic operations. We elaborate a similar concept for iOS. As there are no decompilers available for iOS binaries yet, we introduce a generic decompiler transforming ARMv8 binaries to LLVM IR code. With applications developed in Objective-C or Swift, it is significantly harder to obtain a meaningful program control flow on iOS compared to Android. In addition, the need for pointer analysis and the reconstruction of information from the binary are major differences that required new approaches to be pursued.

Although static binary analysis is a well-established practice, only few contributions target the iOS platform. In [17], the authors approach this field by studying privacy threats in iOS applications using the disassembly of binaries. They create a control flow graph and perform a reachability analysis to identify possible privacy leaks. Other works [15, 36, 58] survey the usage of private APIs or pursue a source-to-sink analysis using static and dynamic methods.

We use static slicing to extract a subset of the program affected by a specific variable. Introduced by Weiser [55], the idea was to describe dependencies between statements using data flow equations. Subsequent works extended the concept to Program and System Dependency Graphs (PDG, SDG) [33, 42], defined as a reachability problem. Agrawal et al. [1] presented a solution to create program slices using PDGs that handle pointers and arrays in intra-procedural programs. In [9], the supportive impact of pointer analysis on program slicing has been underlined.

For a reliable analysis of data flows, it is inevitable to determine what values are referenced and modified by pointers. Shapiro and Horwitz [48] compared the precision of different pointer analysis methods [3, 47, 49]. Of the three approaches, Andersen's [3] was the most precise but had a runtime of $O(n^3)$. Steensgard's algorithm [49] runs in almost linear time having less precise results. The third algorithm [47] is a compromise between runtime and precision. The main difference between Andersen's and Steensgard's algorithms is that Andersen uses so-called *inclusion relations*, while Steensgard builds on *equality relations*.

Since iOS applications usually consist of a very large code base, Andersen's initial algorithm would lead to a poor overall performance. However, various improvements [7, 28–30, 43] have been proposed to tackle this issue. Hardekopf and Lin [29] improved the approach to an almost linear runtime while still providing results similar to Andersen's algorithm. In our work, we use the constraint optimizations proposed in [28]. By merging the definitions of similar pointer variables, the input size for constraint solving becomes smaller. In combination with [29], strongly connected components in a graph are detected and get collapsed to a single node since they form a cycle in which each node still points to the same locations.

3 BACKGROUND

In this section, we present background information about the structure of iOS applications, program slicing, and pointer analysis.

3.1 iOS Applications

iOS executables use the Mach-O file format, which allows a single binary file to contain multiple Mach-O files for different CPU architectures. Since 2013, new iOS devices are equipped with a 64-bit ARMv8 CPU and it is required that applications are provided for this architecture, which is why we focus on this particular architecture. A Mach-O file is split into three regions:

Header: Identifies the file as Mach-O file and includes information about the target CPU architecture.

Load Commands: Specifies the file layout and designated memory location of segments.

Data: Consists of different regions and sections that are loaded into memory. The *Dynamic Loader Info* defines where dynamic symbols have to be stored during execution.

We evaluate the *Load Commands* section to refer to the data located in the subsequent region. The *Data* region itself contains all information needed for execution, including the machine instructions. However, the instructions alone are not sufficient to accurately decompile the executable binary of an iOS application.

Method invocations are handled by a dynamic dispatch function in the Objective-C runtime library, which requires the type of the object and the name of the method to be called. Therefore, the following sections in the binary play an essential role:

__objc_classlist: List of pointers to descriptions of classes in a binary. The pointers point to addresses of the section *__objc_data*.

__objc_data: Contains a pointer to the superclass and a pointer to *__objc_const* for retrieving class infos.

__objc_const: Has details for every class, implemented methods (name and memory location) and protocols, instance variables and defined properties in the binary.

Dynamic Loader Info: Includes pointers to classes that are not present in the binary, e.g., references to the Objective-C runtime library. By traversing this structure, we manage to reconstruct a complete class hierarchy.

During execution, applications can invoke externally defined library functions using *indirect symbols*. Occurring as *lazy* or *non-lazy* symbols, our analysis has to detect and handle these calls correctly as they may create or modify data. Non-lazy symbols are resolved when the binary is loaded using the binding information stored within the *Dynamic Loader Info* section. In contrast, lazy symbols are followed when they are first accessed.

The Mach-O file includes method signature definitions for all contained Objective-C and Swift methods. The type definitions for arbitrary methods are stored in a compressed way, prefixing all objects with the universal identifier @. Method signatures for protocol methods are stored with their complete signature. This information supports a more precise call graph generation since parameters, passed to protocol methods, are probably not allocated in the code of the binary but included from other libraries. E.g. UI protocol methods are called from within the *UIKit framework*. Without these definitions there would be no way to find out the types of externally allocated parameters and the call graph generation would lack calls made to these objects.

3.2 Program Slicing

Static slicing can be used to determine all code statements of a program that may affect a value at a specified point of execution (*slicing criterion*). The resulting *program slices* cover all possible execution paths and allow conclusions to be drawn about the functionality of the program. We adopt the algorithm of Weiser [55] to create slices of LLVM IR code and to find paths from the origin of a parameter to its use, e.g., in cryptographic functions.

Weiser presented an *intra-procedural* method that models the data flow within a function using equations. Relevant variables and statements are determined in an iterative manner. As summarized by Tip [52], the algorithm consists of two steps:

1. **Follow data dependencies:** This step is executed iteratively, if control dependencies are found.
2. **Follow control dependencies:** Includes relevant variables of control flow statements. Step 1 is repeated for affected variables.

To create slices over multiple functions, the approach can be extended to *inter-procedural* slicing in two steps: first an intra-procedural slice of a function P is computed, followed by the generation of new slicing criteria for every function that calls P or is called by P. In Weiser's concept, the generation of new criteria is described as $DOWN(C)$ for the callers of P and as $UP(C)$ for functions that are called by P based on the slicing criterion C. Due to the way how parameters are passed with ARMv8, this approach is not immediately applicable to our use case (cf. Section 7).

3.3 Pointer Analysis

When identifying instructions and variables with an impact on program statements, it is essential to also know where they might point to during execution. Pointer analysis can be used to support the slicing process with accurate information about pointer states.

Introduced by Andersen [3], pointer analysis is described as a *set-constraint* problem in which a constraint system C is created for a given program. By solving the system, it is possible to determine the locations a variable might point to during program execution. All constraints are of the type $a \supseteq b$, which means that information flows uni-directional from b to a. There are four different types of constraints that may be added to C (see Table 1). In our work, constraint generation is done according to these types by identifying patterns in the decompiled LLVM IR code.

Andersen defined a *context-sensitive* and a *context-insensitive* version of his algorithm. While the former leads to more precise results by separating information originating from different paths, its

Table 1: Constraints and their meaning [29]

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

time complexity is exponential [18, 56]. Therefore, context-sensitive pointer analysis does not scale well for larger programs with a large number of calling contexts. In the scope of our work, we prefer the context-insensitive method for the computation of points-to sets.

Flow-sensitive pointer analysis takes the control flow of a program into account and computes an individual points-to set for each instruction. The *flow-insensitive* pendant disregards branches and collects all information in a single points-to set. Hind and Pioli [32] have shown that a flow-sensitive pointer analysis does not improve the precision of the results if a context-insensitive algorithm is used. In our solution we, thus, opted for the insensitive approach.

3.4 Cryptography on iOS

Applications rely on cryptography to protect sensitive data. On iOS, the platform libraries *CommonCrypto* and *Security* expose APIs to perform security-related operations. The first library provides symmetric ciphers, hash functions, and a key derivation function (PBKDF2). The second library provides functionality for asymmetric ciphers, certificate handling, and keychain access.

All rules described later in this work have their origin (the point where the slicing criterion is defined) in functionality of *CommonCrypto*. The following three functions and their parameters play an essential role in the definition of slicing criteria:

CCCryptorCreate: Comprises a family of similar functions that initialize a *CCCryptorRef* reference. This object is then used to perform encryption or decryption. *CCCryptorCreate* takes the following parameters, relevant for our analysis:
- *op:* Defines if data is to be encrypted or decrypted.
- *options:* Specifies the cipher mode (ECB or CBC).
- *key:* A pointer to the key material.
- *iv:* A pointer to the initialization vector (IV).

CCCrypt: Similar to *CCCryptorCreate* functions, *CCCrypt* is a self-contained alternative that expects all data and cipher-related options to be passed immediately. The parameters of interest for us are the same as for the *CCCryptorCreate* function family.

CCKeyDerivationPBKDF: Derives a key using PBKDF2. The subsequent parameters are crucial for the security:
- *password:* The password as a C string.
- *salt:* A pointer to the salt value.
- *rounds:* The number of iterations.

4 SYSTEM DESIGN

In this section, we introduce our analysis concept and outline the pursued steps. As depicted in Figure 1, the analysis starts with a raw iOS binary. The outcome is a report that describes, for every checked security rule, whether any violations have been detected. If this is the case, the affected execution path(s) and the value of the insecure parameter(s) are appended. The overall workflow can be summarized as follows:

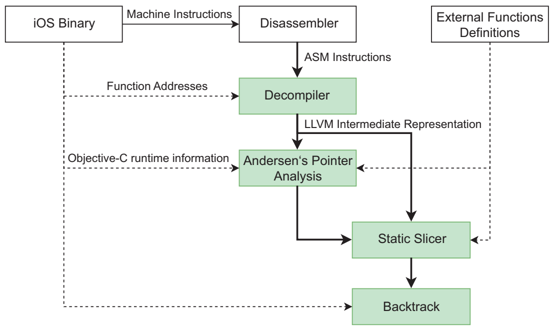
iOS Binary
Machine Instructions
Disassembler
External Functions Definitions
ASM Instructions
Function Addresses
Decompiler
LLVM Intermediate Representation
Objective-C runtime information
Andersen's Pointer Analysis
Static Slicer
Backtrack

Figure 1: Analysis workflow of an iOS binary

1. **Disassemble:** After extracting the 64-bit ARMv8 binary from the Mach-O file, we leverage the disassembler of the LLVM framework to generate assembly code.
2. **Decompile:** We translate the ARMv8 assembly code into LLVM IR code by extending an existing decompiler framework with instruction semantics for ARMv8.
3. **Pointer Analysis:** The points-to sets are computed for all pointers using an enhanced context-insensitive approach that scales well for iOS applications of any size.
4. **Static Slicing:** Relevant segments are identified in LLVM IR code based on the slicing criteria derived from the predefined security rules.
5. **Parameter Backtracking:** All execution paths a parameter can take are backtracked to the slicing criterion. Thereby, we verify whether encountered statements meet our security rules.

Except for the initial disassembly step, we contribute new approaches and augment existing frameworks. In the subsequent sections of this paper, our solutions are explained in detail.

5 DECOMPILATION TO LLVM IR

In this step, we translate 64-bit ARMv8 binaries into LLVM IR code. The aim is to obtain a simpler representation that still models a semantically correct control flow and data flow of the application.

The LLVM compiler *frontend* takes source code as input. After the code has been fully tokenized, parsed, and analyzed, LLVM IR code is emitted. The compiler *backend* is then responsible to optimize it, assemble machine code and link the resulting object. Within this process, LLVM tailors the output to a specific CPU architecture using the corresponding register and instruction descriptions. While the former specifies the processor's register types and relations, the latter includes pattern definitions used to select machine instructions in place of IR instructions during code generation [39].

In order to decompile binaries, we apply the patterns in the opposite direction. Therefore, we build on the reverse-engineering framework *Dagger* [2]. In contrast to similar approaches [51, 53], it extends LLVM and relies on *instruction semantics* [38] to translate machine instructions to LLVM IR code based on target descriptions of registers and instructions. The semantics describe the different types of operands a machine instruction can take and what operations have to be applied to get an equivalent in LLVM IR code.

For most machine instructions, semantics can be generated from the descriptions. However, since not all instructions necessarily have a counterpart in LLVM IR, we manually supplemented missing definitions for the 64-bit ARM architecture. The decompilation process is described in Appendix A in more detail.

The ARMv8 instruction set knows various control flow statements that all have to define a target where the branch should end up. While for most instructions the address is statically defined, the unconditional branch statements BR, BLR, and RET read the destination from a register whose value cannot be determined during decompilation. As a remedy, we perform a pointer analysis on the register and then update the branch targets in the control flow.

Function calls (BL and BLR instruction in ARMv8) are translated to the `call` instruction in LLVM IR. Passing parameters to functions is done by storing values in registers and/or on the stack [40]. Further steps are not required since the called function retrieves the values from the registers and the stack via the register set. This, however, means that we do not have information about parameters or return values. As static analysis requires this information to reconstruct the data flow between functions, we have to restore the missing type information from the binary (see Section 7.1).

5.1 Recovering Lost Information

During compilation, the LLVM backend strips function prototypes, local variables, and other type information from the LLVM IR code. Without this knowledge, it is non-trivial to generate a valid call graph for use with static analysis. In this section, we describe how the needed information can be reconstructed from the binary.

5.1.1 Intraprocedural Control Flow. Grouping statements into basic blocks helps to understand the control flow in a function. Branch instructions always indicate the exit point of one basic block and point to succeeding statements. Since they are referenced by their instruction offset in the binary, we can leverage this address to unambiguously find the entry points of the subsequent basic blocks. The immediate predecessors of entry points can then be set as exit instructions of basic blocks. As a result, we obtain an accurate control flow graph from the basic blocks. Without the binary addresses, successors would not always be clearly visible.

5.1.2 Function Parameters and Return Values. Knowing about these definitions is essential for the data flow analysis. As function prototypes are completely removed during compilation, we can only make assumptions based on the calling convention [40]. A function parameter is assumed if a value is read from a memory location where a parameter may have been stored previously. To identify these locations, we traverse the control flow graph from the entry point to the `load` instruction. A parameter is also assumed if no instruction is found that stores a value to a location.

To identify return values, the control flow graph is not traversed from the function entry point but from the `call` instruction to which will be returned after executing the function.

5.1.3 External Symbols. iOS applications include placeholders for symbols that refer to external libraries. At runtime, the placeholders are replaced with the correct associations from the loaded libraries. For the purpose of static analysis, we can imitate this behavior and store a reference to a symbol's name and its library association within the decompiled code.

5.2 Implementation

As the *Dagger* framework is only capable of translating x86 binaries to LLVM IR, we have extended it with the definition of instruction semantics for 64-bit ARMv8. By also considering specifics of iOS binaries, we succeeded in modeling the correct control flow and can complement the subsequent data flow analysis with accurate type information. In the following, we highlight some changes to Dagger that were made to optimize further analysis.

5.2.1 Registers. Dagger uses the register description of the CPU to create a data structure which simulates internal storage. However, in practice, registers of an ARMv8 CPU overlap with the LLVM definitions and storing multiple sub-registers in a single super-register will lead to difficulties when computing data dependencies. Thus, we modified Dagger's model, such that a single super-register will now only store the value of exactly one physical register. Originally, the largest super-registers contained the values of four physical registers and each of them is stored in exactly four different super-registers. This means that three values of each of the largest super-registers can be removed and 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 Dagger's approach to this solution still keeps the data stored in registers consistent while allowing an easier computation of data dependencies of a super-register.

5.2.2 Non-Volatile Registers. The content of these registers must be preserved across function calls (*callee-saved registers*). Although the callee accesses these values only for saving and restoring them, this operation causes data dependencies between the caller and callee. Since the values are not used for anything else, we can optimize the code by cutting these data dependencies.

5.2.3 Tail Calls. In ARMv8, a *branch-and-link* (BL) instruction is replaced with `branch` (B) if `call` is the last instruction of a function before returning to the caller. Since `branch` cannot leave the scope of a function in LLVM IR, Dagger replicates the target's code to the current function. This leads to a wrong call graph as code inlining prevents that an edge is added for the call. We resolve this issue by checking whether the target address of a branch instruction is outside the function body. In that case, we replace the tail call with a regular `call` and return statement.

6 POINTER ANALYSIS

Pointer analysis is required to support the subsequent slicing step with information about pointers and to compute an accurate call graph that is necessary for data flow analysis. We rely on context- and flow-insensitive constraint generation with a focus on LLVM IR and the characteristics of Objective-C (and implicitly Swift).

6.1 Iterative Constraint Generation

The algorithm presented by Andersen [3] generates constraints once for each instruction, which only works if all values are already provided within an instruction. However, in case of function pointers and calls to Objective-C methods, points-to knowledge is required before being able to process these calls. Therefore, we adapted the constraint generation process to handle call instructions with pointers in an iterative manner (cf. Appendix B.1).

Focusing on the C language, Andersen's analysis does not cover an important feature that occurs in decompiled code: All memory accesses are done by converting an integer to a pointer using the `inttoptr` instruction. Without considering these instructions, constraint generation would produce a points-to set with all pointers referencing the abstract location *Unknown*. We modify the rules for constraint generation to cover the `inttoptr` instruction. Based on the constraints by Hardekopf and Lin [29] (see Table 1), we identify patterns in the decompiled code for all sources of `inttoptr`.

Binary offsets are accessed by `inttoptr` via static addresses. This is the simplest memory access since the operand of `inttoptr` includes the source address (constant integer value).

Heap addresses are created via memory allocation functions (e.g. `malloc`). Since the return values describe an abstract location, it can immediately be used for pointer analysis.

Stack memory is usually accessed by adding a static offset to the stack pointer (or frame pointer). However, different instructions that access the same location on the stack are not easily identifiable since a separate pointer is created for each of them. Thus, we have to match stack locations with all variables which might contain the same value during execution. This is done by determining all operations that use the stack pointer in an `add` or `sub` operation with a constant integer. As these operations are mainly responsible to create the stack address for a subsequent use with `inttoptr`, we can match different pointers to the same stack location.

6.2 Objective-C Peculiarities

As iOS applications are compiled from source code in the runtime-oriented languages Objective-C and Swift, the LLVM frontend rewrites direct method invocations to use a dynamic dispatcher function instead. `objc_msgSend()` is a function in the Objective-C runtime library, responsible to decide at runtime what method to call. Therefore, it takes two parameters that specify the class or object, and the name of the method to call. Both arguments ($X0$ and $X1$ in ARMv8) hold pointers to locations in the binary where the actual values are retrieved from. Consequently, this also affects the call graph generation which naturally fails to add edges if neither type information nor selector names are defined in the LLVM IR code. We, thus have to restore the correct call graph by adding edges for the methods referenced in calls to `objc_msgSend`. The details can be found in Appendix B.2.

Other language-specific peculiarities that need special processing include the Objective-C features *Blocks* and *Fast Enumeration*. For both cases, constraint generation has to be adapted to identify the instructions that access locations on the stack.

7 STATIC SLICING

The purpose of static slicing is to compute all code segments that affect a slicing criterion. We adopt the algorithm of Weiser [55] to work with LLVM IR code by considering characteristics of ARMv8.

As parameters can be passed using both the stack or registers on ARMv8, Weiser's approach does not immediately work for decompiled LLVM IR code: Each function has exactly one formal parameter that represents the register set. Since this set is the same for all functions, no parameters could be substituted to generate a new set of slicing criteria when a function is called. Our solution for this issue is to extend the collection of all relevant variables $ROUT(i)$ with information about `load` and `store` operations that read or modify registers before a function call.

Let $S(v,r)$ be a `store` instruction that stores a variable v to the register r and $L(v,r)$ the corresponding `load` instruction. We now define $STORE(i)$ to return the set of last variables that were stored to a register before the instruction i and $LOAD(i)$ to return the set of first variables, loaded from a register after the instruction i:

$$\begin{aligned} STORE(i) = \Big\{(v,r) \mid \exists\, S_i(v,r) \rightarrow {}^{*}_{CFG}\, i, \\ \nexists\, L_i(v,r) \rightarrow {}^{*}_{CFG}\, i\, S_j(v',r) \rightarrow {}^{*}_{CFG}\, i\Big\} \end{aligned} \tag{1}$$

$$\begin{aligned} LOAD(i) = \Big\{(v,r) \mid \exists\, i \rightarrow {}^{*}_{CFG}\, i\, L_i(v,r), \\ \nexists\, i \rightarrow {}^{*}_{CFG}\, L_j(v',r')\ {}^{*}_{CFG}\, L_i(v,r)\Big\} \end{aligned} \tag{2}$$

Using these sets, parameter substitution from formal to actual parameters is possible even if all functions use the same register set. If the stack is used for parameter passing between functions, interprocedural slicing is achieved using the points-to analysis. If a stack parameter is a relevant variable, the corresponding memory location will be the same in the caller and callee functions.

7.1 Restoring Missing Type Information

A conclusive data flow analysis requires knowledge about all object types. After recovering function parameters and return values in the decompilation step, type information is still missing for instance variables and protocol methods. If available in the code of the binary, the type definitions can easily be resolved (see Section 3.1). However, if instance variables are allocated by a function of an external library, finding their type information can be a challenging task.

Instance variables are always accessed by loading an offset from an individual static address in the binary. This enables us to find all instructions that refer to a particular variable and a single abstract location [47] can be specified. Similarly, the binary has no precise type information for parameters of methods that are declared by a protocol in Objective-C or Swift. Although creating abstract locations for each parameter does not allow for a precise pointer analysis, it is still possible to identify calls made using these objects.

7.2 Parameter Backtracking

Program slices summarize all relevant statements and variables that influence a certain parameter. This information can now be split into single execution paths. For the verification whether a parameter complies with our security rules, we use backtracking to evaluate all paths a variable can take until it is first defined. In case a rule violation is detected, we are able to pinpoint the affected information flow and can highlight problematic statements.

In the following, we explain our solution to find predecessors, isolate execution paths, and how to avoid cycles while backtracking.

7.2.1 Finding Predecessors. The static single assignment form (SSA) of LLVM IR already provides a simple way of backtracking instructions that do not use pointers for memory access. However, if a value is read from a location referenced by a pointer, it is not possible to determine the preceding store instruction by inspecting only the statement. For program slicing, it is necessary to add the location to the relevant set and traverse the control flow graph 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, we add the statement s, which induced this location, to a separate set $R_{Sources}(i,l)$. It contains all statements that added the relevant variable l at statement i and 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 ($i \rightarrow_{CFG} j$):

$$\begin{aligned} R_{Sources(i,l)} &= \{i \mid i \in S_C,\ l \in REF(i)\} \\ &\cup \{i \mid i \in R_{Sources}(j,l),\ i \notin S_C\} \end{aligned} \tag{3}$$

With this information, it is now possible to determine the preceding 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:

$$Pred'(r) = \{s \mid r \in R_{Sources}(s,l),\ l \in DEF(s)\} \tag{4}$$

The set of predecessors that includes all instructions, which modify a referenced variable can then be described as:

$$\begin{aligned} Pred(i) = \{s \mid r \in R_{Sources}(s,l),\ l \in DEF(s),\ l \in Loc\} \\ \cup \{op \mid op \in Operators(i),\ op \in Instructions\} \end{aligned} \tag{5}$$

7.2.2 Extracting Execution Paths. Using the $Pred(i)$ set of an instruction i we get all predecessors of a statement. All possible paths of a value v to its origin can be seen 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:

$$\begin{aligned} E^{(0)} &= \{(v,j) \mid j \in Pred(v)\} \\ E^{(k)} &= \{(i,\ j) \mid (j,\ k) \in E^{(k-1)},\ i \in Pred(j)\} \end{aligned} \tag{6}$$

7.2.3 Avoiding Cycles. Whenever parts of the code are executed inside a loop, cyclic dependencies may occur in the execution path. To avoid infinite loops during analysis, we discard the branch of a path immediately if an instruction is found that is already contained in the path. This means that branches without cyclic dependencies are still tracked and will lead to the initial definition of a value.

7.3 Implementation

We implemented program slicing and parameter backtracking for LLVM IR code. The slicing step is required to integrate with points-to information, and the subsequent backtracking step has to support the call graph restored during pointer analysis.

We build on *LLVMSlicer* [34], an implementation of Weiser's algorithm for static slicing, which also includes Andersen's algorithm for pointer analysis. While this works fine for smaller programs, it leads to a poor performance for iOS applications that usually consist of a very large code base. We tackle this issue by replacing the implementation with constraint optimization techniques [27] proposed by Hardekopf and Lin [28]. Taking up this idea, we look for pointers that have an equivalent points-to set and merge their representations. Constraint solving is optimized by detecting and collapsing strongly connected components in the constraint graph [29].

We further augment the slicer with parameter backtracking to extract single execution paths. Subsequently, we can verify whether the paths meet or violate predefined security-critical conditions.

8 DETECTING CRYPTOGRAPHIC MISUSE

In this section, we present a set of security rules tailored to detect common security-critical implementation flaws in iOS applications.

8.1 Rule Checking

Using parameter backtracking, we obtain all execution paths that affect a certain slicing criterion. To verify whether or not a path violates a specific rule, each element is tested against predefined conditions. In case a problematic value is identified, the affected subpath and slicing criterion are written to a report.

The rules are designed using JSON definitions and allow for easy extensibility. It is also possible to group similar functions, such as those belonging to the `CCCryptorCreate` family.

8.2 Evaluating Security Properties

In this work, we focus on cryptographic functionality provided by the *CommonCrypto* library, *i.e.*, the default crypto library on iOS. We consider custom crypto implementations out of scope, as crypto should not be implemented by developers themselves anyways.

Egele et al. [16] described common rules in cryptography that have to be considered by Android developers in order to avoid security issues, and they also summarize possible security implications. We define similar rules based on the anatomy of *CommonCrypto* methods to detect cryptographic misuse in iOS applications.

Rule 1: Do not use ECB mode for encryption. In ECB mode, data blocks are enciphered independently from each other and cause identical message blocks to be transformed into identical ciphertext blocks. Consequently, data patterns are not hidden and confidentiality may be compromised. By default, iOS prefers CBC mode. ECB mode is only used if the developer explicitly specifies to use this mode of operation. Thus, for this rule to be satisfied, `CCCrypt()` or `CCCryptorCreate()` must not be invoked with the flag `kCCOptionECBMode` being set as option.

Rule 2: Do not use a non-random IV for CBC encryption. Constant or predictable IVs lead to a deterministic and stateless encryption scheme that is susceptible to chosen-plaintext attacks. If CBC mode is selected for encryption, the developer has to provide a cryptographically secure IV. If no IV is specified at all, the cipher uses an all-zeros IV, which is at least as bad as using a constant IV.

We consider the rule to be fulfilled, if an IV used with `CCCrypto()`, `CCCryptorCreate()`, or `CCCryptorCreateWithMode()` is generated by a cryptographically secure random number generator. This can either be `CCRandomGenerateBytes()` in *CommonCrypto* or `SecRandomCopyBytes()` in the *Security* library.

Rule 3: Do not use constant encryption keys. Keeping encryption keys secret is a vital requirement to prevent unrelated parties from accessing confidential data. Statically defined keys clearly violate this basic rule and render encryption useless. A key used for symmetric encryption with `CCCrypto()`, `CCCryptorCreate()`, or `CCCryptorCreateWithMode()` has to originate from a non-constant source. Analogous to keys, this rule can also be applied to passwords that are passed to `CCKeyDerivationPBKDF()`.

Rule 4: Do not use constant salts for PBE. A randomly chosen salt ensures that a password-based key is unique and slows down brute-force and dictionary attacks. To fulfill this rule, a call to the key derivation function `CCKeyDerivationPBKDF()` must not be provided with a salt that depends exclusively on constant values.

Rule 5: Do not use fewer than 1,000 iterations for PBE. A low iteration count significantly reduces the costs and computational complexity of table-based attacks on password-derived keys. To satisfy this rule, we expect applications to specify more than 1,000 rounds as an argument for `CCKeyDerivationPBKDF()`.

Rule 6: Do not use static seeds to seed `SecureRandom`. If a PRNG is seeded with a statically defined value, it will produce a deterministic output which is not suited for security-critical applications. As the platform-provided APIs do not support seeding the underlying PRNG, this rule cannot be violated on iOS.

9 EVALUATION

The goal of this evaluation is twofold. First, by manually investigating the output of our framework and the source code of real-world applications, we identify and fix possible weaknesses in our approach. Second, applying our tool on a large number of closed-source applications, we identify security-critical misconceptions.

9.1 Method and Dataset

9.1.1 Manual Analysis. The objective of this step was primarily to test whether all components interact appropriately with each other and to refine the implementation where needed. Therefore, we applied the framework to open-source applications and checked the obtained results against the source code. Besides identifying opportunities for improvement, we also benefited from seeing what our security rules were able to (not) cover in a real-world scenario.

For this analysis, we downloaded 15 open-source applications from GitHub that use *CommonCrypto* for encryption. 8 of them were password managers, intended to protect user-entered credentials by means of cryptography. The remaining applications belonged to different categories, aimed at providing secure e-mail, data container, cloud storage, or messenger functionality.

We supplied the applications to the framework and analyzed them with respect to the defined security rules. We manually observed the analysis and improved the framework, e.g., by supplementing new *instruction semantics* that were not covered by the decompiler yet. After checking all security rules, the framework generated a report including all paths to statements that modified a specific parameter. We then verified and iteratively refined the accuracy of the analysis by studying the applications' source code. To facilitate this process, we leveraged a utility in Apple's IDE Xcode that enables the generation of call hierarchies for selected functions. A hierarchy is basically a subgraph of the call graph, containing only the nodes from which the targeted function was reached.

9.1.2 Automated Analysis. The objective of this step was to investigate whether iOS application developers know how to apply cryptographic APIs correctly. Inspired by the work of Egele et al. [16], we performed a similar empirical study for iOS applications.

We downloaded 634 free applications from the official iOS App Store. We focused on applications where the use of cryptography seems indispensable, e.g., password managers, (secure) messengers, cloud storage, and data containers, with more than 10,000 installations. Due to the fact that most were closed-source, we had to rely on the developer-provided descriptions to select those which might employ cryptography. After fetching them via iTunes, we used the tool *Clutch* [35] on a jail-broken iPhone to decrypt the applications. It turned out that 495 (78%) of the crawled applications included calls to the cryptographic API and were relevant for further analysis. Interestingly, the remaining set of 139 applications (without *CommonCrypto*) also included password managers and applications where the use of cryptography seemed appropriate. Aside from a faulty or absent implementation, this could be caused by the use of third-party libraries that implement crypto routines themselves.

After extracting all applications, we checked whether their library bindings indicate the usage of *CommonCrypto*. If this condition was fulfilled, we sequentially supplied the ARMv8 binaries to our framework. By inspecting the generated output reports, we ensured that any claimed security rule violation was indeed the consequence of a problematic execution path.

False Positives and False Negatives. Since our framework cannot miss calls to *CommonCrypto* methods, we avoid false negatives. In contrast, due to the nature of closed-source applications, we cannot formally exclude the existence of false positives. However, by manually examining the analysis reports, we also minimized false positives that could have occurred, e.g., if a backtracked value was a parameter to a function that had never been invoked.

9.2 Results

In total, we evaluated 495 closed-source and 15 open-source applications that included calls to *CommonCrypto*. Iteratively refining the components during manual analysis ensured that all open-source applications could be decompiled and inspected. Afterwards, for a total of 417 + 15 = 432 applications, the analysis workflow terminated successfully. As summarized in Table 2, the analysis of the remaining 78 closed-source applications failed due to one of three reasons. First, 7 applications contained only binaries for the ARMv7 platform, which are considered as deprecated by Apple and, hence, are not supported by our decompiler. Second, 9% or 46 applications could not be decompiled due to missing *instruction semantics*. Third, for 25 applications constraint solving ran out of memory.

9.2.1 Automated Analysis. Of 417 successfully inspected closed-source applications, we found that 82% or 343 applications violate at least one rule. Table 3 summarizes our observations of violated security rules. We discuss the findings in more detail below.

Rule 2: Do not use a non-random IV for CBC encryption. This was the most commonly violated rule: 69% or 289 applications used a cryptographically insecure IV with CBC encryption. Among

Table 2: Reliability for Closed-Source Applications

	Count	[%]
Downloaded from iOS App Store	634	
No *CommonCrypto* calls	139	22%
With *CommonCrypto* calls	495	78%
Binary only for ARMv7	7	1%
Not decompilable	46	9%
Out of memory	25	5%
Analyzable with *CommonCrypto* calls	417	84%

Table 3: Violations of security rules

Violated Rule	# Applications	[%]
Rule 2: Uses non-random IV	289	69%
Rule 3: Uses constant encryption key	268	64%
Rule 1: Uses ECB mode	112	27%
Rule 4: Uses constant salts for PBE	72	17%
Rule 5: Uses < 1,000 iterations (PBE)	49	12%
Applications with ≥ 1 rule violations	343	82%
No rule violation	74	18%

Table 4: Origin of constant secrets

	# Violations
Constant string used as encryption key	193
Constant password for PBKDF2	84
Hash value of constant string	18
Secret retrieved from *NSUserDefaults*	14
Constant key data	6
Applications violating rule 3	268

them, 92% or 265 applications specify a constant or NULL IV. The remaining 24 used the hash value of a constant string as IV.

Rule 3: Do not use constant encryption keys. 64% or 268 applications use constant data as key material. Although not immediately used for encryption, we also consider constant passwords passed to a key derivation function as misuse. Table 4 highlights the provenience of the key material. The total number of violations is higher than the number of applications due to the fact that some applications violate the rule multiple times. 193 keys were plain C strings that did not undergo any form of key derivation.

Rule 1: Do not use ECB mode for encryption. Overall, we found 27% or 112 applications that explicitly declared to use this mode of operation for symmetric encryption.

Rule 4: Do not use constant salts for PBE. We identified that 17% or 72 applications specified constant salt values as input to the key derivation function. This effectively undermines the protection of password-based encryption against table-based attacks.

Rule 5: Do not use fewer than 1,000 iterations for PBE. This was the least violated rule with only 12% or 49 applications applying less than 1,000 rounds in key derivation functions. Among them, 59% or 29 applications used exactly 1 round, 14% or 7 applications specified 100 iterations. The remaining 13 applications used other values below the threshold of 1,000.

9.2.2 Manual Analysis. While most rule violations were found where expected, the analysis also pointed out deficiencies in our concept. Relying on a context-insensitive pointer analysis means that the points-to information of a pointer is independent of its calling context and might also include wrong locations. This further results in spurious paths being followed during parameter backtracking. Nevertheless, besides wrong values (or values belonging to a different calling context), the output will also always include an actually correct execution path.

Subsequently, we exemplify the analysis process based on one—the *Damn Vulnerable iOS App (DVIA)* [45]—of the 15 open-source apps. We explain the security-critical weaknesses and contrast the source code with the results of our framework. DVIA is designed for penetration testing and includes common mistakes on purpose. Among other issues, it contains the kind of cryptographic misuse we are looking for. However, DVIA does not invoke functions of *CommonCrypto* directly but relies on a wrapper (*RNCryptor* [46]). Nevertheless, violations should be detected by our framework.

Constant IV and Salt Value. Our framework identified multiple paths that violate rules 2 and 4. In both rules, the input parameter should have been generated using a cryptographically secure random number generator. DVIA, however, calls `+[RNCryptor randomDataOfLength:]`, a function belonging to *RNCryptor*. For all function calls using an IV or salt value, the report included three separate paths, of which only the one using `SecRandomCopyBytes()` was considered secure. Verifying these results manually revealed that *RNCryptor* only relied on `SecRandomCopyBytes()` if this function was actually defined, which is always the case on iOS devices. However, due to the nature of static analysis, we also found two alternative execution paths that tried to read bytes from `/dev/random`. This could fail due to two reasons: first, if the file descriptor was not available, and second, if the number of bytes to read was zero. In both cases, the IV or salt would have consisted of NULL values. Although this represents a correctly identified rule violation, its security impact is negligible. Based on this observation, we learned that reports generated during the automated process should be manually inspected in order to confirm critical rule violations.

Constant Password for PBE. The automated analysis of DVIA further reported two different origins of a password used for key derivation. One path ends up at a call to `-[UITextField text]`, which indicates that the password was retrieved from a text field. This signifies no rule violation and was also not detected as such. At the end of the second path, a hard-coded string was found. As highlighted in Listing 1, the constant *Secret-Key* was directly passed to *RNEncryptor* where it was subsequently used as a password input for `CCKeyDerivationPBKDF()`. Our framework correctly uncovered this rule violation.

Listing 1: Constant password in DVIA

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

As can also be seen in the listing, the argument used as second parameter is an object with the name `kRNCryptorAES256Settings`. If set, *RNCryptor* performs the key derivation with 10,000 rounds. With regard to the corresponding rule, this does not represent a misuse, as has also been correctly determined by our framework.

Dead Code. The automated analysis reported rule violations by a function that was included in the binary but never called. Related to the generation of an IV for symmetric encryption, it also used the method `+[RNCryptor randomDataOfLength:]` to generate random data and caused the same paths to be emitted.

9.3 Limitations

As a result of analyzing 15 open-source applications by manual and automated means, we identified possible limitations that may affect the reverse-engineering process. In the following, we summarize the most relevant drawbacks we discovered during analyses.

Polymorphism. Under certain conditions, the pointer analysis might follow implausible function calls because of polymorphism. As a result, the call graph contains spurious edges that could have a negative influence on the accuracy of parameter backtracking. As exemplified in Listing 2, depending on `someCondition`, `object` has either type `ClassA` or `ClassB`. However, determining the actually used type can only be done inside the branches of the allocation statement. Due to the fact that our approach for pointer analysis is flow-insensitive, the points-to set of the subsequent variable `foo` includes both loc_A and loc_B. Assuming an instance method is now called using this variable, both types have to be considered.

Listing 2: Pointer Analysis with Polymorphism

```
@class BaseClass {
  - (void) foo;
  - (void) foobar;
}
@class ClassA : BaseClass {}
@class ClassB : BaseClass {
  - (void) bar;
}
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}
}
```

To represent polymorphism more accurately in the call graph, we would have to use a flow-sensitive pointer analysis. Practically, this is rarely an issue as both allocations would have to be assigned to the same variable and parameter backtracking only considers paths where a variable of interest is modified. Still, in theory, it could happen that a spurious path with a wrong callee is tracked.

C Arrays. By using Andersen's field-insensitive solution [3] for pointer analysis, different fields of an array or struct cannot be distinguished. Although, in general, this causes less precise analysis results, it usually does not affect parameter backtracking where arrays are passed to functions using only their base pointer.

User Interface Elements. Another identified issue affects missing type information that results in an incomplete call graph. As a consequence, the data flow between functions cannot entirely be modeled. This circumstance is primarily caused by calls to external libraries that cannot be resolved, e.g., in case of user interface events. Since we also cannot determine the function signatures and parameter types, backtracking them becomes infeasible.

Defining event listeners and actions for UI elements works either by creating the user interface directly in the code or using the Interface Builder in Xcode. With the latter option, elements are stored in *.nib* files that are parsed during application start. As our analysis only considers information retrieved from the binary, it is not aware of user interface actions declared elsewhere. In the context of static slicing, this might lead to situations where, e.g., an input parameter to the function `CCKeyDerivationPBKDF()` cannot be fully backtracked due to missing declarations of the associated *UITextField* object. Nevertheless, user input represents dynamic information and cannot be captured by static analyses anyways.

9.4 Discussion

In the manual evaluation scenario, we focused on refining the framework to cope with closed-source iOS applications. While most of the rule violations were found where expected, we also discovered cases that could not be handled correctly. In a context-insensitive pointer analysis, the points-to set of a variable is always independent of the underlying calling context. Consequently, parameter backtracking might traverse and report spurious paths. With regard to the overall analysis workflow, we recognized that it is crucial to restore function signatures and types from the binary during the decompilation step, to facilitate further analysis.

The automated analysis of closed-source applications revealed that 82% were subject to at least one security-critical implementation flaw. Besides these findings, inspecting the execution paths in all reports also revealed that specific rule violations often result from a similar misunderstanding of the intended API usage, perhaps due to insufficient documentation.

Alternatively to specifying the number of iterations for key derivation explicitly, we identified applications that rely on the system-provided method `CCCalibratePBKDF()` to compute the number of rounds with respect to the current device. Likewise, if *RNCryptor* was used as a wrapper for *CommonCrypto*, the number of rounds was typically set to 10,000 (the default setting).

Related to constant encryption keys, we repeatedly noticed an execution path that transformed a password to a key without using a genuine derivation function. Thereby, a password string of arbitrary length was either truncated to the block size of the used cipher or if too short, filled up with zero bytes. This behavior did not violate our rules and, thus, was also not reflected in the previously presented results. Still, it significantly weakens security by facilitating attacks on the encryption key.

10 CONCLUSION

A rising number of iOS apps claim to protect sensitive data by means of encryption. However, since the source code of most applications is not available, it remains unclear how cryptographic APIs have been employed. In this work, we developed a multi-step approach to facilitate such an analysis on iOS platforms. Instead of inspecting a low-level representation of a binary, we proposed a solution for a generically applicable decompiler that translates 64-bit ARMv8 binaries to LLVM IR code. By reconstructing lost information from the binary, we are able to precisely model control and data flow graphs for use with program slicing and parameter backtracking.

Based on this framework, we analyzed popular iOS applications in order to detect common misuse of security-critical APIs. We observed that 343 out of 417 investigated applications violate at least one security rule. This result does not only highlight the viability of our solution but also underlines that cryptographic misuse is a common issue in iOS applications.

REFERENCES

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Symposium on Software Testing and Analysis – ISSTA 1991*. ACM, 60–73.

[2] Ahmed Bougacha. n. d.. dagger – Binary Translator to LLVM IR. https://github.com/repzret/dagger. (n. d.). Accessed: February 2018.

[3] L. O. Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. DIKU, University of Copenhagen. (DIKU 94/19).

[4] Androguard. n. d.. Reverse engineering, Malware and goodware analysis of Android applications. https://github.com/androguard/androguard. (n. d.). Accessed: February 2018.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Programming Language Design and Implementation – PLDI 2014*. ACM, 259–269.

[6] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *State of the Art in Java Program Analysis – SOAP*. ACM, 27–38.

[7] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Programming Language Design and Implementation – PLDI 2003*. ACM, 103–114.

[8] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *IEEE Symposium on Security and Privacy – S&P 2015*. IEEE Computer Society, 931–948.

[9] David Binkley and Mark Harman. 2004. A Survey of Empirical Results on Program Slicing. *Advances in Computers* 62 (2004), 105–178.

[10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation – OSDI 2008*. USENIX Association, 209–224.

[11] Patrick P. F. Chan, Lucas Chi Kwong Hui, and Siu-Ming Yiu. 2012. DroidChecker: Analyzing Android Applications for Capability Leak. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*. ACM, 125–136.

[12] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2015. Evaluation of Cryptography Usage in Android Applications. In *Bio-inspired Information and Communications Technologies – BICT 2015*. ICST/ACM, 83–90.

[13] Xin Chen and Sencun Zhu. 2015. DroidJust: Automated Functionality-aware Privacy Leakage Analysis for Android Applications. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2015*. ACM, 5:1–5:12.

[14] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Privilege Escalation Attacks on Android. In *Information Security – ISC 2010 (LNCS)*, Vol. 6531. Springer, 346–360.

[15] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iRiS: Vetting Private API Abuse in iOS Applications. In *Conference on Computer and Communications Security – CCS 2015*. ACM, 44–56.

[16] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Conference on Computer and Communications Security – CCS 2013*. ACM, 73–84.

[17] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed System Security Symposium – NDSS 2011*. The Internet Society.

[18] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Programming Language Design and Implementation – PLDI 1994*. ACM, 242–256.

[19] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Symposium on Operating Systems Design and Implementation – OSDI 2010*. USENIX Association, 393–407.

[20] William Enck, Damien Octeau, Patrick D. McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *USENIX Security Symposium 2011*. USENIX Association.

[21] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Conference on Computer and Communications Security – CCS 2012*. ACM, 50–61.

[22] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium 2011*. USENIX Association.

[23] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Foundations of Software Engineering – FSE 2014*. ACM, 576–587.

[24] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In *Artificial Intelligence and Security – AISec*. ACM, 45–54.

[25] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing – TRUST 2012 (LNCS)*, Vol. 7344. Springer, 291–307.

[26] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection. In *Mobile Systems – MobiSys 2012*. ACM, 281–294.

[27] "grievejia". n. d.. andersen – Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen. (n. d.). Accessed: February 2018.

[28] Ben Hardekopf and Calvin Lin. 2007. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Static Analysis Symposium – SAS 2007 (LNCS)*, Vol. 4634. Springer, 265–280.

[29] Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Programming Language Design and Implementation – PLDI 2007*. ACM, 290–299.

[30] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Programming Language Design and Implementation – PLDI 2001*. ACM, 254–263.

[31] Julien Henry, David Monniaux, and Matthieu Moy. 2012. PAGAI: A Path Sensitive Static Analyser. *Electr. Notes Theor. Comput. Sci.* 289 (2012), 15–25.

[32] Michael Hind and Anthony Pioli. 2001. Evaluating the Effectiveness of Pointer Alias Analyses. *Sci. Comput. Program.* 39 (2001), 31–55.

[33] Susan Horwitz, Thomas W. Reps, and David Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Prog. Lang. Syst.* 12 (1990), 26–60.

[34] Jiri Slaby. n. d.. LLVMSlicer – Static Slicer for LLVM. https://github.com/jirislaby/LLVMSlicer. (n. d.). Accessed: February 2018.

[35] "Kim Jong Cracks". n. d.. Clutch – Fast iOS executable dumper. https://github.com/KJCracks/Clutch. (n. d.). Accessed: February 2018.

[36] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2014. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Network and System Security – NSS 2014 (LNCS)*, Vol. 8792. Springer, 349–362.

[37] LLVM. n. d.. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html. (n. d.). Accessed: February 2018.

[38] LLVM. n. d.. TableGen. http://llvm.org/docs/TableGen/. (n. d.). Accessed: February 2018.

[39] LLVM. n. d.. The LLVM Target-Independent Code Generator. http://llvm.org/docs/CodeGenerator.html. (n. d.). Accessed: February 2018.

[40] ARM Ltd. 2013. Procedure Call Standard for the ARM 64-bit Architecture (AArch64). http://infocenter.arm.com/. (2013).

[41] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments – VSTTE 2012 (LNCS)*, Vol. 7152. Springer, 146–161.

[42] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. In *Software Engineering Symposium on Practical Software Development Environments – SDE 1984*. ACM, 177–184.

[43] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.* 30 (2007), 4.

[44] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society.

[45] Prateek Gianchandani. n. d.. DVIA – Damn Vulnerable iOS App (DVIA). https://github.com/prateek147/DVIA. (n. d.). Accessed: February 2018.

[46] RNCryptor. n. d.. RNCryptor – CCCryptor (AES encryption) wrappers for iOS and Mac in Swift. https://github.com/RNCryptor/RNCryptor. (n. d.). Accessed: February 2018.

[47] Marc Shapiro and Susan Horwitz. 1997. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Principles of Programming Languages – POPL 1997*. ACM Press, 1–14.

[48] Marc Shapiro and Susan Horwitz. 1997. The Effects of the Precision of Pointer Analysis. In *Static Analysis Symposium – SAS 1997 (LNCS)*, Vol. 1302. Springer, 16–34.

[49] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages – POPL 1996*. ACM Press, 32–41.

[50] Mingshen Sun, Tao Wei, and John C. S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Conference on Computer and Communications Security – CCS 2016*. ACM, 331–342.

[51] The Charles Stark Draper Laboratory, Inc. n. d.. fracture – An architecture-independent decompiler to LLVM IR. https://github.com/draperlaboratory/fracture. (n. d.). Accessed: February 2018.

[52] Frank Tip. 1995. A Survey of Program Slicing Techniques. *J. Prog. Lang.* 3 (1995).

[53] Trail of Bits. n. d.. mcsema – Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. https://github.com/trailofbits/mcsema. (n. d.). Accessed: February 2018.

[54] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - A Java Bytecode Optimization Framework. In

Conference of the Centre for Advanced Studies on Collaborative Research – CASCON 1999. IBM, 13.

[55] Mark Weiser. 1981. Program Slicing. In *International Conference on Software Engineering – ICSE 1981.* IEEE Computer Society, 439–449.

[56] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Programming Language Design and Implementation – PLDI 1995.* ACM, 1.

[57] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *World Congress on Software Engineering – WCSE 2012.* 101–104.

[58] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John C. S. Lui. 2015. Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates. In *Asia Conference on Computer and Communications Security – AsiaCCS.* ACM, 463–474.

A DECOMPILATION

As shown in Algorithm 1, the decompilation itself consists of two steps. First, all *MachineInstructions* are grouped into basic blocks that resemble the control flow (lines 1 to 16). If an instruction is a control flow statement, its target address is determined. The second step iterates over *MachineFunctions* and its basic blocks of disassembled instructions and decompiles them to LLVM IR using the semantics (lines 17 to 24). If the exit point of a basic block is reached, a terminator instruction is added to the LLVM IR code that defines the subsequent basic blocks or returns from the function.

Algorithm 1: Decompilation workflow

```
1  for MI in MachineInstruction do
2      if MI.address in FunctionStarts then
3          createMachineFunction(MI.address)
4          createMachineBasicBlock(MI.address)
5      end
6      DIS ← disassemble(MI)
7      if branchInstruction(DIS) then
8          if not callInstruction(DIS) then
9              if getTarget(DIS) then
10                 createMachineBasicBlock(getTarget(DIS))
11             end
12             createMachineBasicBlock(MI.address +
               InstructionSize)
13         end
14     end
15     addToMachineBasicBlock(DIS)
16 end
17 for MF in MachineFunctions do
18     switchToFunction(MF)
19     createAllBasicBlocks(MF)
20     for MBB in MF.MachineBB do
21         switchToBasicBlock(MBB)
22         decompileInstruction(MBB)
23     end
24 end
```

B POINTER ANALYSIS

B.1 Iterative Constraint Generation

As shown in Algorithm 2, we extend the original algorithm by Anderson [3] to differentiate between call instructions and others. After generating constraints for all regular statements (line 5), the subsequent loop solves the constraints by propagating points-to information through the program. The gained knowledge is then used to update the constraints for call statements. This step is repeated until no new edges are added to the call graph.

Algorithm 2: Constraint generation

```
1  for I in Instructions do
2      if isCallInstruction(I) then
3          addInstruction(I, CallInstructions)
4      else
5          generateConstraints(I)
6      end
7  end
8  repeat
9      solveConstraints()
10     for I in CallInstructions do
11         updateConstraints(I)
12     end
13 until no new constraints added
```

B.2 Objective-C Peculiarities

Algorithm 3 describes how the original call graph is restored using the points-to sets of the class or object (PtsTo_a), and method name (PtsTo_b) parameters passed to `objc_msgSend()`. The algorithm iterates over the values of the two parameters and based on the locations they point to, it determines what method can be called. The first parameter has to point either to a class info location or to a dynamically allocated location. If the value points to a class info location, a class method will be called. If it points to a dynamically allocated memory location and is annotated with type information, an instance method is called. The result is a call graph that also models all calls that are performed at runtime via `objc_msgSend()`.

Algorithm 3: Call graph reconstruction using points-to sets

```
   Input: CallInstruction, $PtsTo_a$, $PtsTo_b$
1  for $loc_a$ in $PtsTo_a$ do
2      ClassMethod ← false
3      if PointsToClassInfo($loc_a$) then
4          ClassMethod ← true
5      end
6      Type ← GetTypeName($loc_a$)
7      for $loc_b$ in $PtsTo_b$ do
8          if not PointsToClassInfo($loc_a$) then
9              continue
10         end
11         Selector ← GetSelectorName($loc_b$)
12         if not KnownMethod(Type, Selector, ClassMethod) then
13             continue
14         end
15         if HasEdge(CallInstruction, Type, Selector, ClassMethod)
           then
16             continue
17         end
18         AddEdge(CallInstruction, Type, Selector, ClassMethod)
19     end
20 end
```