University of Passau

Department of Informatics and Mathematics

Master's Thesis

TypeChef meets SPL$^{\text{LIFT}}$ Interprocedural Data-Flow Analysis of Configurable Software Systems

Author:

Andreas Janker

December 5, 2016

Advisors:

Dr. rer. nat. Alexander von Rhein, Andreas Stahlbauer

Chair of Software Engineering

Correctors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering

Prof. Christian Lengauer, Ph.D.

Chair of Programming

Janker, Andreas:

TypeChef *meets* SPL$^{\text{LIFT}}$ - *Interprocedural Data-Flow Analysis of Configurable Software Systems*

Master's Thesis, University of Passau, 2016.

Abstract

Today's commonly used cryptographic algorithms for enforcing the confidentiality of data and communications are considered as sound. When implemented correctly, for most of the state-of-the-art cryptographic algorithms there are no known practical attacks that would allow a third party to read encrypted data.

However, the most popular cryptographic library implementing these algorithms - OPENSSL - is frequently affected by highly problematic security issues, most notably by the infamous Heartbleed bug in 2014. These issues are mostly caused by flaws in the actual implementation. OPENSSL is a highly configurable software written in C with heavy usage of the C PREPROCESSOR (CPP) to implement variability, resulting in a huge number of variants. This leads to system variants that contain bugs which are not detected by current approaches.

Recent academic advances allow us to fully analyze such code: the tool TYPECHEF enables us to parse C code with preprocessor directives and provides intraprocedural data-flow analysis of the parsed code. The framework SPL$^{\text{LIFT}}$ is able to run interprocedural data-flow analysis on variable Java Code. In this thesis we will combine both tools for applying SPL$^{\text{LIFT}}$'s capabilities on real-world, large-scale C code. Based on this combination, we develop an interprocedural variability-aware taint analysis and evaluate it on two cryptography libraries.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

Since the very first days of software engineering, developers face the challenge of software defects (more commonly known as *bugs*). Although it is said, that one of the first software defects in history was an actual bug[1], in modern software systems a *software defect* is defined as a false result or unexpected behavior of the software system caused by incorrect instructions in the source code [noa90].

Despite the presence of defects, today's society relies heavily on software systems and its dependency on such systems is growing as we are currently witnessing the rise of the *Internet of Things* and see the beginning of the age of *autonomous driving*. Unfortunately, a defect in an autonomous driving software system may put people's live at risk. For this reason, preventing faults is a very import task in the development process of software systems. As a result, academia and industry developed many different techniques, such as automated testing or code review, to detect and consequently avoid defects during code-development time. One of these techniques is *static code analysis*. Static code analysis performs analysis strategies aiming to detect potential defects on the software system's source code without actually executing it [NNH15]. It is a very powerful tool to detect programming errors and is in practice an essential part in the daily life of an developer: for example development environments, such as INTELLIJ, or compilers such as GCC, support the developer by issuing warnings for potentially uninitialized variables. Furthermore, in some security-critical fields, like aviation, static code analysis is a prerequisite before a software system can go into productive use in these fields [oT-FAA15].

Even though static code analysis is a prerequisite in some fields, it is still imperfect. OPENSSL, a cryptography library implementing all essential state-of-the-art cryptographic functions, which is widely used to secure the internet's communication, claims to use static code analysis to ensure its code quality[2]. Nevertheless, this library has been affected by some major security issues in the past. These issues have not been caused by flaws in the used cryptography algorithms but rather by defects in the library's source code. One of them was the infamous *Heartbleed Bug*. Briefly described: by sending a malformed heartbeat message to a webserver using a vulnerable version of the OPENSSL software, OPENSSL exposed parts of the memory to the attacker in its response. The leaked data

[1] The anecdote says, in 1945 a moth did block a relay in one of the first electro-mechanical computers causing erroneous behavior of the machine. To solve this issue, the moth has been removed and the following entry was entered into the logbook: "First actual case of bug being found."

[2] As described here: https://wiki.openssl.org/index.php/Static_and_Dynamic_Analysis

contained confidential information like user credentials, private keys, and other sensitive data[3]. The bug was part of the library for almost two years before being detected in April 2014. As a first workaround, prior a fixed version was released, users of the library were advised to recompile OPENSSL without the affected feature as only one certain variant of the library was affected[4].

The previously described defect stands as an example for current shortcomings of state-of-the-art static code analysis approaches. OPENSSL is written in the programming language C with heavy usage of the C PREPROCESSOR. With the use of the C preprocessor one is able to configure during compile-time which parts of the source code, annotated with preprocessor directives, are ex- or included in the compiled product. This process is called conditional-compilation and because of its extensive usage to implement variability in OPENSSL, the library is defined as a highly configurable software system. This fact leads to a huge number of variants: OPENSSL has 589 configuration options, with which 6.5 x 10^{175} variants can be derived [LJG+15]. This huge number leads to system variants that contain bugs which are not detected by current static code analysis approaches as they would have to check each variant individually.

1.1 Problem Statement and Contribution

Figure 1.1 illustrates a simplified version of a configurable software system written in C with usage of the C PREPROCESSOR. This example is adopted from the actual high level interface EVP to access OPENSSL's cryptographic functions[5]. It consists of a main file as seen in Listing 1.1, which contains two functions `ctx_init` and `ctx_do`, both are externally defined in Listing 1.3 and respectively in Listing 1.4 and linked to the main file during compile-time. Furthermore, the function `ctx_init` assigns, depending on which variant (wether configuration option A is enabled or not) is chosen, one of two different functions, `cipher1` or `cipher2`, to a function pointer `c`, which gets dereferenced later in the function `ctx_do`. `cipher1` and `cipher2` represent in OpenSSL the concrete implementation of a certain cryptography algorithm like AES or Blowfish. Current static analysis approaches would have to analyze all 16 possible variants of the example software system individually, to recognize that the value of the variable `secret` will be printed out in two variants ($A \wedge \neg B \wedge C$, $\neg A \wedge \neg D$) by the `printf` instruction. In practice, this approach is infeasible for real-life software systems, as potentially millions of possible variants can be derived [LAL+10].

Recent academic advances allow us to fully analyze such code: the tool TYPECHEF enables us to parse C code with preprocessor directives and provides intraprocedural control-flow and data-flow analysis of the parsed code [KGR+11, LvRK+13, Lie15]. The framework SPL$^{\text{LIFT}}$ is capable to run data-flow analysis, which are formulated as interprocedural, finite, distributive, subset (*IFDS*) problems, on variable Java source code [BTR+13].

[3] For a detailed description of this vulnerability we refer the interested reader to: [DKA+14]

[4] FAQ-Entry: "How can OpenSSL be fixed?" at the official bug website: `http://heartbleed.com`

[5] The interface itself is described in detail here: `https://wiki.openssl.org/index.php/EVP`

```c
#include <minimalLinking.h>

int main() {
    struct cipher_ctx *c = malloc (sizeof(struct cipher_ctx));

#ifdef A
    ctx_init(c, &cipher1);
#else
    ctx_init(c, &cipher2);
#endif

    int secret;
    secret = 666;

    int sink = ctx_do(c, secret);
    printf("%i\n", sink);

    return 0;
}
```

Listing 1.1: main.c

```c
#include <minimalLinking.h>

int cipher1(int i) {
   int r, x = 1;

#ifdef B
   i = x;
#endif

   if (i < 0) {

#ifdef C
    r = i;
#else
    r = x;
#endif

  }

   return r;
};

int cipher2(int i) {

#ifdef D
    i = 0;
#endif

    return i;
};
```

Listing 1.2: ciphers.c

```c
#include <minimalLinking.h>

void ctx_init(struct cipher_ctx * c, int (*f)(int)) {
    c->func = f;
    return;
}
```

Listing 1.3: init.c

```c
#include <minimalLinking.h>

int ctx_do(struct cipher_ctx *c, int value) {
    int result = c->func(value);
    return result;
}
```

Listing 1.4: do.c

Figure 1.1: Running example of a configurable software system: secret is printed if B is disabled while A and C are enabled or A and D are disabled

In our approach we will combine both tools, for applying SPL$^{\text{LIFT}}$'s capabilities on large-scale C code. Combined together, these tools are able to run interprocedural data-flow analysis simultaneously on every variant of configurable software systems in C instead of analyzing each variant individually.

To do so, we will extend TYPECHEF with capabilities to resolve linked function names, trace the possible destinations of function pointers and provide an interprocedural control-flow graph. Further, we adjust and consequently improve the lifting mechanism of the tool SPL$^{\text{LIFT}}$ to be fully compatible with the control-flow graph concept of the TYPECHEF infrastructure and eliminate all of the tool's known limitations. Based on this combination, we develop, as a *proof of concept*, a simple, assignment-based, interprocedural taint analysis and apply it on the implementation of the AES-encryption algorithm of two different configurable real-world open-source cryptography libraries: MBEDTLS and OPENSSL.

1.2 Structure of the Thesis

In Chapter 2, we explain the background topics of this thesis. First, we introduce and explain configurable software systems in general. We focus on systems using the C Preprocessor (CPP) to implement variability and discuss shortly the benefits and disadvantages of this technique. Second, we introduce the concept of static code analysis. In particular, we explain the necessary program representation to perform this kind of analysis, namely abstract syntax trees, control-flow and data-flow graphs. Further, we highlight the idea of taint analysis as an example static code analysis method.

Chapter 3 explains the steps we performed to make the parsing infrastructure TYPECHEF meet SPL$^{\text{LIFT}}$'s upfront requirements. We describe the approach of each extension we had to add to TYPECHEF in order to provide an interprocedural control-flow graph. Finally, all modifications to the intermediate source code representation are documented.

In Chapter 4, we present the used data-flow analysis framework used within SPL$^{\text{LIFT}}$. Afterwards, we formulate the underlying data-flow analysis problem for taint checking within this framework.

Afterwards in Chapter 5, we present SPL$^{\text{LIFT}}$'s original lifting strategy for data-flow analysis problems on every possible variant of a configurable software systems simultaneously. In this section, we describe why the proposed lifting strategy is not applicable using the TYPECHEF infrastructure and present an adapted approach which is more precise and eliminates all known shortcomings of SPL$^{\text{LIFT}}$.

In Chapter 6, we first show the correctness and scalability of the developed tool-combination in a series of experiments. Second, we apply the developed static analysis strategy on the implementation of the AES-encryption algorithm of two different highly configurable real-world open-source cryptography libraries: MBEDTLS and OPENSSL. Finally, we discuss the result of the performed analysis.

In Chapter 7 we present an overview of current research, ongoing and finished, regarding static code analysis.

2. Background

This chapter describes background knowledge that is fundamental to our challenges and to our approach to statically analyze variable C code.

2.1 Configurable Software Systems

Most software systems provide mechanism to be configured according to user's requirements. This is common practice, as different application scenarios (e.g., different hardware environments) require different variants of a software system. For example, the LINUX KERNEL is in productive use on servers as well as on mobile devices. Each platform offers a different hardware environment and needs to provide different functionalities: a webserver may not support touch input devices, but support load-balancing technologies. As a result, the LINUX KERNEL offers over 13000 different configuration options [PPB$^+$15]. In general, a configurable software system is a system that can be custom-tailored to fulfill various roles in different environments and application scenarios [vR16].

Next, we will introduce an existing mechanism to implement the functionality of configuration options.

2.1.1 Implementing Configurable Software Systems with the C Preprocessor

The C PREPROCESSOR (CPP) is a macro processor that is used for source code transformation [KR88]. By the means of different directives the preprocessor provides functionality for file inclusion, text substitution, conditional compilation, line control, and diagnostics. Its syntax is independent from the underlying programming language. The preprocessor is intended to be used in the source code of the C programming language family (C, C++, C# and Objective C), however it can be abused to process other text files. It is generally executed before the actual compilation task is performed.

In this thesis we focus on conditional compilation, because of its widespread usage for the implementation of configurable software. However, before we explain the process of conditional compilation itself, we introduce two mechanisms of the CPP first:

Macro A macro is a named source code or text fragment. Macros can be divided into two types: (1) *object-like* macros and (2) *function-like* macros. Both types are created by the directive `#define` followed by an identifier, in case of a function-like macro

```c
#define PI 3.14 // object-like

#define PERIMETER(r) (2*r*PI) // function-like
```

Listing 2.1: Preprocessor macros

directly followed by a parameter list and the replacement as shown in an example in Listing 2.1. After a macro has been defined, the preprocessor replaces all subsequent appearances in the source code by the macro replacement list. This process is called macro expansion.

Fileinclusion The directive `#include <fileToInclude>` includes an external (header) file in C/C++. A header file is typically used for collecting externally defined declarations and macros in a central place which are shared between several source files. The include process is precisely equivalent to the process of directly copying the content of the file to include into the source code file. A common usage of this directive in the C programming language is to include system header files, which provide for example basic I/O functions.

2.1.2 Conditional Compilation

Conditional compilation directives, also called *conditionals* in the offical documentation, allow the programmer to advise the preprocessor which chunk of code to include or exclude in the output passed to the compiler. Its syntax follows the classical `if-then-else` construct. A conditional group in the C PREPROCESSOR begins with one of the following directives: `#if`, `#ifdef` or `#ifndef` and ends with the directive `#endif`. The following CPP *conditional directives* exist to implement conditional compilation [KR88, Jan13]:

`#ifdef`, `#ifndef` One of the two possible starting conditionals. In Lisitng 2.2 on the following page we see the simplest way of using this directive. The *controlled source code* by this directive will be included (`#ifdef`) or excluded (`#ifndef`) in the preprocessed output stream to the compiler if and only if MACRO has been defined previously by the CPP-directive `#define MACRO`.

`#if` This conditional evaluates the value of an arithmetic expression following the directive. The value of this arithmetic expression is calculated at preprocessing-time and the *controlled source code* is included in the output if the arithmetic expression evaluates to nonzero. An example usage of this conditional is shown in Listing 2.3.

`defined` To test in arithmetic expressions if the name of a macro is defined, the directive `defined` is used. If a macro has been defined, it will evaluated during preprocessing to 1, otherwise to 0. Thus, `#if defined MACRO` is precisely equivalent to `#ifdef MACRO`.

```
#ifdef MACRO
controlled source code
#endif
```

Listing 2.2: Example usage of the conditional #ifdef

```
#define VALUE 5
#if (0 < VALUE)
controlled source code
#endif
```

Listing 2.3: Example usage of the conditional #if

`#else` This directive provides an alternative that is included into the preprocessed output if the previous conditional directives, `#if`, `#ifdef`, `#ifndef` or `#elif`, fail.

`#elif` Analogues to the previous described `#else` - directive, this conditional provides an potential alternative chunk of code to include in the output. It is used to provide more than one alternative as this conditional evaluates in the same matter as the directive `#if` arithmetic expressions. It stands for term `else if` in common programming languages.

`#endif` This directive ends a conditional group.

2.1.2.1 Build Systems and Configuration Knowledge

In practice, configurable software systems using the CPP and conditional compilation to implemented configuration options provide a user-friendly interface for configuration. As this process takes place during compile-time, these systems are called *Build Systems*. Well-known and widely used build-systems are for example GNU Build System[1] and KBuild[2]. Both tools offer an interface in which the user chooses desired configuration options. Afterwards, based on the chosen configuration, the corresponding macro directives are either defined or undefined and the previously described process of conditional compilation can start.

However, build systems commonly take dependencies between certain configuration options into consideration, as not every possible configuration variant is valid: e.g. a configuration options may exclude one other or a configuration option requires another option. The tool chain of this thesis is able to respect these dependencies during computation. For our case studies, which we will use in Chapter 6, these dependencies have been extracted from the underlying build systems as *configuration knowledge*. The constraints between configuration options are represented in a tree-like structure called *Feature Model* [ABKS13]. Without considering the feature model our tool chain would be unable to distinguish results for valid configurations from those for invalid ones.

2.1.2.2 Disadvantages of Conditional Compilation

While conditional compilation is extremely flexible and easy to use at first glance, in real-life software systems it leads to code that is very hard to read and to maintain [SLSA14, SSF+12], as shown in the real-life source code excerpt in Listing 2.4. For a

[1] https://www.gnu.org/software/automake/automake.html

[2] https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

```
#if defined(__GLIBC__)
// additional lines of code
#elif defined(__MVS__)
    result = pty_search(pty);
#else
#ifdef USE_ISPTS_FLAG
    if (result) {
#endif
        result = ((*pty = open("/dev/ptmx", O_RDWR)) < 0);
#endif
#if defined(SVR4) || defined(__SCO__) || defined(USE_ISPTS_FLAG)
    if (!result)
        strcpy(ttydev, ptsname(*pty));
#ifdef USE_ISPTS_FLAG
        IsPts = !result;
}
#endif
#endif
```

Listing 2.4: Example of the `#ifdef` - hell in XTERM

long time, researchers have heavily criticized the usage of conditional compilation, with characterization of such code as `#ifdef` - hell [LST+06]. Many problems with C - code that prevent the full support of current static analysis approaches can be traced back to the usage of CPP directives [MKR+15, LKA11].

Despite all shortcomings and criticism, conditional compilation is, and probably will remain, a technique which is used in a wide range of software systems in industry as well as in open-source software [HZS+16].

2.2 Static Code Analysis

Static code analysis performs analysis strategies aiming to detect potential defects on the software system's source code without actually executing it [NNH15]. Despite being mostly unnoticed by developers, static code analysis is a heavily used technique in practice. In the same fashion word processing programs highlight spelling errors, modern Integrated Development Environments (IDE), like Eclipse, IntelliJ or XCode, support the developer during coding-time by highlighting potential syntax- or type errors. But static code analysis is not limited to syntax- and type-checking: a wide range of different analysis exists in academia and industry to cover not only "spelling" errors, but support the developer in the means of program verification, bug finding, and security review [CW07]. For example, Liebig et al. use the TypeChef infrastructure, which we also use in this thesis, to implement various intraprocedural static data-flow analyses techniques on C source code to detect defects according to the Secure Coding Guidelines for C [LvRK+13, Sea05].

As mentioned before, static analysis does operate on the source code without executing it. This brings a major benefit, as static analysis is able to indicate potential code smells during coding-time and before execution-time of the software system. However, static analysis can not incorporate external program arguments and can therefore only reveal potential defects which are independent from external influences. For this reason, static code analysis techniques are known to produce *false positives* and *false negatives*. For static code analysis a false positive indicates an error when it is actually none, whereas a false negative does not indicate the presence of a defect despite its presence. Using static code analysis, one should always keep in mind that this technique can provide proof for the presence of an potential defect but can not provide evidence of the absence of an bug. Despite these shortcomings, static code analysis is a mandatory perquisite to ensure the software quality in security critical fields like aviation [oT-FAA15].

An alternative code analysis technique is *Model Checking* [BK08]. Like static code analysis, model checking also analyzes the source code without actually executing it. In contrast to static code analysis however, model checking transforms the program's behavior into a finite-state machine and checks if the resulting state-machine fulfills a given set of specifications (e.g. the absence of deadlocks).

Different techniques exist to perform static analysis. As static analysis works on the raw source code it reuses technologies derived from compiler construction [NNH15], namely *Abstract Syntax Trees*, *Control-Flow Graphs* and *Data-Flow Analysis*. These techniques are commonly combined for specific static code analysis methods and will be explained in the following section.

2.2.1 Variational Abstract Syntax Tree

Analysis strategies for programming languages require a data structure which represents the syntactic structure of the source code. For this purpose, commonly a tree based data-structure called *Abstract Syntax Tree* (AST) is chosen, which represents the syntactic structure of the source code of a certain programming language in tree form, while excluding unnecessary syntactic details [Jon03]. Each source code construct is denoted as a tree node.

Due to the fact that C PREPROCESSOR directives are not a part of the C programming language [KR88], classic AST representations for C code generally represent preprocessed code and consequently only one possible variant of a configurable software system written in C. Since generating individual ASTs for all possible variants of the C source code is infeasible, but mandatory for a complete analysis, a single variational AST is required. To do so, we enrich the classic abstract syntax tree representation with information in each tree node about its presence or absence in the compilation unit according its surrounding `#ifdef` directives in the source code as *presence-condition* (c.f. Section 2.1.1).

Even though parsing C code, especially in the presence of preprocessor directives, is a highly difficult task, recent advances in academia made such a data structure available for us. The research parsing infrastructure TYPECHEF by Kästner et al. [KGR+11, KGO11] is able to parse un-preprocessed C code and provides a sound and complete representation of C source code annotated with `#ifdef` directives in a so called *variational abstract syntax tree* [WKE+14].

```
 1  ...
 2  int cipher1(int i) {
 3     int r, x = 1;
 4
 5  #ifdef B
 6     i = x;
 7  #endif
 8
 9     if (i < 0) {
10
11  #ifdef C
12      r = i;
13  #else
14      r = x;
15  #endif
16
17    }
18
19     return r;
20  };
21  ...
```

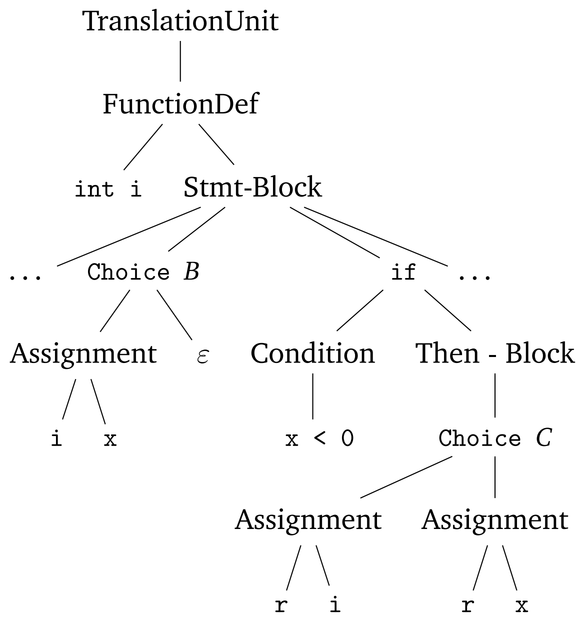


Figure 2.1: Minimalistic configurable source code fragment extracted from the running example with two features and its corresponding AST

The structure of the variational AST representation is analogous to the structure of a "classic" AST. However, to express variability in the tree, an additional node, called `Choice` node, is introduced into the tree's syntax. This node represents alternative subtrees according to the CPP-conditionals and their conditions in the original source code. Figure 2.1 shows a minimalistic source code example and the corresponding variational abstract syntax tree. In the source code example both statements in Line 6 as well as in Lines 12ff are variable by the means of conditional compilation. For example, the assignment to `x` in Line 6 is only part of the the preprocessed source code if B is defined. This fact is represented in the variability-aware abstract syntax tree by the tree node `Choice(`B`, Assignment, `ε`)`, where the node indicates under Condition B the assignment is presented, otherwise it is absent without any alternative. Analogous in Line 11ff we see a conditional block. This time however an alternative is present if the condition of the `#ifdef` - directive does not hold, which is represented by a second non-empty leaf of the corresponding choice node in the abstract syntax tree: `Choice(`C`, Assignment, Assignment)`.

As described in Section 2.1.1, C PREPROCESSOR directives are also used for file inclusion and macro definition. The TYPECHEF parsing infrastructure resolves both directives. Files referenced by the `#include` directive are parsed as well and become a part of the resulting AST. Macros get expanded and their occurrences in the source code are replaced in the abstract syntax tree representation according to their definition.

2.2.2 Variational Control-Flow Graph

Solely based on the previously described AST, static code analysis is not able to inspect the control- or data-flow of a software system. Analysis strategies which take the execution path of individual program instructions (such as statements) into consideration, require a data-structure that represents the possible execution path of the program. Such data-structures are *Control-flow Graphs* (CFGs). A CFG is a directed graph which describes all possible execution paths through a program during its execution-time [All70, NNH15].

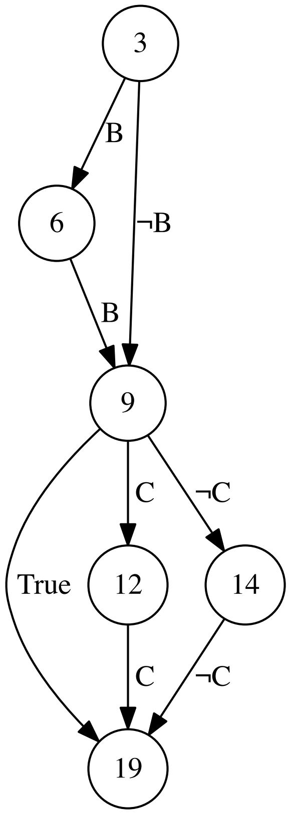


Figure 2.2: A variability-aware control-flow graph for the running example of Figure 2.1

In the graph representation, a node denotes a single program instruction, e.g. assignments or function calls. Directed edges exists between two nodes if control can flow from the statement represented by the source node to the statement represented by the target node. A single node may have several different in- and outgoing edges, as control-flow statements, such as `if-then-else` or `loops`, branch the program's execution path. The actual control-flow conditions which lead to branching are normally not evaluated by the CFG, because not all necessary run-time informations are available. Consequently, a control-flow graph is only a approximation of the program's behavior with potential dead paths within the graph.

Unfortunately, in the presence of conditional-compilation directives, the successors of a node may differ according to the chosen configuration. To project all possible system variants into a single CFG representation, we have to encode variability into the graph likewise the previously described AST. In classic textbook definitions, a CFG is generated by computing all possible successor statements of a statement:

$$succ : CFGStmt \rightarrow List[CFGStmt]$$

As we are already aware of the presence-condition of every single statement in the AST, we are able to compute the corresponding presence-condition of the resulting edge. In order to represent variability in the graph, we annotate each edge with the calculated presence-condition. In the successor function each potential successor node is annotated according its individual edge presence-condition:

$$succ : CFGStmt \rightarrow List[V[CFGStmt]]$$

The resulting CFG for the running example of Figure 2.1 is illustrated in Figure 2.2. The first statement in Line 3 has two potential successors, either the assignment in Line 6 or the `if`-statement in Line 9, based on whether configuration option *B* is chosen or not. In the visualized graph representation (node numbers refer to the corresponding line of code), we see two outgoing edges from the first node, one reaching the node representing the assignment in Line 6, the other one going to the node for Line 9. The edge between Node 3 and Node 6 is annotated with the presence-condition *B* as the successor is only reachable if configuration option *B* is set. Analogous, Node 9 has three outgoing edges: (1) if the condition of the corresponding if statement does not hold, Line 19 is directly reached, (2) if option *C* is set Line 12 is the successor, and finally (3) if option *C* is not set, Line 14 is reached.

The described variational CFG is already provided by the TypeChef parsing infrastructure.

2.2.3 Data-Flow Analysis

Data-flow analysis is a static code analysis technique which investigates dependencies and relationships between variables of a computer program [NNH15, KSS09]. Interesting data-flow properties are commonly gathered and propagated along the control-flow graph. Well-known classic data-flow analysis are for example *Reaching Definitions* or *Liveness Analysis*. In this thesis we will implement data-flow analysis techniques for function-pointer destinations and a simple taint-analysis.

Various data-flow analysis techniques differ in the means of precision. In the scope of this thesis we are using several data-flow analyses with different properties, which we will introduce next [NNH15, MRR04]:

Intraprocedural vs. Interprocedural Analysis

Intraprocedural analysis analyzes individual procedures of a program in isolation. This technique assumes that a single procedure does not have any side-effects or assumes alternatively the worst-case scenario.

In contrast to intraprocedural analysis, interprocedural analysis is done across an entire program. It hereby additionally analyzes relationships between call- and callee-procedures as well.

Flow-Sensitive vs. Flow-Insensitive

A flow-sensitive analysis takes the control-flow and therefore the execution order between program instructions into account in its computation. Flow-insensitive analysis however does ignore the execution order of a program and is as a logical consequence more imprecise.

Contex-Sensitive vs. Contex-Insensitive

```
1  int foo (int *p) {
2     return *p;
3  }
4
5  int main () {
6    int a, b, c, d;
7    a = 0;
8    b = 1;
9
10   c = foo(&a);
11   d = foo(&b);
12
13   return c + d;
14 }
```

Listing 2.5: Example program for distinguishing context-sensitivity

A context-sensitive analysis considers the calling context when analyzing the target of a function call and analyzes each call separately according to its context. Context-insensitive analysis does not distinguish between different invocation contexts for a procedure. The sample program in Listing 2.5 illustrates the difference between both analysis properties. A context-sensitive analysis would compute both calls to function `foo` in Line 10 and 11 separately. The resulting values for variable `c` and `d` would correctly differ. In contrast, a context-insensitive analysis does not distinguish between the two different invocation contexts of `foo` and would compute a false result with both variables `c` and `d` pointing to identical values.

Field-Sensitive vs. Field-Insensitive

Some data-structures in programming languages, like objects or structures, have field values. A field-sensitive analysis can distinguish between different fields whereas a field-insensitive analysis encapsulates all fields into the parent variable.

2.2.4 Example Static Code Analysis: Taint Checking

Taint checking is a static code analysis technique which tries to identify potentially malicious data-flows within the analyzed source code of a program [ARF$^+$14]. A malicious data-flow exists, if a sensitive information (*taint*) reaches throughout its data-flow through the program a given *sink* (e.g. a method printing the information out to the console) [SAB10]. More generally speaking, taint analysis can be seen as a kind of subclass of information-flow analysis [SM03]. An information-flow is the transfer of information from a variable `x` to a successor variable `y` [DD77].

To illustrate the principle of taint checking, we consider our running example in Figure 1.1. The variable `secret` in Line 12 of the file main.c holds in the scope of our example configurable software system sensitive information. By carrying out a taint analysis on the example software system, we want to examine if a potential information-flow from the variable `secret` to the method `printf` in Line 16 exists.

In scope of this thesis, we implement this static analysis strategy as proof of concept for the combination of the parsing infrastructure TypeChef and the data-flow analysis framework SPL$^{\text{LIFT}}$.

3. Preparing TypeChef for SPL$^{\text{LIFT}}$

The tool TypeChef focuses on parsing and analyzing C source code in the presence of CPP-directives: it is hereby able to parse the entire x86 Linux Kernel, however it only offers intraprocedural data-flow analysis strategies [Lie15]. On the other hand, SPL$^{\text{LIFT}}$ is able to solve data-flow analysis problems in an interprocedural manner, but is limited to configurable software systems expressed within the Colored Integrated Development Environment (CIDE), a research-based extension of the Eclipse IDE [FKF^{+}10, BTR^{+}13]. In CIDE, configurable software systems are written in the Java programming language and variability in the source code is expressed by marking code fragments with different colors. Each color hereby represents a single configuration option [Käs10].

In order to solve interprocedural data-flow analysis problems, SPL$^{\text{LIFT}}$ itself is independent from the used programming language, but requires a variational interprocedural control-flow graph. Currently, this control-flow graph is provided by the underlying tool chain of SPL$^{\text{LIFT}}$, namely CIDE, Eclipse and SOOT, which causes the limitation on CIDE-based configurable software systems. In the scope of this thesis, we replace this tool chain by the parsing infrastructure TypeChef to enable the support of SPL$^{\text{LIFT}}$ for configurable software systems written in the programming language C.

In this chapter we describe the extensions we added to the tool TypeChef to meet SPL$^{\text{LIFT}}$'s requirements of an interprocedural control-flow graph.

3.1 Precise Call Graph

As we have learned in Section 2.2.2, TypeChef already provides an intraprocedural control-flow graph. This control-flow graph can be exploited to become interprocedural by computing the program's call-graph. A call-graph indicates which procedures can call which other procedures, and from which call points within the program [NNH15]. To retrieve a precise call-graph for complete C programs, it is necessary to identify procedures located in other (in C language terms *linked*) source code files and the potential values of function pointers.

3.1.1 Linked-Function Analysis

One of the most important design principle in software engineering is *Separation of Concerns* [Lap07]. The idea behind this principle is to improve the software quality in

```c
int cipher1(int i);

int cipher2(int i);

void ctx_init(struct cipher_ctx *c, int (*f)(int));

int ctx_do(struct cipher_ctx *c, int value);

struct cipher_ctx {
    int (*func)(int);
}
```

Listing 3.1: Header file `minimalLinking.h` of our running example of Figure 1.1

the means of maintainability and reusability by decomposing program functionalities into distinct sections. In the scope of this thesis we focus on configurable software systems written in C. One way to achieve separation of concerns within the programming language C is the usage of header files and linking.

As described in Section 2.1.1 header files are used for collecting externally defined (function-) declarations in a central place which are shared between several source files. Our running example in Figure 1.1 includes the header file `minimalLinking.h` in every single source file. However, different functionalities (e.g. initialization, cipher, execution) are separately implemented in different source code files. During compile-time the corresponding function calls and their destination function definitions are resolved and linked together.

As we aim in this thesis to solve data-flow problems interprocedural, we need to resolve the location of externally defined functions. But first, we have to distinguish between externally defined functions of the C standard library and externally defined functions within the scope of the analyzed configurable software system. The C standard library provides functions for operating system tasks, like input/output processing (e.g. `printf`) or memory management (e.g. `malloc`). The concrete implementation of theses functions is system-specific and is considered as a part of the programming language C. Our approach does not resolve the location of the implementation of functions which are part of the C standard library or compiler-specific language extensions.

Our strategy to detect which functions are linked together is rather simple: in the C programming language function declarations generally are visible to the outside for linking unless they are marked with the language keyword `static`. The TypeChef infrastructure already determines for every successfully parsed C source file which functions are exported for potential linking or are declared externally. However, TypeChef currently does only support the analysis of C source code files one at a time, but typically, large-scale configurable software systems consist of several hundred files (e.g. OpenSSL has a total number of 733 source code files). In order to extract and generate a complete matching of corresponding function calls and declarations, our approach analyzes in a first step each source code file individually and extracts all externally visible function definitions along with its signatures and its presence-conditions. Based on this data, in a second step we map corresponding function calls and the location of their declaration together. This mapping is used later in the variational control-flow graph to determine the file of the target of an external function call and, finally, continue the execution path within the linked file.

3.1.2 Function-Pointer Analysis

The concept of pointers is a fundamental part of the programming language C. A pointer generally references a concrete location in the memory. In C it is possible to reference not only the location of data within the memory, but as well as the location of executable code of a function which should be invoked. These special kind of pointers are called *function pointers*. Like in the previous Section 3.1.1, it is essential for a complete and precise result of interprocedural data-flow analysis, to resolve potential target sites of such function pointer references. Unfortunately, determining the potential value of a pointer reference is not a trivial task and pointer analysis in general is considered as *undecidable* [Lan92]. Over the last decades, academia has proposed a large amount of different approaches to solve this problem [Hin01]. Nevertheless, the silver bullet has not been found yet - each approach provides some tradeoffs between cost and precision.

Our implementation on resolving the destination of function pointers is based on the initial contribution of Ferreira [FKPA15] to the TypeChef infrastructure. Ferreira did choose the **F**low-insensitive **A**lias (FA)-Pointer analysis by Zhang et al. [ZRL96, Zha98] as underlying algorithm. However, like TypeChef, the initial implementation of the function pointer analysis strategy is limited to single C source files only with resolving included file headers but does not take the linking of external files into consideration. Additionally, the existing pointer analysis implementation does not provide a full interprocedural, field-sensitive precision. In this section we will present the concept of the FA-Pointer analysis, and our contribution to overcome the described shortcomings for a precise use with SPL$^{\text{LIFT}}$.

FA-Pointer Analysis

The **F**low-insensitive **A**lias (FA)-Pointer analysis strategy is an inexpensive but precise pointer analysis technique [MRR04]. It is an alias analysis, which means it determines pairs of pointers which *may* reference the same memory location during run-time. Its complexity is almost linear in respect to the size of the analyzed program [ZRL96]. In terms of precision, the FA-Pointer analysis is flow-insensitive, context-insensitive, field-sensitive, and symmetric. The key concept of the analysis is to compute a *Pointer-related Equality (PE) equivalence relation*: pointers which may share the same memory location are hereby partitioned into equivalence classes.

To identify and distinguish pointer variables and referenced memory locations, the FA-pointer analysis uses *object names* as intermediate representation from the source code. In our implementation, a object name consists of two parts: a prefix, which contains

$$\begin{gathered}
(minimalLinking\S GLOBAL\$cipher1,\ True),\\
(minimalLinking\S GLOBAL\$cipher2,\ True),\\
(main\S main\$\&cipher1,\ A),\ (main\S main\$\&cipher2,\ !A),\\
(main\S main\$*c,\ True),\ (main\S main\$c,\ True),\\
(init\S ctx\_init\$c \rightarrow func,\ True),\ (init\S ctx\_init\$f,\ True)\\
(init\S ctx\_init\$*c,\ True),\ (init\S ctx\_init\$c,\ True),\\
(do\S ctx\_do\$*c,\ True),\ (do\S ctx\_do\$c,\ True),\\
(do\S ctx\_do\$c \rightarrow func,\ True)
\end{gathered}$$

Figure 3.1: Every object name of the running example in Figure 1.1

the filename of variable in the source code file and the scope of the variable within the file. This prefix is followed by the syntactical name of the variable in the source code and, if existing, with its surrounding pointer operators, like field access- or pointer (de-) reference operators. In order to keep the analysis strategy variability-aware, every single generated object-name is stored along with its presence-condition. To illustrate this, we recall our running example in Figure 1.1. In our running example, we have a struct which has a function pointer as field member. This function pointer gets, depending on the configuration, referenced by two different functions: `cipher1` and `cipher2`. Figure 3.1 shows the resulting set of object names and their corresponding presence-conditions for our running example.

Extracting all pointer-related object names from the source code is a prerequisite for the actual algorithm. The algorithm can be divided into two phases: in the first phase the initial equivalence classes are created. In the second phase each and every pointer-related assignment statement gets examined and corresponding equivalence classes are merged together.

The algorithm for constructing the pointer equivalence relation is shown in Algorithm 3.1. It requires the following procedures to compute the PE equivalence relation:

- `InitEquivClass(`o`)`: initialises a new equivalence class for a given object name o
- `Find(`o`)`: retrieves the corresponding equivalence class for a given object name o
- `Union(`$e1,\ e2$`)`: merges two equivalence classes $e1,\ e2$ into a single one equivalence class e

In the first phase from Line 1 to 13 the initialisation steps are performed: for every object name o a corresponding equivalence class is created. Theses individual equivalence classes consist of two parts: a set of related object names (during the initialisation phase each equivalence class has the size of one) and a set of prefixes. The prefix set is required for maintaining the relation to object names in other classes. It is generated in Line 5 to 13

Algorithm 3.1: Algorithm for Computing the PE - Relation, reproduced from [Zha98]

```
Input  : A set $B$ with all object names of the program
Output: Equivalence Relation of all object names potentially sharing the same
        memory location

CalculatePERelation()
    foreach $o \in B$ do
        InitEquivClass($o$);
        Prefix(Find($o$)) = $\emptyset$;
    end
    foreach $o \in B$ do
        if $o == \&o_1$ then
            Add(*, $o_1$) to Prefix(Find($o$));
        else if $o == *o_1$ then
            Add(*, $o$) to Prefix(Find($o_1$));
        else if $o == o_1.field$ then
            Add($field$, $o$) to Prefix(Find($o_1$));
        end
    end
    foreach $x = y$ in the program do
        if Find($x$) $\neq$ Find($y$) then
            Merge($x$, $y$);
        end
    end

Input  : Two Equivalence Classes $e1$ and $e2$
Output: One merged Equivalence Class $e$

Merge()
    $e$ = Union($e1$, $e2$);
    $newPrefix$ = Prefix($e1$);
    foreach ($a$, $o$) $\in$ Prefix($e2$) do
        if ($a_1$, $o_1$) $\exists$ Prefix($e1$) && ($a == a_1$) then
            if Find($o$) $\neq$ Find($o_1$) then
                Merge($o$, $o_1$);
            end
        else
            $newPrefix = newPrefix \cup \{(a, o)\}$;
        end
    end
    Prefix($e$) = $newPrefix$;
```

PE Relation Set	Prefix Set
$\{(main\S main\$c,\ True)\}$	$\{(*, main\S main\$*c,\ True)\}$
$\{(main\S main\$*c,\ True)\}$	$\{\ \}$
$\{(init\S ctx\_init\$c,\ True)\}$	$\{(init\S ctx\_init\$*c,\ True)\}$
$\{(init\S ctx\_init\$*c,\ True)\}$	$\{(func, init\S init\$c \rightarrow func,\ True)\}$
$\{(init\S ctx\_init\$c \rightarrow func,\ True)\}$	$\{\ \}$
$\{(do\S ctx\_do\$c,\ True)\}$	$\{(*, do\S ctx\_do\$*c,\ True)\}$
$\{(do\S ctx\_do\$*c,\ True)\}$	$\{(func, do\S ctx\_do\$c \rightarrow func,\ True)\}$
$\{(do\S ctx\_do\$c \rightarrow func,\ True)\}$	$\{\ \}$

Table 3.1: Selected initial equivalence relations set and their corresponding prefix set for Figure 1.1

by the algorithm. Consequently, every field member access or pointer reference operation is added to the prefix set of the pointer's original declaration object name $o1$ along with its presence-condition.

In Table 3.1 we see the initial equivalence relation and the corresponding prefix set for chosen example pointers from our running example in Figure 1.1. Intuitively, we see that each individual pair of the generated prefix sets $(a, o) \in prefix(e)$, and $o \in e1$, represents a connection, or in graph terms an *edge*, from e to $e1$ labeled with the pointer accessor a.

In the second step from Line 15 to 18, the program computes all pointer related assignment statements within the program. Pointer related assignments can be classic assignment statements, like `x = y`, but also memory allocations, i.e. `memcopy`, or function parameters and function return values. Logically, function parameters and their corresponding call parameters are matched together as an assignment[1].

As a recall, in a PE equivalence relation, object names in the same equivalence class may share the same memory location. It is straightforward in the concept of computing PE equivalence relation, to merge corresponding equivalence classes of object names which are part of the same assignment within the analysed program.

[1] One may notice, that the pointer analysis strategy requires at that point as well the location of externally linked functions for a complete result. Unfortunately, as the FA-Pointer analysis is by design context- and flow-insensitive. As a consequence, resolving all linked functions upfront, and consequently recursively for every linked file of a linked file, causes in practice a high computation time. This is caused by the fact, that real-life software libraries, like OpenSSL, have different execution paths for individual tasks (e.g. the execution path for AES-encryption differs to the execution path of certificate validation), which individually does not visit every single procedure within the library, but as the analysis strategy is context- and flow-insensitive, these functions would be computed as well. However, by exploiting the fact, that the CFG for SPL$^{\text{LIFT}}$ is flow-sensitive, externally linked functions are only computed, if and only if they are visited when traversing the execution path of the CFG. We observed, that this lazy calculation technique reduces the calculation costs without any loss in precision, as unreferenced files are not loaded.

In our running example the following equivalence classes are created solely based on assignments without considering the prefix set:

$$
\begin{gathered}
\{(main\S main\$\&cipher1,\ A),\ (main\S main\$\&cipher2,\ !A),\\
(init\S ctx\_init\$c \rightarrow func,\ True),\ (init\S ctx\_init\$f,\ True)\}\\
\{(ctx\_init\S ctx\_init\$c,\ True),\ (do\S ctx\_do\$c,\ True),\\
(main\S main\$c,\ True))\}\\
\{(do\S ctx\_do\$c \rightarrow func,\ True)\}
\end{gathered}
$$

As we can see, the object names of the function pointers `&cipher1` and `&cipher2` were not merged into the same equivalence class together with their referenced struct field `c->func` in the procedure `ctx_do`.

However, as explained before, the prefix set of each equivalence class maintains relation between its object names and those in other classes. By computing this additional relation during a merge, such indirect assignments are resolved as well. Every time, two merged equivalence classes share the same accessor, e.g. `*` or a field reference, in their prefix set, the corresponding accessor's equivalence classes are recursively merged as well. Let's examine the assignment of the function call parameter `c` and the function parameter `*c` of the method `ctx_do` as an explanatory example[2]:

In the first step, the algorithm would start by uniting the equivalence classes:

$$
\begin{gathered}
\{(do\S ctx\_do\$c,\ True)\}\\
\{(main\S main\$c,\ True)\}
\end{gathered}
$$

As we can read from Table 3.1, both equivalence classes have a non-empty prefix set and share the same accessor `*`. Consequently, both corresponding equivalence classes are merged as well, however their prefix sets have no matching common accessor, which leads to the following equivalence class and its prefix set:

$$
\begin{gathered}
\{(main\S main\$*c,\ True),\ (do\S ctx\_do\$*c,\ True)\}\\
\{(func, do\S ctx\_do\$c \rightarrow func,\ True)\}
\end{gathered}
$$

Next, we assume the assignment of the function call parameter `c` and the function parameter `*c` of the method `ctx_init` are computed by the algorithm. Like before, both equivalence classes share the same prefix accessor and consequently the referred equivalence classes are merged as well:

$$
\begin{gathered}
\{(main\S main\$*c,\ True),\ (do\S ctx\_do\$*c,\ True),\ (init\S ctx\_init\$*c,\ True)\}\\
\{(func, do\S ctx\_do\$c \rightarrow func,\ True),\ (func, init\S ctx\_init\$c \rightarrow func,\ True)\}
\end{gathered}
$$

[2] Syntactically pointer related call-to parameter assignments, as for example the call `foo(x)` to the function `int foo(int *y)`, are treated like `int *y; y = x;`

We can see in the resulting prefix set, both merged equivalence classes shared the same prefix accessor `func`. As a result both equivalence classes are merged as well:

$$\{(main\S main\$\&cipher1,\ A),\ (main\S main\$\&cipher2,\ !A),\ (init\S ctx\_init\$f,\ True),$$
$$(init\S ctx\_init\$c \rightarrow func,\ True),\ (do\S ctx\_do\$c \rightarrow func,\ True)\}$$

Because of the fact, that the function pointers `cipher1` and `cipher2` are assigned to `f` and `f` gets assigned to to the struct field `c->func` in the procedure `ctx_init`, both struct fields are now correctly in the same equivalence class.

In the original contribution of Ferreira, the matching process for struct field members did not produce the expected result as defined in the algorithm. We reimplemented from scratch the prefix set generation and merge mechanism for a more precise result. Additionally, for C structs and unions, who are initialised by so called *curly initializers* (i.e. `struct s = { field1, field2 }`), no object names were generated. Finally, the pointer analysis strategy was limited to single C source files only. As mentioned before, our addition to the mechanism resolves externally linked functions on-the-fly when traversing the variational interprocedural control-flow graph in SPL$^{\text{LIFT}}$.

3.2 Variability-Aware Interprocedural Control-Flow Graph

As outlined in Section 2.2.2, Liebig et al. [LvRK^{+}13] already implemented a variational intraprocedural control-flow graph within TYPECHEF. The only requirement of SPL$^{\text{LIFT}}$ is an interprocedural control-flow graph for the target programming language. Using the previously described additions to the tool TYPECHEF, we are able to construct a variational interprocedural control-flow graph, which is compatible to the requirements of SPL$^{\text{LIFT}}$[3]. In this section, we will present this CFG and describe which transparent modifications to the underlying abstract syntax tree we did apply for compatibility reasons.

3.2.1 Implementation Details

SPL$^{\text{LIFT}}$ is written in Java and requires the interface InterproceduralCFG<N,M> (cf. Listing 3.2) to be implemented as connector to the underlying interprocedural control-flow graph (ICFG). The interface itself is generic, its parameters, `N` and `M`, represent nodes and methods of the respective ICFG and AST. Due to the fact, that TYPECHEF is mostly written in the programming language Scala[4], the ICFG interface and the implementation of SPL$^{\text{LIFT}}$ are on paper fully compatible to the existing TYPECHEF source code base.

As a consequence of the fact, that our implementation of the interprocedural CFG is based on the existing intraprocedural CFG, we did choose the type `V[CFGStmt]` for the generic node parameter `N` and the type `V[CFGFDef]` for the parameter `M`, as these are the

[3] The CFG provided by TYPECHEF differs from the one provided by SOOT and CIDE in the matter of variability encoding. As a consequence we did alter SPL$^{\text{LIFT}}$'s lifting mechanism.

[4] Scala runs on the Java VM and therefore Scala source code is in theory fully compatible to Java source code and, except some high order Scala language features, vice versa.

```java
public interface InterproceduralCFG<N,M>  {

        public Set<N> allNonCallStartNodes();

        public FeatureExpr getCondition(N n)

        public Set<M> getCalleesOfCallAt(N n);

        public Set<N> getCallersOf(M m);

        public Set<N> getCallsFromWithin(M m);

        public M getMethodOf(N n);

        public Set<N> getStartPointsOf(M m);

        public List<N> getReturnSitesOfCallAt(N n);

        public List<N> getSuccsOf(N n);

        public List<N> getPredsOf(N n);

        public boolean isCallStmt(N stmt);

        public boolean isBranchTarget(N stmt, N succ);

        public boolean isExitStmt(N stmt);

        public boolean isFallThroughSuccessor(N stmt, N succ);

        public boolean isStartPoint(N stmt);
```

Listing 3.2: Interface for the implementation of the interprocedural control-flow graph for SPL$^{\text{LIFT}}$

return types of the the underlying successor function of the intraprodcedural CFG (cf. Section 2.2.2).

Following, we present the key procedures[5] of the ICFG interface and their implementation details:

- getSuccsOf: $V[CFGStmt] \rightarrow List[V[CFGStmt]]$: This function is the equivalent to the classic successor function of CFGs. It returns all intraprocedural successor nodes of a given CFG node n and its individual control-flow presence-condition.
- getCondition: $V[CFGStmt] \rightarrow \mathcal{PC}$: Returns the presence-condition of the current node n. Note: in our implementation, this function returns the current incoming control-flow condition encoded in the CFG node data-type[6].

[5] Not every function defined in the interface is used by the actual implementation of SPL$^{\text{LIFT}}$. We implemented every function of the interface, however in the scope of this thesis, we focus our explanation on used functions only.

[6] We will comment on this special behavior in Section 5.7.

- isCallStmt: $V[\mathit{CFGStmt}] \rightarrow \mathit{Boolean}$: Determines wether the current node is a outgoing call statement to another procedure or not.

- getCalleesOfCallAt: $V[\mathit{CFGStmt}] \rightarrow \mathit{Set}[V[\mathit{CFGFDef}]]$: Returns all possible and valid target function definitions of a function call. Internally, this method examines the in Section 3.1.2 presented PE equality relation to resolve the potential targets of function pointers. Further, it resolves externally linked function definitions. In case the target of a function call is defined in an external file, the corresponding abstract syntax tree is loaded and transparently included into the control-flow graph. Additionally, the pointer equivalence relation classes are newly computed in respect to the added file. The result set contains all potential destination function definition, such as different function pointer destination for different configurations. Note: in case of a recursive function call within the function, an empty set is returned.

- isStartPoint: $V[\mathit{CFGStmt}] \rightarrow \mathit{Boolean}$: Determines wether the current node is the entry statement of its surrounding procedure.

- getStartPointsOf: $V[\mathit{CFGFDef}] \rightarrow \mathit{Set}[V[\mathit{CFGStmt}]]$: Retrieves all statements of a function, which are executed as entry points.

- isExitStmt: $V[\mathit{CFGStmt}] \rightarrow \mathit{Boolean}$: Determines wether the current node is an exit statement of its surrounding procedure.

- getReturnSitesOfCallAt: $V[\mathit{CFGStmt}] \rightarrow \mathit{List}[V[\mathit{CFGStmt}]]$: Retrieves all statements to which a function call could return to. In our implementation, we compute at this point internally all intraprocedural successor statements of the call statements.

Additionally, our TypeChef specific implementation of this ICFG interface provides a few additional methods. We introduced the following methods for convenience reasons only, as they provide static information, which are gathered during parsing-time:

- getEntryFunctions: $\mathit{Unit} \rightarrow \mathit{List}[V[\mathit{CFGFDef}]]$: Returns a list of all potential starting methods (such as `main`) of the currently analyzed case study.

- getTS: $V[\mathit{CFGStmt}] \rightarrow \mathit{CTypeSystem}$: Retrieves the variational type-system of TypeChef. With the help of the type-system we are able for example to determine the field members of a struct.

- getTUnit: $V[\mathit{CFGStmt}] \rightarrow \mathit{AST}$: Retrieves the AST of given node. We use this function to identify globally defined variables, which are logically not visited by the ICFG.

At last, a side note on a very important implementation detail of the data-types `CFGStmt` and `CFGFDef`. Internally the TypeChef infrastructure implements different kinds of AST nodes as Scala *case classes*. This brings the advantage of using pattern matching to distinguish between different kinds of AST nodes. Another language feature of case classes is *structural equality*. Per default, the Scala compiler generates an `equals` method for case classes which compares two instances of an case class by the properties of their field values instead of by classic object references in Java. Hence, a comparison of two instances of the case class representing a single `return` statement would return true as the distinct source code position is not a field property of the implemented case class. Despite of the benefits of this very practical language feature of Scala, this fact lead to unpredictable behavior in the underlying solver of SPL$^{\text{LIFT}}$, HEROS. HEROS is written in Java and stores its intermediate and final computation results in data-structures using hash tables. Additionally, HEROS is partly multi-threaded, so consequently false and non-deterministic results were computed by HEROS as different AST nodes have been identified as equal nodes despite being unequal in respect to their source code position. Especially, the occurrence of expanded macros and `return` statements in the AST caused false behavior, as they occur commonly more than once within the source code.

To overcome this fundamental shortcoming, we implemented the data-types `CFGStmt` and `CFGFDef` as a wrapper case class for AST nodes with a custom `equals` method which still implements structural equality, but additionally compares the location within the original source code as well. The most obvious approach however, to implement the described custom `equals` method directly on AST nodes is not applicable, since the TYPECHEF infrastructure relies on the default equality comparison mechanism of case classes for custom type definitions in C, such as `typedef` specifier, during the parsing process. Finally, the TYPECHEF infrastructure sets for expanded macros the position of the macro definition rather than the position of the macro usage within the source code for corresponding AST nodes. Our proposed solution would fail at this point, since we are again unable to distinguish different AST nodes for such statements since their positions are always the same. To avoid this faulty behavior, the TYPECHEF-infrastructure must be started with the flag `adjustLines` to enforce position information in AST nodes matching to their real occurrence within the source code.

3.2.2 Modifications to the AST

The C programming language and the resulting AST for C source code files are more expressive than the analysis framework SPL$^{\text{LIFT}}$ can compute. This is caused by the fact, that the used framework CIDE in the original tool-chain of SPL$^{\text{LIFT}}$ enforces disciplined feature annotations which cover a complete statement node in the control-flow graph [Käs10]. However, the use of CPP-directives is not bound to any language expression, which leads to *undisciplined annotations* as shown in Listing 3.3. This is not only a theoretical problem, but it is existing in real-life software [LKA11]. Fortunately, theses limitations can be resolved without losing precision by rewriting incompatible source code fragments into compatible source code fragments.

```
int foo(int x, int y) {
    int bar = x +
#ifdef A
    y
#else
    x
#endif
;
    return bar;
}
```

Listing 3.3: Example for undisciplined CPP-annotations

```
int foo(int i, int j) {
#ifdef A
    int bar = x + y;
#else
    int bar = x + x;
#endif

    return bar;
}
```

Listing 3.4: Duplicated code for undisciplined CPP-annotations

To transform undisciplined CPP annotations into disciplined annotations we use a brute force approach of *code duplication*. We hereby scan the original AST for presence-conditions nested in statement nodes of the control-flow graph. For every found statement, we replace the original statement node with every possible and valid combination according to the feature model of this statement in the AST and annotate the entire duplicated code fragment with its corresponding presence-condition. An example result for this process can be seen in Listing 3.4.

Additional rewrite operations are necessary on function calls, which are not supported by the underlying solving framework HEROS. Incompatible function calls are nested function calls (e.g. `foo(bar(x))`) and function calls located in exit statements (such as `return foo(x)`). For example, the statement `return foo(bar(x))` is replaced within the AST by the following fully equivalent list of statements:

```
ReturnTypeOf(bar) tmp = bar(x);
ReturnTypeOf(foo) tmp2 = foo(tmp);
return tmp2;
```

Rewriting the intermediate source code representation syntactically while preserving its semantic behaviour is a well-known technique in academia and in practice [Lat08, NMRW02].

4. Taint Checking as IFDS-Problem

In this chapter, we will present the basic concept of the IFDS-framework and formulate an assignment-based information-flow analysis within this framework for software written in the programming language C. This IFDS based data-flow analysis problem we will use later to taint check all variants of the implementation of the AES cipher in two configurable cryptography libraries.

4.1 IFDS-Framework

The IFDS-*framework* by Reps, Horwitz and Sagiv [RHS95] is a framework for formulating and solving *interprocedural, finite, distributive, subset* (IFDS) data-flow problems. These problems are solved in a flow-sensitive, fully context-sensitive manner by the framework's algorithm. Data-flow analysis expressed as a IFDS-problem must hold the following conditions [RHS95]:

- A finite set D of data-flow facts. Data-flow facts hold information regarding the current analysis objective (i.e. a variable `x` has been tainted).
- A set of flow functions: $F \subseteq 2^D \rightarrow 2^D$. A flow function transfers a data-flow fact from a source statement to its successor statement.
- A distributive merge operator $\sqcap$ which holds:
  $\forall\, a, b \in D,\ f \in F : f(a) \sqcap f(b) = f(a \sqcap b)$

If a data-flow analysis problem formulated in this framework fulfils the described conditions, Reps et al. have shown that these problems can be reduced to a pure graph-reachability problem. Problems which meet these conditions are for example classic "*gen/kill*" problems, like e.g. reaching definitions, or live variables, but are not limited on such problems.

The algorithm works by generating a so called "*exploded supergraph*" based on the program's interprocedural CFG. Similar to the previously described control-flow graph in Section 2.2.2, nodes of the supergraph denotes to statements of the program and consecutive statements are connected together via directed edges. In distinction to the classic CFG edges, which only represent the execution order between statements, the supergraph of the IFDS-framework knows four different types of edges:

normal edge: a intraprocedural edge modelling the flow from a statement to its successor statements. Its behaviour is mostly similar to classic CFG edges including branching, but with the sole difference, this edge is not applicable for call- nor return statements.

call edge: a interprocedural edge modelling the flow from a call statement to the corresponding entry statement at callee site.

return edge: a interprocedural edge pointing from the return statement of the callee site to the return-site of the call statement. Note: as normal edges are not applicable for call statements, this edge points to the intraprocedural successor statements of the call statement.

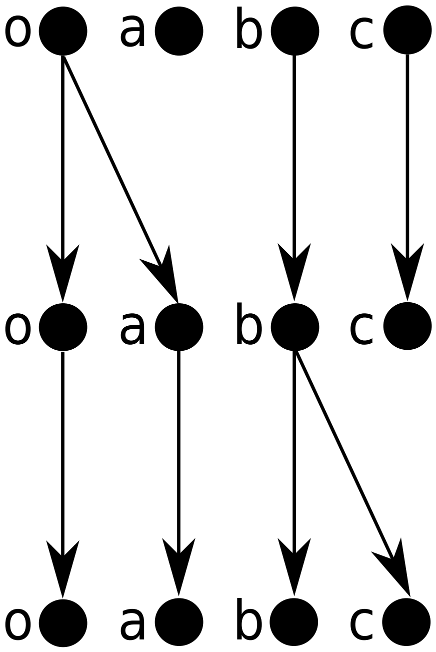
call-to-return edge: a intraprocedural edge modelling the flow from a call statement to a given return-site. As call statements are excluded from normal edges, this edge is used to preserve flow facts which are not passed on the call edge.

These edges denote the flow of data-flow facts in the exploded supergraph and are called flow functions according to the edge type (e.g. for a normal edge it is called normal flow function). So, from a users point of view, to formulate a data-flow analysis problem within the IFDS-framework, only these flow functions must be defined and implemented for each type of edge. These four flow functions compute in the scope of the analysis context which data-flow facts are generated, propagated or killed along the edge they are modeling. In textbook terms for data-flow analysis, a flow function within the IFDS framework is the equivalent to a transfer function [NNH15]. This is the very basic concept of the IFDS framework: based on the exploded supergraph and transfer rules for individual edges expressed as flow functions, one is able to determine if a data-flow fact from a source node within the graph may reach a target node or not. At worst case, the algorithm has a complexity of $O(ED^3)$, where E denotes the total amount of control-flow edges of the analyzed source code and D the number of data-flow facts.

Flow Functions

Figure 4.1 illustrates a composition of flow functions and the corresponding source code as an explanatory example. To explain the typical *gen-* and *kill-* functions of classic data-flow analysis problems [NNH15] as flow functions, we use a simplified information-flow analysis (cf. Section 2.2.4).

In the graphical representation of flow functions, nodes in a row represent the state before and after a single statement: In our explanatory example, the top row represents the state of data-flow facts before Line 2 of the corresponding source code snippet. Arrows between two rows of nodes map, which data-flow facts are generated, propagated or killed at a certain statement. So consequently, the last row of our example is the final state after the statement in Line 3 has been examined, whereas the middle row describes the state between Line 2 and 3.



```
1 void foo(int b, int c) {
2     int a = 0;
3     c = b;
4     return;
5 }
```

Figure 4.1: Composition of different local and non-locally separable flow functions adopted from [RHS95, BTR+13] and the corresponding source code snippet

First, the IFDS-framework knows a special data-flow fact O, a fact which always holds, as to be seen at the very left of the example. This special fact is required to generate new data-flow facts unconditionally (or as often referred in literature as *locally separable* flow function). For example, a data-flow analysis can generate in this manner the data-flow fact of a newly introduced variable definition - in our case the introduction of variable `a` in Line 2 by connecting the fact O with fact `a`, in the upper case.

To express the process of propagating or killing a certain data-flow fact, the linking arrow between the origin node and its successor node is either kept or removed. In Figure 4.1 this is represented at the very right flow fact `c`: at the top half, the fact is propagated with an edge from `c` to `c`, where as in the bottom half the fact `c` is killed by introducing a new fact `c` and consequently the old fact is no longer reachable in the graph. The introduction of the fact `c` is the alternative way of generating a new data-flow fact. The previously described method does not include the values of already existing facts. However, many data-flow analysis problems require these facts, e.g. the taint analysis which we will present in the scope of this thesis. Such flow functions are called *non-locally separable*. Our explanatory example models a variable reassignment: `c = b`. The previous value of `c` is no longer valid after the reassignment and it is not necessary to propagate the fact along the graph anymore. However, the newly introduced data-flow fact value of `c` depends on the value of `b`, which is represented by the edge from `b` to `c`.

4.2 Taint Checking as IFDS Problem

As outlined in Section 2.2.4, taint checking is a static code analysis technique which aims to identify potentially malicious data-flows within the analyzed source code of a program [ARF+14]. It hereby traces the data-flow of an interesting input value (= *source*) (e.g. the secret key for AES-encryption) to a potential *sink* (e.g. a instruction which prints the secret value out to the console). But the definition of potential sources and malicious sinks depends on the concrete application scenario. In order to keep our taint checking approach applicable to all different kinds of application scenarios, we formulate a more general, assignment-based *information-flow* analysis problem within the IFDS-framework as underlying basis for taint checking. Our information-flow analysis problem traces every potential data-flow from its origin to its end. To finally perform taint checking, we simply have to filter the resulting set of all detected information-flows for interesting source- and/or sink-values.

In this section we will formulate first the general information-flow analysis problem within the IFDS-framework[1]. In a second step, we will apply this analysis strategy on a product of our running example configurable software system. We reuse this general information-flow analysis in Section 6.4 to taint check the used *private key* in the implementation of the AES cipher in the cryptography libraries MBEDTLS and OPENSSL.

4.2.1 Information-Flow Analysis

The very basic concept of our assignment-based information-flow analysis shares commonalities with the classic data-flow analysis *reaching definitions*. Reaching definitions is a static code analysis technique which determines which variable definitions may reach a certain target variable in the analysed source code fragment [NNH15]. Our idea is not to trace which variable definition may reach a certain target instruction, but rather to trace which *information* may reach a certain target instruction.

```
 1  int flow() {
 2    int a, b, c;
 3
 4    a = 5;
 5    b = a;
 6    a = 2;
 7    c = b;
 8
 9    return c;
10  }
```

Listing 4.1: Minimal code example for information-flow.

To illustrate the idea of information-flow analysis consider the minimal example illustrated in Listing 4.1. There are three variables a, b, and c. The first variable a gets an assignment by the value of 5. During the execution of the example source code fragment, the variable b gets an assignment from a. Variable a is reassigned in following statement and lastly, b is assigned to c. A classic reaching definition analysis would identify, that the definition of variable a would reach b, and b reaches variable c. Nevertheless, it would not detect the flow of information from a to c. Our analysis aims to detect such flows of information. Fortunately, we can extend the concept of a reaching definitions analysis problem to trace the flow of information within the source code of the analyzed program.

Consider again the code example in Listing 4.1: a classic textbook reaching definition analysis would model as data-flow fact the origin of a single variable definition. In our case for variable a the data-flow fact would contain the assignment to a in Line 4. The same analysis strategy would kill this fact at Line 6 of the example as for variable a a new definition is introduced and consequently the old value of a is no longer available.

The idea of our approach is to encode in every newly introduced definition data-flow fact every matching incoming definition in order to propagate this information further

[1] Please remind, the formulated problem itself will be completely unaware of the challenges of configurable software systems, as SPL$^{\text{LIFT}}$ is lifting IFDS problems transparently for such systems [BTR^{+}13].

even if the original definition is no longer valid. Applied on the code example of Listing 4.1, our "reaching information" analysis strategy would encode the definition of variable `a` from Line 4 into the definition of `b`. This way, we are able to trace every potential flow of information from its origin to its very last sink even if its original definition has exceeded its life-time.

As previously described, data-flow analysis problem within the IFDS framework are specified by a number of flow functions F and a finite set of data-flow facts D (cf. Section 4.1). For our information-flow problem, we define the following set of *data-flow facts* D which follow the previously described principle:

ZERO: The special default data-flow fact O of the IFDS-framework which must always hold. Consequently, this data-flow fact is transferred at any flow function.

SOURCE: Flow facts of this type hold information about the name, the origin statement location and the scope of a variable definition. This data-flow fact is *field-sensitive* for the C language data types of structs and unions. For data-flow facts of the type SOURCE we distinguish between two cases:

SOURCEDEFINITION: This data-flow fact is generated for every definition of an variable. It contains the previously described data and is valid until a new definition of the corresponding variable is introduced replacing the previous one. Formally we define this fact as `Src`$[x, s]$ where x denotes to the variable name as it does occur in the ordinary source code and s denotes the visibility scope of the variable's name. The visibility scope is hereby a numeric range starting with the value of 0 for globally visible variables and increases by 1 for every added visibility scope (i.e. a variable visible in a classic method scope would have the scope value of 1). For simplicity reasons, when not explicitly referenced as `Src`$[x.f]$, struct fields are expressed by the expression `Src`$[x]$, where x is an abbreviation for $s.f$.

SOURCEDEFINITIONOF: This data-flow fact is equally related to the previous fact SOURCEDEFINITON, but additionally contains information of a previous source fact which reaches the definition of the current variable. Its formal representation `SrcOf`$[x, s, o]$ is similar to the previously described SOURCEDEFINITON: x and s denote to the variable name and respectively to the visibility scope. The parameter o references the incoming reaching SOURCE data-flow fact. For example, at the assignment of variable `a` to `b` in Listing 4.1 the SOURCE fact of `a` is encoded into as parameter o in the generated SOURCEDEFINITIONOF fact of `b`.

SINK: This data-flow fact holds information about an detected sink of an variable at a certain source code statement. Note: these facts are only generated to report potential sinks at a certain statement. For further computation at any flow function, facts of the type SINK are not evaluated nor transferred.

Notation	Statement	Example
$x \mapsfrom \texttt{def}$	Definition	`int x;`
$x \mapsfrom y$	Assignment	`x = y;`
$x \mapsfrom y.f$	Assignment with Field Read	`x = y.f;`
$x.f \mapsfrom y$	Assignment with Field Write	`x.f = y;`
x	Expression	`x < 0;`

$m_{Call}([cp, fp])$	Method Invocation	`foo(cp);`
		`foo(int fp){}`

$x \mapsfrom m_{Ret}(r)$	Method Return	`x = foo(cp);`
		`int foo(fp) {return r;}`

Table 4.1: Examined C statement types by the information-flow analysis

- **SinkToAssignment:** This flow fact is generated for every reach of a SOURCE fact to the right-hand side of an assignment.
- **SinkToUse:** This fact is generated for every reach of a SOURCE fact to source code statement which is not an assignment (i.e. sink because of use as a parameter in the `printf` instruction).

The notation of these data-flow facts is equal to the notation commonly used for taint-checking: the fact *Source* is the origin value of a certain variable, while the fact *Sink* represents the reach of a given source fact at a certain statement. Note: as our information-flow analysis is more general than a taint analysis, we define every reach of a SOURCE fact as a sink. For taint checking, the resulting set of sink facts must be filtered for interesting sinks in the scope of the application scenario as mentioned before.

Next, we formulate the flow functions to compute the previously introduced data-flow facts at the different types of edges of the exploded supergraph within the IFDS-framework. The formulated flow functions will evaluate all of in Table 4.1 referenced instructions of the C programming language. In general, a flow function of our information-flow analysis has the following type:

$$\llbracket s \rrbracket_I \colon 2^S \times 2^D \mapsto 2^D$$

S is hereby the set of all statements of the analyzed configurable software system (cf. Table 4.1). Since our analysis strategy uses *non-locally separable* flow functions, we need to decompose the flow functions into their individual effect on each single incoming data-flow fact. Therefore, we define a flow function with the argument I, which denotes a set in 2^D consisting of all incoming data-flow facts $\{d_1, ..., d_n\}$ for a statement s as follows:

$$\llbracket s \rrbracket(I) := \llbracket s \rrbracket(O) \cup \bigcup_{d \in I} \llbracket s \rrbracket(d)$$

Following, the decomposed flow functions of our analysis strategy, afterwards we will apply the presented concept on our running example. For simplicity reasons, we abbreviate the data-flow fact `SourceDefinition` to `Src` and `SourceDefinitionOf` to `SrcOf` in the different flow function equations.

4.2.1.1 Initial Data-Flow Facts

Flow functions of the IFDS-framework generate, propagate and kill individual data-flow facts along edges of the CFG. However, by traversing the CFG our analysis approach may not visit every potential source of information. This is caused by the fact, that nodes outside of procedures are not part of the CFG. In the C programming language such nodes typically represent type-definitions, forward function declarations, etc. but also global variables. Additionally, we must point the analysis framework to the start of the execution path of the program. This is mostly likely the `main` method of a program, but may also be any other entry point as defined in a unique application scenario.

In order to resolve the mentioned conditions, the IFDS-framework knows an initial value set, called *initial seeds*. These initial seeds are a list of tuples consisting of a number of starting points of the execution path within the analyzed program along with a set of precomputed data-flow facts.

In our information-flow analysis problem we report in this initial seeds set the analysis' starting points (in our case if not defined otherwise the function `main`) and a set of data-flow facts containing the corresponding `Src` facts of already assigned global variables. In order to collect all global variables, we use a identity data-flow problem for IFDS. This data-flow problem simply traverses the execution path reported by the interprodecudral CFG without executing any flow function. Afterwards all visited files are reported, we extract from each visited file any globally assigned variable as a new initial `SourceDefiniton` data-flow fact.

4.2.1.2 Normal Flow Function

A normal flow function models the intraprocedural flow from a source statement to its successor statement within a single procedure. Figure 4.2 illustrates the various decomposed cases which apply for a normal flow function in our information-flow detection strategy.

For the normal flow we distinguish between various combinations of the current statement type and the incoming data-flow fact:

- **Definition and (Re-) Declaration:** in case of definition and (re-) declaration of variables, we rely on the default data-flow fact of the IFDS-framework O as formulated in Equation 4.1 and 4.2. For every newly introduced or (re-) assigned variable in the analyzed program, this default fact generates the corresponding data-flow fact type `Source`. This way we are able to trace back the origin and the potential initial value

$$\llbracket x \mapsfrom \texttt{def} \rrbracket(O) = \{\texttt{Src[}x\texttt{]}\} \tag{4.1}$$

$$\llbracket x \mapsfrom y \rrbracket(O) = \{\texttt{Src[}x\texttt{]}\} \tag{4.2}$$

$$\llbracket x \mapsfrom y \rrbracket(\texttt{Src[}v\texttt{]}) = \begin{cases} \{\texttt{Src[}v\texttt{]},\ \texttt{SrcOf[}x, \texttt{Src[}v\texttt{]]}, \\ \ \texttt{Sink[}x, \texttt{Src[}v\texttt{]]}\} & \text{if } x \neq v \wedge y = v \\ \{\texttt{SrcOf[}x, \texttt{Src[}v\texttt{]]},\ \texttt{Sink[}x, \texttt{Src[}v\texttt{]]}\} & \text{if } x = v \wedge y = v \\ \emptyset & \text{if } x = v \wedge y \neq v \\ \{\texttt{Src[}v\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.3}$$

$$\llbracket x \mapsfrom y \rrbracket(\texttt{SrcOf[}v, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}v, o\texttt{]},\ \texttt{SrcOf[}x, o\texttt{]}, \\ \ \texttt{Sink[}x, \texttt{SrcOf[}v, o\texttt{]]}\} & \text{if } x \neq v \wedge y = v \\ \{\texttt{SrcOf[}x, o\texttt{]},\ \texttt{Sink[}x, \texttt{SrcOf[}v, o\texttt{]]}\} & \text{if } x = v \wedge y = v \\ \emptyset & \text{if } x = v \wedge y \neq v \\ \{\texttt{SrcOf[}v, o\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.4}$$

$$\llbracket x \mapsfrom y \rrbracket(\texttt{Src[}v.f\texttt{]}) = \begin{cases} \{\texttt{Src[}v.f\texttt{]},\ \texttt{Sink[}x.f, \texttt{Src[}v.f\texttt{]]} \\ \ \texttt{SrcOf[}x.f, \texttt{Src[}v.f\texttt{]]}\} & \text{if } x \neq v \wedge y = v \\ \{\texttt{SrcOf[}x.f, \texttt{Src[}v.f\texttt{]]}, \\ \ \texttt{Sink[}x.f, \texttt{Src[}v.f\texttt{]]}\} & \text{if } x = v \wedge y = v \\ \emptyset & \text{if } x = v \wedge y \neq v \\ \{\texttt{Src[}v.f\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.5}$$

$$\llbracket x \mapsfrom y \rrbracket(\texttt{SrcOf[}v.f, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}v.f, o\texttt{]},\ \texttt{SrcOf[}x.f, o\texttt{]}, \\ \ \texttt{Sink[}x.f, \texttt{SrcOf[}v.f, o\texttt{]]}\} & \text{if } x \neq v \wedge y = v \\ \{\texttt{SrcOf[}x.f, o\texttt{]}, \\ \ \texttt{Sink[}x.f, \texttt{SrcOf[}v.f, o\texttt{]]}\} & \text{if } x = v \wedge y = v \\ \emptyset & \text{if } x = v \wedge y \neq v \\ \{\texttt{SrcOf[}v.f, o\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.6}$$

$$\llbracket x \rrbracket(\texttt{Src[}v\texttt{]}) = \begin{cases} \{\texttt{Src[}v\texttt{]},\ \texttt{Sink[}x, \texttt{Src[}v\texttt{]]}\} & \text{if } x = v \\ \{\texttt{Src[}v\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.7}$$

$$\llbracket x \rrbracket(\texttt{SrcOf[}v, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}v, o\texttt{]},\ \texttt{Sink[}x, \texttt{SrcOf[}v, o\texttt{]]}\} & \text{if } x = v \\ \{\texttt{SrcOf[}v, o\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.8}$$

$$\llbracket s \rrbracket(\texttt{Sink[\_]}) = \emptyset \tag{4.9}$$

default:

$$\llbracket s \rrbracket(d) = \{d\} \tag{4.10}$$

Figure 4.2: Decomposed Normal Flow Function

of an variable within the scope of the analyzed program. Note: for the C data-type of structures the flow fact O generates a `Source` fact only for the struct name but not yet for its field members. In case of an field declaration, a fact for the field is generated and finally, in case of an curly initialization of a struct, for every field a `Source` fact is generated.

- **Right-Hand Side Assignment:** in contrast to the previously described flow function case for declarations at the left-hand side of an assignment, for the right-hand side of assignment statements we evaluate the incoming `Source` and `SourceOf` data-flow facts. In Equation 4.3 and 4.4 this behavior is outlined in a formal way. As previously is described, the very basic principle of our analysis strategy is to trace the flow of information. Consequently, for every incoming flow fact `Source` or `SourceOf`, which is used on the right-hand side of an assignment, we copy the reaching information value into a newly generated `SourceOf` fact for the assignment. This means, in case the incoming fact is a `Source` fact, we copy the fact directly into the generated `SourceOf` fact; if the incoming fact type is `SourceOf` we copy the nested origin `Source` fact o. Additionally, for every generated `SourceOf` fact our analysis reports a potential sink of the incoming `Source` or `SourceOf` fact in form of data-flow fact `SinkToAssignment` containing the incoming `Source` or `SourceOf` and the assignment statement.

  Finally, to cover the *killing* process of no longer valid `Source` and `SourceOf` facts: in case the variable name v of the incoming data-flow fact equals the name of the newly assigned variable, the flow fact is killed and no longer propagated along further edges as its values are replaced by new ones. Otherwise, or in case the incoming flow fact is not applicable on the statement, the data-flow fact is simply propagated further.

- **Struct Field:** for the C data-type of structs the previously described instances of the decomposed normal flow function apply as well. Nevertheless to keep the analysis field-sensitive, some additional cases must be considered in the transfer computation of data-flow facts which concern structs. Currently, we are able to track information on clean field reads and writes of structs, but we do not cover the case if the parent of a struct field is used in an assignment or is reassigned. In Equation 4.5 and 4.6 this case is processed: if the incoming data-flow fact contains a field reference to the corresponding parent struct used in the right-hand side of an assignment, the field value is copied into a new `SourceOf` fact of the newly declared struct. In case the parent struct of a field value is newly assigned and the incoming data-flow fact has a field reference of this struct, we kill the data-flow fact since it is no longer valid.

- **Expression Statements:** in our analysis scope, we consider any statement which does not introduce or alter the declaration or definition of a given variable as an *expression statement*. Nevertheless, as our analysis strategy aims to detect every potential information-flow within the analyzed program, we report for every used variable in expression statements potential sinks as well. Consequently, every incoming matching data-flow fact for the currently analyzed expression statement node generates a corresponding `SinkToUse` fact. Since a expression statement does not end the life-time of a variable or struct, all data-flow facts of the type `Source` are transferred to the successor flow function.

- **Visibility Scope:** the previously described techniques do consider different visibility scopes of the same symbol name. As outlined in the description of the used data-flow facts in our analysis, each `Source` data-flow fact contains its individual visibility scope within the analyzed program as an numeric value. In order to avoid false matching, we filter non-applicable data-flow facts according to their visibility scope before applying the described transfer functions. In case of normal flow functions we use the following strategy. First, if the name of the incoming data-flow fact has the same at the current program location as encoded within the flow fact, the transfer functions are applied. Second, if the visibility scope value of the incoming data-flow fact is smaller than the actual scope value for this name at the current statement location, the fact is propagated further down the execution path, but no flow function case is applied. This way, the information-flow of globally defined variables is propagated further down the execution path while being shadowed by a location definition using the same name. In case the data-flow fact has a greater numeric visibility scope than the matching symbol at the current source code instruction, the data-flow fact is killed since we left its visibility scope.
- **Default Case:** in case none of the presented decomposed flow functions is applicable, the incoming data-flow fact is propagated further along the edge to the successor node.

4.2.1.3 Call Flow Function

In contrast to the normal flow function, this flow function models the interprocedural data-flow from a call statement to the target callee method and is formally described in Figure 4.3. At this case, information encoded in variables or structs can reach the target callee method only if the corresponding variables are passed as function arguments in the call statement or are globally defined. To transfer relevant data-flow facts on this edge type the following strategy applies: likewise to the normal flow function, the default data-flow fact O generates the initial `Source` fact with the name of the function parameter fp at callee site. Further, a parameter cp of the call statement is treated like an assignment to the corresponding argument fp of the callee function. This way, we can propagate the flow of information in the same way as for assignments within the normal intraprocedural flow: every incoming data-flow fact `Source` or `SourceOf` which influences the value of the function parameter is encapsulated into a new `SourceOf` fact for this parameter. Further, to indicate a flow of information to a potential sink, likewise to previously described normal flow function, a `SinkToAssignment` data-flow fact is generated too. However, as the

$$\llbracket x \mapsfrom m_{Call}([cp, fp])\rrbracket(O) = \{\texttt{Src[}fp\texttt{]}\} \tag{4.11}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp])\rrbracket(\texttt{Src[}v\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}fp, \texttt{Src[}v\texttt{]]}, \\ \;\texttt{Sink[}fp, \texttt{Src[}v\texttt{]]}\} & \text{if } cp = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.12}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp])\rrbracket(\texttt{SrcOf[}v, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}fp, o\texttt{]}, \\ \;\texttt{Sink[}fp, \texttt{SrcOf[}v, o\texttt{]]}\} & \text{if } cp = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.13}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp])\rrbracket(\texttt{Src[}v.f\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}fp.f, \texttt{Src[}v.f\texttt{]]}, \\ \;\texttt{Sink[}fp.f, \texttt{Src[}v.f\texttt{]]}\} & \text{if } cp = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.14}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp])\rrbracket(\texttt{SrcOf[}v.f, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}fp.f, o\texttt{]}, \\ \;\texttt{Sink[}fp.f, \texttt{SrcOf[}v.f, o\texttt{]]}\} & \text{if } cp = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.15}$$

default:

$$\llbracket s\rrbracket(d) = \emptyset \tag{4.16}$$

Figure 4.3: Decomposed Call Flow Function

incoming data-flow fact is not valid in the scope of the callee function, it is killed on the interprocedural edge.

An exception to this rule are data-flow facts of the type `Source` which model globally visible variables. These facts are transferred by the call flow function as their information is visible at callee-site and may potentially be altered or reach a sink.

Finally, as default behavior for this flow function all other data-flow facts are killed as they do not reach the target function and their intraprocedural data-flow is computed by the call-to-return flow function.

Note: for built-in C platform functions or compiler specific functions, we report the sink to the used call parameter of the currently visited function since no further implementation details of such functions are available.

4.2.1.4 Return Flow Function

The return flow function computes data-flow facts from the exit statement of a called function back to the successor statement of the original call statement. The working principle of this flow function is similar to the principle of the call flow function: all intraprocedural data-flow facts of the callee function are killed as they are invalid in the scope

$$\llbracket x \mapsfrom m_{Ret}(y) \rrbracket(O) = \{\texttt{Src[}x\texttt{]}\} \tag{4.17}$$

$$\llbracket x \mapsfrom m_{Ret}(y) \rrbracket(\texttt{Src[}v\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}x, \texttt{Src[}v\texttt{]]},\ \texttt{Sink[}x, \texttt{Src[}v\texttt{]]}\} & \text{if } y = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.18}$$

$$\llbracket x \mapsfrom m_{Ret}(y) \rrbracket(\texttt{SrcOf[}v, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}x, o\texttt{]},\ \texttt{Sink[}x, \texttt{SrcOf[}v, o\texttt{]]}\} & \text{if } y = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.19}$$

$$\llbracket x \mapsfrom m_{Ret}(y) \rrbracket(\texttt{Src[}v.f\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}x.f, \texttt{Src[}v.f\texttt{]]}, & \\ \ \texttt{Sink[}x.f, \texttt{Src[}v.f\texttt{]]}\} & \text{if } y = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.20}$$

$$\llbracket x \mapsfrom m_{Ret}(y) \rrbracket(\texttt{SrcOf[}v.f, o\texttt{]}) = \begin{cases} \{\texttt{SrcOf[}x.f, o\texttt{]}, & \\ \ \texttt{Sink[}x.f, \texttt{SrcOf[}v.f, o\texttt{]]}\} & \text{if } y = v \\ \emptyset & \text{otherwise} \end{cases} \tag{4.21}$$

default:

$$\llbracket s \rrbracket(d) = \emptyset \tag{4.22}$$

Figure 4.4: Decomposed Return Flow Function

of the return-to function. No rule without exception: likewise to the call flow, data-flow facts with a globally visible scope value are transferred and not killed for the same reason already mentioned at the call flow function.

Nevertheless, information is transferred from the callee site to the call site via `return` statements. Again, our decomposed flow functions in Figure 4.4 compute this kind of information-flow in an assignment based matter. First, the default data-flow fact O generates the initial fact `Source` for assigned variables or structures at the call statement. Second, we reuse the previously described practice and map the argument of the `return` statement of the callee function to the assigned variable at call site. Similar to the call flow approach, every matching data-flow fact is encapsulated in a new fact `SourceOf` for the assigned variable while the incoming matching fact itself is killed, as it is no longer valid. Finally, a `SinkToAssignment` flow fact is generated for the assignment, to report the information-flow into the assigned variable or struct.

4.2.1.5 Call-to-Return Flow Function

Finally, the call-to-return flow function (cf. Figure 4.5) is the transfer function for data-flow facts from the call statement to its intraprocedural successor statement. In contrast to the previously described flow functions, this function is rather simple. This function propagates all incoming data-flow facts further in the exploded supergraph. Except for

$$\llbracket x \mapsfrom m_{Call}([cp, fp]) \rrbracket(\texttt{Src[}v\texttt{]}) = \begin{cases} \emptyset & \text{if } x = v \\ \{\texttt{Src[}v\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.23}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp]) \rrbracket(\texttt{SrcOf[}v, o\texttt{]}) = \begin{cases} \emptyset & \text{if } x = v \\ \{\texttt{SrcOf[}v, o\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.24}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp]) \rrbracket(\texttt{Src[}v.f\texttt{]}) = \begin{cases} \emptyset & \text{if } x = v \\ \{\texttt{Src[}v.f\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.25}$$

$$\llbracket x \mapsfrom m_{Call}([cp, fp]) \rrbracket(\texttt{SrcOf[}v.f, o\texttt{]}) = \begin{cases} \emptyset & \text{if } x = v \\ \{\texttt{SrcOf[}v.f, o\texttt{]}\} & \text{otherwise} \end{cases} \tag{4.26}$$

$$\llbracket s \rrbracket(\texttt{Sink[\_]}) = \emptyset \tag{4.27}$$

default:

$$\llbracket s \rrbracket(d) = \{d\} \tag{4.28}$$

Figure 4.5: Decomposed Call-to-Return Flow Function

matching `Source` or `SourceOf` facts which are reassigned or declared at the function call site (e.g. `x` at a call statement `x = foo()`) are *killed*, as they can no longer reach any further sink and the according new data-flow facts are introduced by the return flow edge. These facts are computed, as previously explained, by the call-flow respectively return flow function, as theses facts are influenced by the target callee procedure. Additionally, all data-flow facts modeling global symbols are killed as well since they also are propagated by the call-flow respectively return flow function. The matching strategy for struct fields equals directly to the method used in the previously presented flow functions.

4.2.2 Exploded Supergraph

```
int cipher2(int i) {
    return i;
};
int main() { [...]
    secret = 666;
    sink = c->func(secret);
    printf("%i\n", sink);
    return 0;
};
```

Listing 4.2: Product variant derived from a configurable software system (cf. Figure 1.1)

In Figure 4.6 we illustrate the working principle of our information-flow analysis strategy. For that purpose, we apply our analysis on a single product variant derived from our running example of a configurable software system (cf. Figure 1.1) and display the resulting exploded supergraph. For the sake of the example, we did choose the product with no enabled feature condition: $\neg(A \wedge B \wedge C \wedge D)$. Additionally, we inlined the intermediate function call of the procedure `ctx_do` and show the information-flow analysis from the initial assignment to `secret`. In Listing 4.2 the resulting product source code is displayed. For simplicity reasons, only interesting data-flow facts for explanation purpose are shown in the illustration.

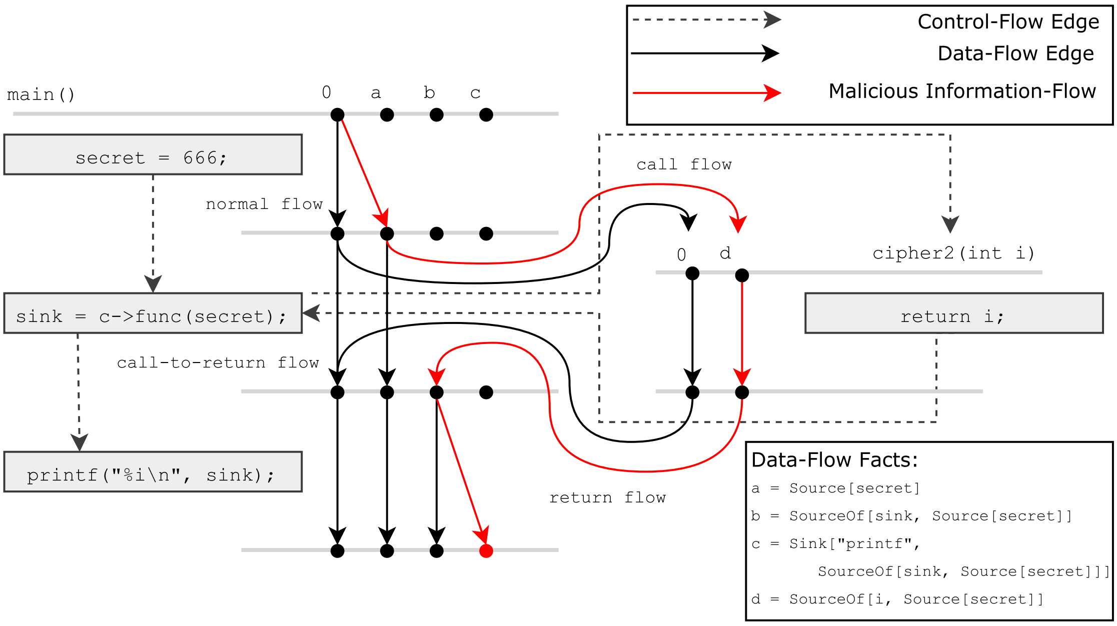


Figure 4.6: Exploded Supergraph of our Information-Flow Analysis applied on the product in Listing 4.2

In our product example, we trace the information-flow of the variable `secret` from its origin definition to a potential sink - in the example encoded as `printf` statement. The initial `Source` fact a for the variable `secret` is introduced at the very first normal flow edge by the default data-flow fact of the IFDS-framework O. As successor statement of the definition of `secret` we directly reach a function call statement. The target destination of this function-call statement is the method `cipher2` (function pointer assignment is not shown in Listing 4.2). At the call statement, the data-flow continues on two different edges of the exploded supergraph: (1) the call-to-return flow edge which targets the intraprocedural successor statement of the current call statement and (2) the call flow edge which is destined to the callee location of the call statement. On the call-to-return flow edge all existing data-flow facts, such as the `Source` fact a of the variable `secret`, are propagated further down the graph, as none of the existing facts is reassigned and therefore killed. While on the call edge only the default data-flow fact O and a newly generated data-flow fact d are propagated. This new fact d is not generated by the default data-flow fact O, but by the `Source` fact a of `secret`, as the call parameter matches the callee parameter `i` of the method `cipher2`. As previously explained, this way we are able to trace the flow of information across assignment boundaries.

Due to the fact, that the target callee function `cipher2` consists only of a single `return` statement in this product variant, the control- and data-flow immediately return back to the call-site of the procedure. As formulated in the previous chapter, the flow function of the return edge of the supergraph transfers information from the return statement to any assignment at call-site. In the analyzed product variant the value of `i` is assigned to the variable `sink`. This case is expressed in our analysis strategy as follows: first, as the value of the callee parameter of `i` reaches its end of lifetime, the corresponding data-flow fact d is killed. Second, as the information-flow of `secret` continues, this data-flow fact generates a new `Source` data-flow fact b for the assigned variable of `sink` back at call-site.

After both flow edges, which originated from the call-site, are joined back together, a last normal flow edge is computed. This final edge propagates all incoming data-flow facts of the type `Source` further down the control-flow within the supergraph as no new variable or struct is introduced at this line. However, our analysis detects a potential information leakage as the argument `sink` of the `printf` instruction has a the incoming `Source` data-flow fact of this variable contains a reference to `secret`. Our analysis reports the sink of information to any given statement by generating a short-term data-flow fact of the type `Sink`.

4.2.3 Discussion

The presented information-flow analysis problem for the programming language C is a basic approach. First of all, like similar static code analysis [Lie15] for C, we ignore the presence of the concept of pointers and treat them like normal variables. Hence, all potential information-flow caused by pointer aliasing, is not examined. Additionally, as the concept of arrays in the programming language C is very closely related to the concept of pointers, our analysis is imprecise at this point as well: array fields are computed field-insensitive. Finally, in the nature of static code analysis, we do not evaluate arithmetic expressions.

In the scope of this thesis, we focus on enabling interprocedural data-flow analysis for configurable software-systems written in C and formulated the presented IFDS problem as a *proof-of-concept* data-flow analysis for this task. For a more precise and sophisticated information-flow analysis, one must introduce an alias analysis into the IFDS abstraction. However, as previously stated (cf. Section 3.1.2), this is a daunting task.

Another possible approach to trace the flow of information across assignments is the use of *static single assignment* (SSA) form [NNH15]. By using static single assignment form, a variable is defined and declared exactly once. On the other hand, in real-life programming languages like C, it is possible to redeclare variables again. As a consequence, ordinary C source code must be rewritten to match static single assignment form. Despite the advantage of using SSA form to simplify the analysis strategy, we did choose not to apply SSA form in our analysis strategy as we focus to enable interprocedural data-flow analysis on configurable software systems written in C. Our developed analysis strategy is a *proof-of-concept* of this approach. In the scope of this thesis, we want to show the universal correctness and feasibility of this strategy for all kinds of IFDS based data-flow analysis problem. Consequently, we decided to avoid any analysis specific rewrite operations on the intermediate source code representation.

5. Variability-Aware Solving of IFDS-Problems with SPL$^{\text{LIFT}}$

Bodden et al. developed the tool SPL$^{\text{LIFT}}$ to lift existing data-flow analysis problems formulated in the previously described IFDS-framework on configurable software systems [BTR^{+}13]. To do so, they exploit the fact, that IFDS-based data-flow problems can be expressed as a data-flow problem in the *interprocedural distributive environment transformers*-framework (IDE). In this chapter, we introduce the concept of the IDE-framework. Further, we describe the correlation of the IFDS-framework to the IDE-framework. Finally, we will explain our adaption of the approach of Bodden to lift data-flow analysis problems formulated within the IFDS-framework on configurable software systems written in C, how we could simplify this approach while improving its generality, precision with the elimination of existing limitations and illustrate it with our proof of concept data-flow analysis technique.

5.1 IDE-Framework

Likewise IFDS, the IDE-framework by the same authors, Sagiv, Reps and Horwitz, is a data-flow analysis framework, which models as well data-flow through edges in an exploded supergraph, but it is more abstract and expressive than IFDS [SRH96], while its complexity remains the same as for IFDS: $O(ED^3)$.

In contrast to IFDS, data-flow analysis problems in IDE are not limited and reduced to a simple graph reachability problem, but can additionally compute values from a separate and independent domain V, called *value domain*, along the graph's edges [SRH96, Bod12]. For every reachable data-flow fact d from the original domain D at a given statement, the IDE's algorithm computes all values $v \in V$ along all valid paths in the exploded supergraph to d. Hereby, every data-flow fact d of each node in the supergraph gets mapped to a value v. In order to express data-flow analysis problems formulated in the IFDS-framework as an IDE-based problems, one can use a *binary value domain* $\{\bot, \top\}$, which maps $d \mapsto \bot$ for every data-flow fact d that holds at the current statement, or either $d \mapsto \top$ for every fact that does not hold.

Obviously, the described *binary value domain* is not limited to only two elements, but can hold a much large value set. This is the main concept of SPL$^{\text{LIFT}}$: by replacing the *binary value domain* with a value domain consisting of presence-conditions, one is able to lift IFDS-based problems on configurable software systems.

5.2 Lifting Flow Functions

The very basic principle how Bodden et al. analyze configurable software systems at once is as follows: assume a given source code statement s is annotated with the presence-condition C. Then it should only have effect to the data-flow computation if its presence-condition is fulfilled, otherwise not [BTR+13]. To achieve this, Bodden et al. *lift* existing flow functions of a given IFDS based data-flow analysis problem at variable nodes in the CFG. Hereby, they introduce two different IFDS flow functions for variable statements: f^C if the presence-condition C holds and for the alternative case $f^{\neg C}$. These two flow functions are combined into a single, lifted flow function:

$$f^{LIFT} := f^C \vee f^{\neg C}$$

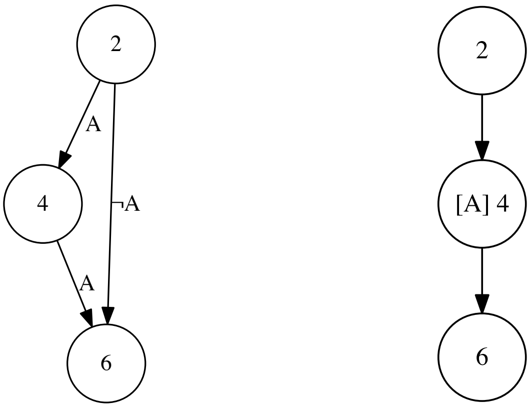
Bodden et al. formulated a set of rules for this combination process depending on the individual flow function (i.e. normal flow, call flow) as well as occurring branching of statements [BTR+13]. These rules are formulated as analysis problem for the data-flow analysis framework IDE. For this set of rules, they did show that it outperforms an traditional approach of analyzing each variant individually for CIDE based configurable software systems. In the concept of this lifting technique, there is no indication, that this approach can not be applied on configurable software systems written in C as well. Its only requirement is an interprocedural control-flow graph of the target configurable software system's programming language.

5.3 Different Control-Flow Graph Concepts

Unfortunately, Bodden's approach can not be applied in the proposed way while using TypeChef as provider of the required interprocedural control-flow graph to analyze configurable software systems written in C. This is caused by the fact, that the interprocedural CFG provided by the underlying tool-chain of SPL$^{\text{LIFT}}$ differs significant from the CFG provided by the TypeChef infrastructure in the matter of variability-encoding.

As described in Section 2.2.2, the control-flow graph provided by the TypeChef infrastructure encodes variability on the graph's edges as presence-conditions. This concept leads to individual edges for every possible successor node from a single source node within the CFG, which is annotated according to its control-flow presence-condition. In contrast, the control-flow graph concept used by SPL$^{\text{LIFT}}$ does not encode variability on its edges, but only in its nodes as presence-conditions [BRT+13]. Unlike the CFG provided by TypeChef, this CFG does not contain every possible execution path like classical, textbook-based control-flow graph representations [NNH15], but rather a consecutive order of statements as they occur in the source code. This esoteric type of CFG is called *instrumented CFG* [BRT+13]. In Figure 5.1 we illustrate the major difference between both control-flow graph concepts for a very basic configurable software source code snippet. We can see in the code snippet, that the instruction in Line 4 is only executed if

```
1  int foo(int x) {
2      int y = x;
3  #ifdef A
4      y = 2;
5  #endif
6      return y
7  }
```



(a) Variational CFG (b) Instrumented CFG

Figure 5.1: Comparison of both CFG concepts with a minimal variable source code fragment

configuration option A is selected. As described, the variational CFG generates unique edges for the different execution paths. In comparison, in the instrument version of the CFG, its edges follow the instruction set within the source code and encode variability into the CFG node. Consequently, not for every potential successor node a direct outgoing edge from a given source node exists, as for the example source code, no direct edge between the statements in Line 2 and 6.

The overall complexity of the instrumented CFG is less than the complexity of the variational CFG while modeling the same control-flow, as the instrument CFG contains less flow edges. On the other hand, it increases the complexity of the lifting process itself, as not every execution path through the graph is represented as an individual edge. As a consequence, an additional layer of computation must be added to the underlying IDE problem, which "*lifts*" IFDS flow functions on variable control-flow nodes in order to determine under which condition the current flow function influences the data-flow fact computation or not. Bodden et al. resolved this issue, as previously described, with the formulation of different combination rules for IFDS flow functions at such variable CFG nodes. Unfortunately, from a user's perspective, this lifting process is *not* as transparent as promoted [BTR+13]: consider again the minimal code example for comparing both CFG concepts in Figure 5.1. The used IFDS/IDE solver by SPL$^{\text{LIFT}}$, HEROS, defines in its interface for IFDS flow functions the normal flow function with the following signature to be implemented by the concrete analysis [Bod12]: `getNormalFlowFunction(N src, N succ)`. The argument `src` denotes to the current source node in the CFG, while the argument `succ` represents its successor node. Consequently, when using an instrumented CFG, the normal flow function for the valid execution path Src:= `int y = x` and $Succ$:= `return y` in our minimal, variable code snippet is never analyzed in the mentioned combination. This will most likely cause erroneous behavior, if an IFDS data-flow analysis validly computes both statements, Src and $Succ$, together.

5.4 Variability-Aware Solving of IFDS Problems in IDE

As explained before, IFDS problems can be fully expressed within the related IDE-framework. Likewise IFDS, the IDE-framework uses the same exploded supergraph. We benefit massively of the fact, that intraprocedural edges of the exploded supergraph map directly to edges in the variational control-flow graph and outgoing edges from a given source node in the variational CFG cover all possible variants. This means that separate control-flow paths for different product variants are modeled *exactly* in the same way as branched execution paths introduced by control-flow statements, such as `if-then-else`, in the exploded supergraph. Likewise to classically branched control-flows, separate paths are joined back together in case the branching ends. Since the variational control-flow graph reports for every edge its individual presence-condition, we are able to reduce the complexity of the solving process down to the initial computation process of IFDS problems as single IDE problem. As a result, we must not lift any flow function of the IFDS-framework by the suggested combination rules.

To trace precisely the individual flow presence-condition of variational execution paths, we replace the binary value domain $V := \{\bot, \top\}$ with a value domain $V_{\mathcal{PC}} := \{\Phi\}$, which encodes presence-conditions as *boolean formula* Φ. Hereby, we change the value $\top$ for every data-flow fact d which does not hold at the current flow function to the boolean value of $False$ and the value $\bot$ to the presence-condition of the current edge $\mathcal{PC}_E$ in the CFG, expressed as boolean formula, for every flow fact d that does hold on this edge. As distributive join operator for joining points of separate control-flow paths of data-flow fact through the supergraph, we use the boolean disjunction operation $\vee$. This way, we precisely combine all valid variants together for which a data-flow fact holds. Straightforward to this concept, the composition operator of consecutive flow functions in the exploded supergraph is modelled by the boolean conjunction operator $\wedge$, so that a data-flow fact can only reach a certain target node from its source node within the graph if all presence-conditions are satisfiable at at least one path through the CFG.

Simply by replacing the binary value domain and using the variational CFG one already can successfully analyze IFDS based data-flow analysis problems simultaneously on each possible variant of a configurable software-system. This concept shares commonalities with the underlying principle of the original approach of Bodden et al. in SPL$^{\text{LIFT}}$, but with the major difference, that we are able to *completely* waive the various complex rules for combining flow functions at variable nodes in the instrumented CFG for the tool-combination of TypeChef and SPL$^{\text{LIFT}}$. Further, as every control-flow edge encodes its individual flow presence-condition, we are able to report for each and every potential execution path through the analyzed program its distinct flow condition. Last but not least, our concept brings the tremendous advantage to avoid the possibility that the formulated combination rules for lifting flow functions are *non-exhaustive* regarding unstructured control-flow within the C programming language such as jump statements.

Additionally, from a user's perspective, the analysis is solved variability-aware in a complete transparent way, as the underlying CFG reports all possible execution paths of different variants, while being able to solve IFDS data-flow analysis simultaneously instead of individually each variant of a configurable software system. This concept is in line with previous work on data-flow analysis techniques for configurable software systems using the TypeChef infrastructure and is proven to be efficient and scalable [LvRK+13, LJG+15, Lie15, vR16]. Finally, Bodden et al. did include the feature model of the analyzed system (cf. Section 2.1.2.1) into the set of combination rules for lifting to distinguish valid variants from invalid ones in order to reduce the overall computation costs. The variability-aware control-flow graph as well does incorporate the feature model in its successor computation and reports only valid execution paths in the source code according to the feature model.

Because of this reasons, we favor the variational control-flow graph over the instrumented one. Further, the use of the variability-aware control-flow graph allows us to simplify the lifting mechanism in SPL$^{\text{LIFT}}$, as the combination of different flow functions into a lifted function is no longer necessary.

5.5 Application to the Running Example

Likewise to the illustration of our information-flow analysis in Figure 4.6, we show in Figure 5.2 a truncated version of the supergraph applied on the example configurable software system from Figure 1.1 using the same information-flow analysis. However, this time we do not analyze a single variant in isolation, but use the presented solving strategy to analyze all potential variants simultaneously in a single analysis run. As we can see, the computation result detects the violating flow of information for both affected variants as outlined in the problem statement of this Thesis: $(A \wedge \neg B \wedge C) \vee (\neg A \wedge \neg D)$.

The most prominent difference between the previously presented exploded supergraph in Figure 4.6 for a single variant and the current variability-aware version, is the additional outgoing call flow edge from the call statement in the `main` function. Based on the chosen configuration, the destination of the function pointer `c->func()` targets either method `cipher1` or `cipher2`. As a logical consequence, our solving strategy annotates both call flow edges with their corresponding flow presence-condition: in our case A for the edge targeting `cipher1` and $\neg A$ for the edge describing the flow to `cipher2`. Every single data-flow fact which is transferred over the described edges is conjuncted with the corresponding presence-condition of the propagation edge. In our illustration the newly generated `Source` data-flow facts for both function parameters are propagated with the presence-condition of either A or $\neg A$ in respect to the target callee function.

To describe the working principle of conditional normal flow edges, we continue the control-flow at the method `cipher2`. Depending on the configuration, the statement `i = 0` is either executed or not. As outlined before, our CFG concept reports for such variable statements two different execution paths. Note: these execution paths are annotated only

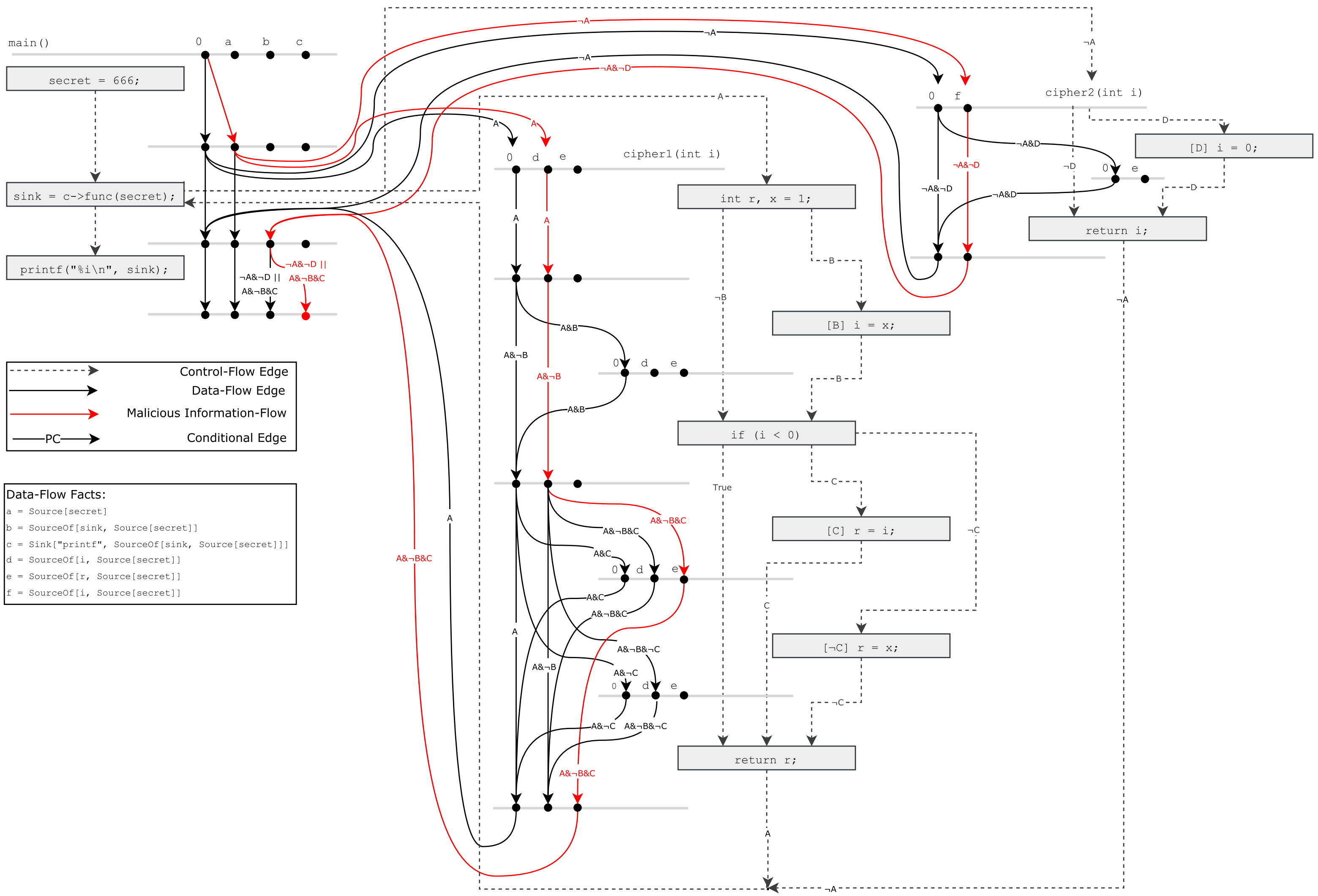


Figure 5.2: Exploded Super Graph of our Information-Flow Analysis applied simultaneously on all products of our example from Figure 1.1

with the surrounding presence-condition of D or $\neg D$ since the CFG itself is intraprocedural at this point and consequently unaware of any call-to-callee flow constraints. But both displayed incoming data-flow edges are annotated with the correct points-to flow presence-condition. So, for both flow edges of the default flow fact O at the entry point of the function `cipher2`, we can report the correct flow condition as we conjunct the incoming flow condition together with the reported intraprocedural flow condition by the variational CFG. With this strategy we detect, that a reference to the information of variable `secret` is only propagated further if both conditions, A and D, do not hold, as under condition D variable `i` is reassigned and on this single execution path all incoming data-flow facts matching to `i` are killed, which includes the reference fact to the variable of `secret`.

Within the method of `cipher1` we continue the same way of proceeding. This time however, we want to explain the join mechanism for conditionally branched execution paths. Our variability-aware CFG reports three possible successor statements for the control statement `if`. In case the condition of the `if`-clause is fulfilled, two different execution paths are executed depending on the chosen configuration, while the fall-through path is equal in every configuration. Consequently, the default data-flow fact O is propagated further in the supergraph on three different edges. Since all three edges eventually meet again at the function's exit statement, their individual flow conditions are merged together by the boolean disjunction operation and the fact O is finally transferred back to the return site of the call under condition A. In the same manner we detect, that only in one specific configuration a malicious information-flow through function `cipher1` exists.

Since both return flow edges of `cipher1` and `cipher2` report that the value of `secret` is encoded in the variable reaching each function's own `return` statement, the return flow function generates for the assigned value of `sink` by the function-call statement a `Source` data-flow fact which references `secret`. Likewise to the previously presented product-based approach, our information-flow analysis indicates a sink of information referencing the value of `secret` to the `printf` instruction. This time however, the potential information-leakage is reported under which configurations of the analyzed software system it occurs: $(A \wedge \neg B \wedge C) \vee (\neg A \wedge \neg D)$.

5.6 Further Improvements

Additionally, our presented approach allows us to overcome some limitations of the existing SPL$^{\text{LIFT}}$ lifting mechanism. The implementation of the data-flow analysis framework SPL$^{\text{LIFT}}$ is not as sensitive to presence-condition annotations as it could be, as it completely ignores aliasing at call-site. Consider again our motivating example from the beginning in Figure 1.1: depending on the chosen configuration, the function pointer of `c->func()` points to either the procedure `cipher1` if condition `A` holds, and otherwise to procedure `cipher2`, which is correctly determined by our function-pointer target analysis strategy (cf. Section 3.1.2). SPL$^{\text{LIFT}}$'s internal call flow lifting rules do not compute the presence-condition for different target destinations and assumes for these edges the flow presence-condition of $True$.

Bodden et al. did propose a potential solution for this limitation by handling the points-to analysis as part of the IFDS abstraction. In a first attempt, we did try to solve this issue by means of the proposed solution, but recognized that it would massively violate the idea of transparent lifting of IFDS-based data-flow analysis problems, as we would have had to encode the current flow condition into every single data-flow fact of the performed analysis. However, we did solve this short-coming by a combination of two aspects: First by extending the control-flow graph interface by an additional method, which reports the correct presence-condition of the flow between the call- and its target callee-statement in case of aliasing:

$$\textsf{getPointsToCondition}\colon V[CFGStmt] \times V[CFGFDef] \rightarrow \mathcal{PC}$$

With the use of this function, we can set the correct flow presence-condition for call and return edges during computation-time. Second, by resolving a imprecision within the used solver HEROS. We comment on this essential fix in the following section.

5.7 Implementation Details on the Correct Tracing of Control-Flow Presence-Conditions

As mentioned before, to solve variability-unaware IFDS data-flow analysis variability-aware, we express them as IDE problem and replace the binary value domain with a domain consisting of presence-conditions. An IDE analysis uses the exact exploded supergraph like IFDS and consequently uses exactly the same graph edges (cf. Section 4.1).

5.7.1 Edge Functions

Remember: as explained in Section 2.2.2, the successor function of the control-flow graph provided by the TYPECHEF infrastructure encodes the presence-condition of the flow edge between the source statement and its successors in every individual successor node: $succ : CFGStmt \rightarrow List[V[CFGStmt]]$. Beginning at the initially selected starting node, all further visited nodes by the IDE algorithm are consequently provided by this successor function. In its low-level implementation, this successor function remains intraprocedural, as it is a basic requirement of the normal edge in the exploded supergraph of the IDE/IFDS-framework (cf. Section 4.1). The surrounding control-flow graph provides the required mechanism for interprocedural control-flow (cf. Section 3.2.1). In order to determine the exact presence-condition $\mathcal{PC}_{Edge}$ for different flow edges in the exploded supergraph, we need to compute the presence-condition of the incoming edge together with the presence-condition of the outgoing edge using the following rules:

normal edge: the default intraprocedural edge within the supergraph knows its source node Src and its successor node $Succ$. The presence-condition of this edge is determined by combining the presence-condition of the incoming edge of the Src node together with the presence-condition of its outgoing edge to its successor $Succ$:

$$\mathcal{PC}_{Edge} := \mathcal{PC}(Src) \land \mathcal{PC}(Succ) \tag{5.1}$$

call edge: a interprocedural edge modelling the flow from a call statement to the corresponding callee site. This time we are aware of the presence-condition of the incoming edge of the $Call$ node. Likewise to the previous edge, the resulting $\mathcal{PC}_{Edge}$ is the combination of both presence-conditions. This is sound and precise, as long no aliasing occurs, since for the target callee node we only know its condition according to its presence within the source code. To avoid imprecision in case of aliasing, our graph interface offers an additional procedure, which returns the presence-condition for call flows with aliasing:

$$\mathcal{PC}_{Edge} := \mathcal{PC}(Call) \land \mathcal{PC}(Dest) \land \mathcal{P}oints\mathcal{T}o\mathcal{PC}(Call, Dest) \tag{5.2}$$

return edge: an interprocedural edge pointing from the return statement of the callee site to the return-site of the call statement. This edge is similar to the computation of the presence-condition of the call edge, however this time we have to compute additionally the presence-condition of the edge from the $Exit$ node to the intraprocedural successor of the call node, $ReturnSite$.

$$\begin{aligned} \mathcal{PC}_{Edge} := \mathcal{PC}(Call) \land \mathcal{PC}(Dest) \land\ & \mathcal{P}oints\mathcal{T}o\mathcal{PC}(Call, Dest) \\ & \land \mathcal{PC}(Exit) \land \mathcal{PC}(ReturnSite) \end{aligned} \tag{5.3}$$

call-to-return edge: since call statements are excluded from normal edges, this edge models the flow from the $Call$ node to its intraprocedural successor node called $ReturnSite$. Consequently, its presence-condition is determined in the same manner as for the normal edge:

$$\mathcal{PC}_{Edge} := \mathcal{PC}(Call) \land \mathcal{PC}(ReturnSite) \tag{5.4}$$

5.7.2 Additions to the IDE/IFDS Solver HEROS

During the phase of implementation and evaluation of our previsiouly presented IFDS "lifting" methodology we witnessed, that our approach did generate for some information-flows an imprecise, and consequently false result. Hereby we observed, that for intraprocedural flow edges of the exploded supergraph in IDE within conditionally called procedures, a false CFG-flow presence-condition was reported: the condition under which the call occurred was missing. However, every already existing and additionally every newly generated data-flow fact which was propagated over the return edge of the currently examined procedure, did report its own correct flow presence-condition. At first, we estimated that our approach has some unidentified flaw in its concept or implementation. We assumed, that the underlying IDE/IFDS-solver HEROS of SPL$^{\text{LIFT}}$ should provide a sound solving result because of its widespread usage in several other reviewed data-flow analysis projects. Nevertheless, during extensive debugging we traced the origin of the falsely reported flow condition back to the implementation of the used solver. We identified, that the solver propagates for edges from call-site to callee-site the *identity function* of graph edges. In case IFDS data-flow problems are expressed in the classical, variability-unaware way within the IDE-framework, in our opinion this behavior should be correct as the identity function basically expresses the value $\perp$ of the binary value domain V which represents the binary value for all holding data-flow facts on a given supergraph edge. However, in our approach this tiny implementation detail did create the observed imprecision as the identity function reports a flow presence-condition of $True$. Hence, we did not compute occurring flow presence-condition on this edge. This issue lead to the fact, that every subsequently generated data-flow fact was missing the presence-condition of the flow edge.

To overcome this fundamental issue, we removed the propagation of the identity function at this point in the solver implementation. Instead, we propagate the composition of the incoming edge from the predecessor node in the supergraph together with the call edge function. This solution does not alter the behavior for the classic expression of IFDS problems within IDE in any way, but correctly propagates all previously computed data-flow fact conditions further down the execution path along with the extracted call flow presence-condition in our approach.

Additionally, in this process we identified and resolved another potential cause for imprecision and yet in the original implementation of SPL$^{\text{LIFT}}$ completely ignored location of variability. As previously described, the IDE/IFDS framework does compute a set of upfront data-flow facts at its starting point called *initial seeds*. In our information-flow analysis strategy we use these initial seeds to propagate the initial values of global variables. Unfortunately, these seeds are introduced with the identity function of graph edges. Consequently, these seed data-flow facts are all propagated with the presence-condition of $True$. But, these seeds may not be present in every possible configuration of the analyzed software-system. To resolve this additional imprecision, we added an mechanism which propagates initial data-flow facts according to their unique presence-condition at the starting point of the analysis.

6. Evaluation

In a series of experiments we evaluate in this chapter the correctness, feasibility and scalability of our approach to analyze data-flow problems formulated within the IFDS-framework simultaneously on all possible variants of configurable software systems written in the programming language C. We use these experiments to answer the following research questions about our implementation:

RQ1: Does our analysis strategy compute a correct result?

RQ2: Is our technique efficient (scalable) on real-life and large-scale configurable software systems?

Finally, we discuss our experiences of applying the presented information-flow analysis as taint checking technique on the implementation of the AES-encryption algorithm of two different configurable real-world open-source cryptography libraries.

6.1 Subject Systems

To conduct our experiments we choose two different real-world configurable subject software systems: MBEDTLS and OPENSSL. Both software systems are open-source cryptography libraries and implement all essential state-of-the-art cryptographic functions.

6.1.1 MBEDTLS

MBEDTLS[1], formerly known as POLARSSL, is a light-weighted cryptography library designed to fit on small embedded devices. It is highly configurable: each feature, such as a cryptographic function, can be configured independently from the rest of the library. In our experiments we used version 2.2.1 of the library. This version in numbers: 249 distinct feature options for configuration and a total number of 54 809 lines of source code in 120 different source code files. These numbers include an example set of 34 programs which show ways how to use the library. Since software libraries are commonly not designed to be used as standalone modules, we use these example programs as entry points for our experiments. MBEDTLS is in practical use in well-known real-world open-source projects such as OPENVPN or is embedded in internet of things hardware-devices such as network-routers by the firm LINKSYS.

[1] https://tls.mbed.org/

6.1.2 OpenSSL

In contrast to mbedTLS, OpenSSL[2] is the aged and bloated alternative version of a cryptography library. Since OpenSSL is the de-facto standard cryptography library used by web servers to ensure the confidentiality and integrity of communication over the internet, we examined this library as well. We used OpenSSL version 1.0.1 with 233 450 lines of source code in 733 different files and a total of 589 configuration options. In contrast to mbedTLS, OpenSSL does not provide a set of example programs. Notwithstanding, in order to analyze OpenSSL in similar fashion to mbedTLS, we used the provided example in the documentation of OpenSSL[3] for facilitating AES encryption with OpenSSL. Our example uses the C based implementation of AES and omits the implementation using the *AES new instruction set* of modern CPUs, because this implementation is completely written in assembler. Unfortunately, this reason limits our experiments to the implementation of the AES cipher in OpenSSL. But, there is no known obstacle to apply our approach on other parts of the library as well, given a potential entry point for the analysis.

6.2 Experiment Setup

To answer both of our research questions, we use the following experimental setup to retrieve sufficient data for that task.

Before we can conduct our experiments some upfront preparation tasks need to be performed. As outlined in Section 3.1.1, to resolve the potential destination file of externally defined and locally called functions, in a first step each source code file of the target case study needs to be analyzed to collect all necessary information for acquiring a linking map. For this task, we are required to run TypeChef once on each source code file. Hereby, a variational AST is build after lexing and parsing the input file. The resulting AST is type-checked by the TypeChef infrastructure in order to extract the signatures of locally and externally defined functions. After every single file has been analyzed, a global linking map is generated based on the type-check results of every single file reporting all exported and imported function definitions along with their signatures and presence-conditions. Since we are forced to parse each individual source code file prior our actual experiments, we exploit this fact and save all generated ASTs. This way we are able to reduce significant the overall time of our experiments as we are no longer required to lex and parse a source code file for every single analysis run which may take up to several minutes. After this initial precomputing task, we are able to conduct our experiments.

In order to answer our research questions we compare the classic approach of analyzing single variants of a configurable software systems against our approach of analyzing every variant simultaneously in a single run. Therefore two data sets are required for every independent analysis run: (1) a set of all found information-flows and (2) benchmark values regarding the run-time of interesting parts within the analysis.

[2] https://www.openssl.org/

[3] https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption

To collect the described data, we setup our analysis infrastructure as follows: for every program from our experiment set we apply our information-flow analysis. In this analysis process, we use the precomputed AST representation of the individual source code file. Externally linked files, as detected by the linking map, are loaded on-the-fly into the analysis scope. At this point, we use the precomputed AST representation as well. Since we want to analyze single products too, our testing infrastructure provides mechanism to derive a certain product from a given configuration. This product derivation process takes as input the previously precomputed variational AST representation of a source code file and a set of activated and/or deactivated configuration options. Based on this configuration set and the configuration knowledge encoded in the feature model, we are able to generate a valid product configuration. An AST representation for the generated configuration is produced by pruning all irrelevant branches of the input variational AST, so that no variability remains. For every single analysis run, regardless wether variational or product-based, we log each found information-flow data-flow fact, and in case of variability-aware analysis, the reported CFG flow presence-condition. Our analysis logs for every run a set of runtime values for different analysis tasks, such as AST (re-)loading, AST rewrite operations, product generation and solving time of the information-flow problem. In order to avoid bias due to external influences on our testing machines, we measure the elapsed process CPU execution time reported by the Java Virtual Machine (VM) between two time marks.

We conducted all of our experiments on LINUX machines running on Intel Xeon E5-2690 CPUs at 3.0 GHz. Since our testing machines are part of a shared high-performance computer cluster, we reserved exclusively for every running instance of the Java VM 16GB of RAM and 4 CPU cores. We configured the VM to use 12GB maximal heap space and to use the G1 garbage collector.

6.3 Research Questions

Following we answer the previously formulated research questions with the help of our testing infrastructure.

6.3.1 RQ1: Correctness

Since we altered Bodden's original approach of solving variability-unaware IFDS data-flow problems transparently in variability-aware manner, we need to re-ensure that our approach as well reports a correct result and is not overly restrictive. In contrast to Bodden et al. [BTR$^+$13], we do not have a second analysis tool to cross-check our computation result, since we are to our knowledge the first ones to conduct this kind of data-flow analysis on variable C code using the TYPECHEF infrastructure in combination with the data-flow analysis framework IFDS. That being said, we still are able to validate the correctness of our results in the matter of variability-awareness.

The most naive approach to validate, that our variability-aware solving strategy of IFDS data-flow problems is reporting correct results would be to cross-check the results of our strategy against the reported results of every single possible product variant of a configurable software system. However, as previously outlined in the problem statement, this task is infeasible. In order to reduce the number of configurations to check against our approach, we did choose the following approach using the previously described experimental setup. First, we run our analysis on every program of our experiment set using the presented variational solving strategy. Hereby, we log every information-flow reported by our IFDS data-flow analysis along with the reported product configuration of each individual information-flow. By this approach, we avoid testing unaffected or already covered variants [KBBK10]. Based on this result set, we generate every product for that our analysis reported an information-flow and analyze these products as well. Afterwards we compare, if every reported flow of information for a specific product variant is found as well in the corresponding product. Further we cross-check, if the analysis strategy reports any information-flows which are not covered by our approach. Ideally for a correct result, the number of found information-flows is equal in the product-based verification analysis to the number of reported information-flows by our presented variational approach.

In order to cross-check our results the other way around, we use a heuristic. Due to the fact, that analyzing each variant individually is infeasible, we use knowledge about dependencies between configuration options and derive *sample sets of configurations* which we then analyze. Sampling is a well-known technique to reduce the testing space of validating software [NL11]. Several different heuristics exist in order to perform sampling: in the scope of this thesis, we did choose *Code Coverage* as sampling heuristic. This heuristic aims to generate a minimal set of configurations of the analyzed target configurable system, that cover every single code fragment of the source code base in at least one valid configuration. This way, we ensure that every instruction is analyzed once. Given the fact that we are analyzing C software, we have to address the issue of header files in respect to code coverage. The concept of including header files is very common in the C programming language and header files even include additional header files. For example, the well-known "*Hello, World*" program in C includes a total of 19 different header files from the C standard library[4]. Likewise plain C source code files (*.c), header files (*.h) as well contain variable code fragments controlled by CPP directives. But as outlined, the number of included header files is relatively large even for very basic C programs. Although we are using an optimized algorithm for generating code coverage configurations [TLD$^+$11], this fact increases the computation cost significant. Further, we observed for the subject system OPENSSL, that including header variability blows up the configuration set with product variants that do not affect the experiment program at all, since these header files contain instructions which are used somewhere in the OPENSSL library, but not in the program used in our experiments. Likewise similar experiment setups using the TYPECHEF infrastructure for data-flow analysis [Lie15], we ignore variability introduced by header files and focus on C source code files.

[4] http://blogs.grammatech.com/visualizing-include-graphs

To validate our information-flow analysis approach with sampling, we proceed as follows. We analyze every code coverage variant for each program from our benchmark set and log every detected information-flow for each variant. This result set we compare to the reported information-flows of our variability-aware analysis approach. We consider our approach as valid, if all information-flows detected by the code coverage sampling heuristic are also reported by our variability-aware approach in the same configuration as for sampling.

Results

During our experiments we found, that our approach does report a correct result: all flows of information detected using the code coverage sampling heuristic are reported as well by our variability-aware solving strategy with a presence-condition satisfiable by the comparable code coverage configuration. Further we observed, that by the use of the code coverage sampling heuristic, we would cover on average 84% of all reported information-flow by the variability-aware approach. For 37% of our program set the sampling based approach reports an identical set of detected information-flows. But on the other hand, we also witnessed programs for which sampling only detected less than 27% of the information-flows reported by the variability-aware solving strategy.

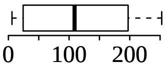
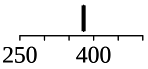
Finally, all information-flows reported by the variability-aware strategy are detected in the affected products too and all information-flows reported in the product-based verification approach are detected by the variability-aware strategy.

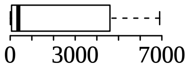
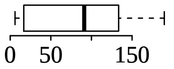
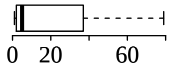
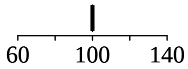
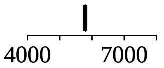
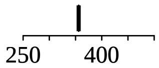
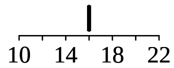
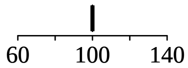
In order to validate the correctness of our approach, 26 programs of MBEDTLS[5] and one program of OPENSSL were analyzed. For MBEDTLS 97 individual source code files were examined because of linking, whereas for OPENSSL 22 different source code files were investigated. Since we did compare our approach with a classical approach of analyzing a specific program variant of a configurable system in isolation, for MBEDTLS a total of 478 variants were analyzed to cover all detected information-flows in different products while for OPENSSL 16 variants were necessary. Finally, our sampling approach of code coverage lead to 348 individual analysis runs for all programs in MBEDTLS and for OPENSSL to 7 different configurations to cover all instructions of our minimal AES program.

6.3.2 RQ2: Efficiency

To answer this research question, we investigate the collected run-time data of the experiments to answer RQ1. This means, we have insights about the run-time for 26 programs of the MBEDTLS case-study and one program of the OPENSSL case-study. Further, we can compare our approach against two benchmarks: (1) a sampling based and (2) an *affected products* based approach. For both benchmarks the general evaluation strategy is

[5] We had to exclude eight problematic programs for which our cross-check strategy ran into a timeout. Nevertheless, until the timeout occurred, no missed information-flow was reported. All presented numbers and plots have been generated after excluding the problematic programs.

Benchmark	**Lifted Solving**	
	time in s	
	mean ± sd	max
mbedTLS	183 ± 251	936
		
OpenSSL	379 ± 0	379
		

Benchmark	**Affected Products**							
	∑ time in s		max time in s		# configurations		% coverage	
	mean ± sd	max	mean ± sd	max	mean ± sd	max	mean ± sd	min
mbedTLS	3903 ± 8140	29625	144 ± 220	912	18 ± 24	79	100 ± 0	100
								
OpenSSL	5791 ± 0	5791	355 ± 0	355	16 ± 0	16	100 ± 0	100
								

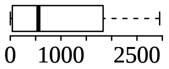
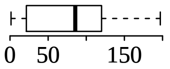
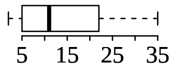
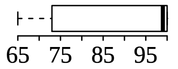
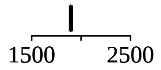
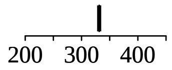
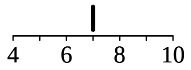
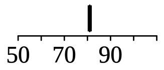
Benchmark	**Code Coverage**							
	∑ time in s		max time in s		# configurations		% coverage	
	mean ± sd	max	mean ± sd	max	mean ± sd	max	mean ± sd	min
mbedTLS	1153 ± 2134	10740	125 ± 191	908	13 ± 10	35	84 ± 28	9
								
OpenSSL	1895 ± 0	1895	331 ± 0	331	7 ± 0	7	81 ± 0	81
								

Table 6.1: Measurement results for different analysis strategies: results include mean (±), standard deviation (sd) and maximum (max) time for performing a solving task in seconds; same for the number of configurations analyzed per program (# configurations) and covered information-flow facts (% coverage) but for that value the minimal (min) coverage rate is shown; box plots show the corresponding distributions (excluding outliers)

the same: for each program included in our experiment set, we have a set of individual configurations. In case of sampling, as previously outlined, these configurations are a minimal set to cover all lexical source code instructions of the analyzed program together with externally linked source code instruction at least once. As for the affected-products based approach, the set of configurations consists of all product variants for which our solving strategy did report an information-flow. This a pretty useful configuration set, because this way we are able to compare the run-time of our solving approach against the run-time of exactly every single affected product. For both configuration sets, we measure the total run-time for all variants of a program and the run-time for analyzing a single program variant.

In Table 6.1 we show our performance measurement results for the individual analysis approaches. Run-time results are shown in total for all configurations as well as single times for separate configurations. For run-time we measured the sole execution time spent for solving the data-flow analysis problem excluding all preparation times, such as AST loading, rewrite operations, etc. All run-times are displayed in seconds as the mean of all analyzed programs along with standard deviation (sd) and maximum (max) time. For the presented solving strategy in this thesis only the single run-time is displayed, since our approach analyzes all variants simultaneously. Same for the number analyzed configurations, they include the mean along with standard deviation (sd) and maximum (max) number of analyzed configurations. The final column of the affected products- and code coverage table shows the coverage rate of detected information-flows by the approach compared to our variability-aware approach. Again, the coverage percentage rate are displayed by the mean and standard deviation (sd), in contrast to the other columns the minimal coverage rate is displayed.

As the table shows, our approach outperforms both approaches clearly compared to the total run-time, since our approach only requires one analysis run to cover all variants of the analyzed configurable software systems. More interestingly is the comparison of our approach against the run-time of a single analysis run of a concrete product variant. For that purpose, we did choose to compare our approach against the most expensive single run-time of an individual product variant. We did choose the most expensive single run-time for the following reasons: first we witnessed, that some configurations of MBEDTLS have surprisingly low run-times in a single-digit range. This is caused by the fact, that depending on the chosen configuration, two different main functions are applicable, one version that actually performs the program's purpose, and another version which only prints out to the console, that the program is incompatible to chosen configuration. Consequently, for some valid configurations of the cryptography library MBEDTLS, only a program consisting of one file with one function and one instruction is analyzed. Another reason, why we did choose the most expensive single run-time, is that this configuration commonly represents the most heavy-weighted program variant. As our approach analyzes all potential variants simultaneously, the most heavy-weighted program variant is included as well at our approach.

Of course, our approach can not beat the single run-time of the most heavy-weighted program, but we observe that our approach does not add significant more time because of additional edges within the exploded supergraph introduced by variable execution paths in the program. At worst, our analysis adds an additional run-time of 48 seconds compared to the most expensive analysis run time, observed at the AES implementation of OpenSSL. However, since analyzing this single product already took 331.3 seconds, the run-time to analyze all variants simultaneously increases only up to 379.0 seconds or in a relative view, the analysis run-time slows down by 14.5%. On the hand, analyzing all 16 affected variants individually in the classical way would have taken a total time of 5791.9 seconds or *96 minutes*. For the program set of mbedTLS, we witness the same effect. In our experience, the increase of run-time introduced by our approach would not be received badly in a real-life analysis setting by an developer.

6.4 Experience with Taint Checking on Cryptography Libaries

For a real-life example application of our developed information-flow analysis, we did choose to taint check the implementation of the AES cipher of our subject systems: mbedTLS and OpenSSL. As previously described (cf. Section 2.2.4), in order to perform a taint check analysis, one must define the interesting taint value or sink depending on the concrete application scenario. In our case, we choose as tainted secret the initial key value used to perform AES encryption or decryption. Fortunately for us, the variable representing the secret AES key in the corresponding implementation base of mbedTLS and OpenSSL, are named `key`. So, in order to taint check the secret key value, we trace the information-flow originating from the first assignment of the variable `key` throughout our example programs to facilitate the AES cipher of our subject cryptography libraries. In a naive way of thinking, one would expect that the key reaches in every variant the actual implementation of the AES cipher algorithm. This reach we use as a very basic sanity check for our information-flow analysis problem formulated for the IFDS-framework.

An important note upfront: our taint check strategy is limited to the application scope of the presented information-flow analysis. As previously described (cf. Section 4.2.3), our analysis is unable to detected the flow of information caused by pointer aliasing. Further, since we do not evaluate arithmetic expressions of control instructions in the C programming language, such as `if` statements, our information-flow analysis is not able to detect any information-flow caused by insufficient memory boundaries checks in the source code. This means, any potential information-leakage caused by buffer overflows is not detected by our approach. That being said, our analysis still is able to detect both obvious and subtle information-flow traps that can be traced via assignments, such as for example a debug instruction to print the taint secret out to the console encoded in a temporary variable.

In detail we present the observed flow of information caused by the AES key within OpenSSL. First of all, we observed that the key definition eventually reaches the C implementation of the AES cipher algorithm in all variants that provide AES. More precisely:

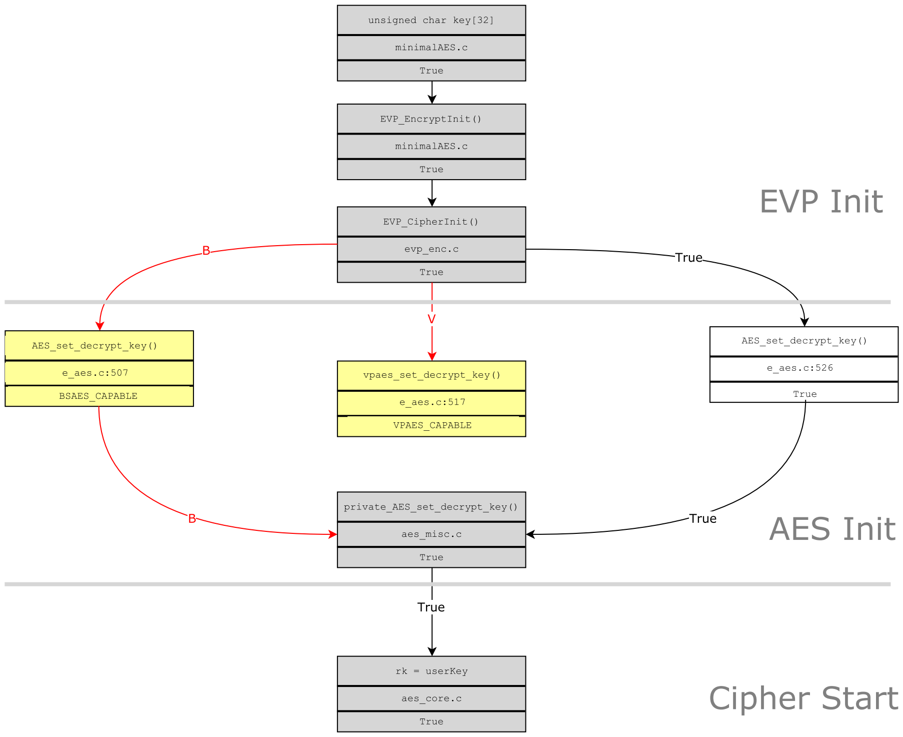


Figure 6.1: Simplified and truncated version of the reported information-flow for OpenSSL

we were able to trace the input key to the point where it is expanded into a number of separate round keys as defined by the underlying Rijndael's key schedule procedure [DR99]. Second, by the means of our analysis strategy, we did not detect any instruction reached by the key value, that we consider as malicious instruction which may cause leakage of the key. But, we did gain an interesting, real-life insight about variability concerning the propagation of the secret key value from its definition to its final destination in the actual implementation of the AES algorithm.

Figure 6.1 shows a simplified and truncated version[6] of the information-flow reported by our analysis strategy of the tainted AES key from its initial definition to the key expansion process in the actual implementation of the AES cipher algorithm. The presented flow-graph encodes variable statements as colored nodes and variable flow edges as colored arrows. We can separate the information-flow of our tainted AES key in three individual phases: (1) initialization of OpenSSL's EVP cryptography interface, (2) initialization of the AES cipher and (3) start of the AES algorithm, namely key expansion. However, only in phase two we witnessed variability affecting the information-flow of the AES key, with no outside effects, since outgoing flows of this phase eventually propagate information of the key further down to the key expansion process in all program variants. In the affected phase, the AES cipher options are set: these options include whether to perform encryption or decryption, the used *mode of operation*, such as cipher block chaining or

[6] The plain version of the information-flow graph is provided on the appendix DVD of this thesis because of its size.

```
#ifdef VPAES_ASM
#define VPAES_CAPABLE   (OPENSSL_ia32cap_P[1]&(1<<(41-32)))
#endif

[...] // some more code
static int aes_init_key () {
    if ((mode == EVP_CIPH_ECB_MODE || mode == EVP_CIPH_CBC_MODE)
        && !enc)
#ifdef BSAES_CAPABLE
        if (BSAES_CAPABLE && mode==EVP_CIPH_CBC_MODE)
            aes_set_decrypt_key(); // aes key init for decryption
        else
#endif
#ifdef VPAES_CAPABLE
        if (VPAES_CAPABLE)
            vpaes_set_decrypt_key(); // aes key init for decryption
        else
#endif
        aes_set_decrypt_key(); // default aes key init for decryption
    else
        // same code from hell again, but for
        // the aes key init for encryption
    return 1;
};
```

Listing 6.1: `#ifdef` - hell for AES key initialization extracted from `e_aes.c` in OpenSSL

counter mode, the used cipher block size, the used key in respect to the set key-length and further additional options. Since we did not observe variational information-flow in phase one and three, we omitted most of these statements in the presented graph for an improved readability.

Interestingly, we found out that the detected variational information-flow during the initialization phase of the AES cipher occurs at source code that can be seen as a textbook example for `#ifdef` - hell. The observed flow does not introduce any malicious sink, but it is a very good example to illustrate some key features of our analysis strategy and to discuss some limitations of static code analysis in general and for the context of analyzing C source code. First of all, the information-flow is variable because of two CPP directives: *VPAES_CAPABLE* (*V*) and *BSAES_CAPABLE* (*B*). Both directives are set by the configuration options *VPAES_ASM* or *BSAES_ASM* respectively, and enable or disable accelerations mechanism of the AES computation task, namely *vector permutations* (VPAES) and *bit slicing* (BSAES). In Listing 6.1 the corresponding source code fragment extracted from the file `e_aes.c` of OpenSSL is shown. As we can see, there are CPP directives nested in various C control statements as *undisciplined annotations*. Further, the used `#ifdef` arguments, *VPAES_CAPABLE* and *BSAES_CAPABLE*, which control wether the annotated source code fragments are included into the product source code or not, are *function-like* macros. These macros are used during run-time as condition of the control statement `if`, to check whether the underlying hardware platform fulfills the requirements of these acceleration mechanism or not. Consequently, these CPP directives alter the program execution path during compilation-time as well as during run-time.

Despite the fact, that this source code fragment easily fulfills the definition of a *code smell* and the *undisciplined annotations* nested in control instructions are non-trivial, our analysis correctly reports the flow of information of the tainted AES key through these statements with the corresponding configuration options for the individual execution paths in different products. In order to analyze the present source code fragment, our analysis is forced to duplicate the fragment into four different variants, because of the undisciplined annotations within control-flow instructions.

At first sight one might be wondering about the missing outgoing edge from the node `vpaes_set_decrypt_key` in the presented information-flow graph. This caused by the fact, that the function call of `vpaes_set_decrypt_key` does not call a function written in C code but in assembler code. We are not able to parse nor analyze assembler code. For such function calls, our analysis strategy ignores the call destination and reports as potential sink all information-flows to the function call parameters. As for the presented example, if configuration option *V* is enabled, there exists an CFG path for the AES key, that will reach this instruction. This a conservative fallback mechanism for incompatible program instructions for our analysis tool. But as explained, the feature constraint *V* is used to control the execution path during run-time as well. So, if *V* is enabled, there also exists a potential execution path that reaches the function call to the plain C implementation of the AES-cipher `aes_set_decrypt_key` in Line 21 of the displayed source code listing. A call to `aes_set_decrypt_key` exists in two distinct locations within the initialization source code of the AES-cipher. Both individual calls are reached under different execution paths and conditions. The call in Line 21 of our source code listing is always reached: in case neither *V* nor *B* are activated, this is the only statement which is included into the output source code fragment. If at least one of the feature constraints is activated, the resulting fall-through path for the variational `if` control statements leads to this function call instruction too. So consequently, our information-flow graph reports validly that this instruction may be reached in every configuration and consequently propagates information about the tainted key further down the execution path. On the other hand, in Line 13 of the source code listing, another call to `aes_set_decrypt_key` is present. This time however, our analysis reports that a flow of the key to this instruction can only exist if *B* is enabled. Note: to interpret the graph correctly, if *B* and *V* are both enabled, all three key set calls can be reached[7].

In contrast to the complex AES cipher initialization process of OpenSSL, we did not observe a similar initialization of the AES key in mbedTLS. First of all, the implementation of mbedTLS for a minimal usage of the AES cipher is very light-weighted compared to OpenSSL: only three files are linked together for such a program, whereas OpenSSL requires a total of 22 different source code files. Likewise to OpenSSL, our information-

[7] In the original graph generated by our information-flow analysis strategy, these call nodes are reported individually since for that kind of variability code duplication is performed.

flow analysis successfully traces the tainted key variable until its expansion into a number of separate round keys in the implementation of the AES algorithm within MBEDTLS. This time however, the reported flow is completely variational. However, this variational information-flow is mostly caused by a design pattern applied in MBEDTLS. As previously outlined, in case the overall configuration of the MBEDTLS cryptography library is not sufficient enough for a given program of the library, the program is executed with a default `main` method, which only prints out to the console, that at least one required feature is missing. Consequently, a minimum of two different execution paths through the program exists, since there are two separate entry points based on the selected configuration options. But still, the information-flow of the tainted key remains variational if we ignore the condition of the starting point. The MBEDTLS cryptography library can be configured for minimal memory footprint. As consequence, if the configuration option *MBEDTLS_AES_ROM_TABLES* is enabled, some static values of the AES algorithm, such as lookup tables, are stored in ROM instead of normally in RAM. This fact results in two different execution paths for the key expansion process.

6.5 Limitations

In this thesis we did successfully adopted the concept of SPL$^{\text{LIFT}}$ to be working on configurable software systems written in the programming language C. To lex and parse such software systems, the parsing infrastructure TYPECHEF is used. While both tools target the analyzation of configurable software systems, they both differ in the way they encode and handle variability. In order to analyze all variants of a configurable software systems simultaneously with an variability-unaware data-flow analysis strategy one limiting assumption must be made: *type uniformity*.

Real-life C configurable software systems often induce different data-types for numeric values. From our own experience, we can report, that for example the LINUX KERNEL uses this concept to support different hardware architectures and word-length. This kind of variability is not directly expressed in the used AST abstraction in TYPECHEF, but detected by the built-in type-check system of TYPECHEF. This system reports the correct type depending on the configuration. Because of the fact, that this variability is independent from the execution path within the program, one can not relay on type-information in a naive, variability-unaware way. But this can be easily resolved, by treating such problems variability-aware in the formulated IFDS data-flow problem, such as data-flow fact duplication. We see this limitation only as minor limitation, since to our knowledge, despite our very own analysis, no further, fully working IFDS data-flow analysis problem for the C programming language exist. Existing IFDS data-flow problems are tailored to be working with other parsing tools, such as for example SOOT for software systems written in Java.

There are numerous case-studies existing for TYPECHEF, but setting up a new case-study to be analyzed with the parsing infrastructure TYPECHEF is a non-trivial task. Currently, the possibly complex setup of a software system (e.g., configuration scripts, library dependencies, and the build-system setup) must converted per hand for TypeChef. We think, that the TypeChef infrastructure would benefit massively from a more simplified setup strategy which does not require special domain knowledge.

7. Related Work

In this thesis, we combined several research fields in order to enable the support of interprocedural data-flow analysis for configurable software systems written in the programming language C. Each individual field is still in focus by academia.

First of all, the problem statement itself has been investigated by researchers. Ferreira et al. did use the TypeChef infrastructure, along with the same function pointer analysis strategy like we did, to investigate the influence of preprocessor variability on the occurrence of vulnerabilities in the source code of the Linux Kernel [FMK+16]. Its key finding, that vulnerable functions have, on average, three times more `#ifdef` directives than non-vulnerable ones, supports our problem statement for the requirement of interprocedural variability-aware code analysis strategies. A similar investigation on the Linux Kernel conducted Abel et al. [ABW14]. They mined the GIT repository of the Linux Kernel for real-life software defects. During their mining process, they found 42 defects which are linked to variability. These defects have been categorized into different categories. As a result, they made the observation that all kinds of defects are affected by variability and variability leads to an increases of the complexity of the found defects. Finally, in the scope of this research field Medeiros et al. conducted an interview study on how developers perceive the CPP in real-life [MKR+15]. Based on these interviews, the authors found that developers are aware of the shortcomings of the usage of the preprocessor in order to express variability. Nevertheless, developers avoid concepts outside of the C language scope to express variability. Consequently, CPP directives remain in practical use for real-life software systems.

As for the research field of static code analysis of configurable systems, Liebig et al. were the first ones to implement variability-aware data-flow analysis in the presence of conditional compilation directives [Lie15, LvRK+13]. Liebig's implementation of the variational control-flow graph made it possible in the first place, to enable the tool combination of TypeChef and SPL$^{\text{LIFT}}$ for interprocedural data-flow analysis on configurable systems. Likewise to our approach, the intraprocedural data-flow analysis techniques by Liebig et al. lack of support for pointer aliasing. In the same scope, Braband et al. [BRT+13] and Bodden et al. [BTR+13] developed analysis strategies for configurable software systems expressed in the programming language Java. Most notably Bodden et al. for their approach of lifting interprocedural IFDS based data-flow problems on configurable software systems in Java. This concept we did migrate in this thesis for variable C source code.

In general, Bodden and his research team are heavily providing mature data-flow analysis strategies expressed within the IFDS-framework. Their contributions to the research community varies from providing a solver-framework for IDE/IFDS-problems [Bod12], to taint analysis strategies for the mobile operating system ANDROID [ARF+14] and the programming language JAVA [LHBM14]. Most recently, Späth et al. did proposed a solution for the missing support of pointer aliasing in IFDS problems [SNAB16]. They presented a flow-, field-, and context-sensitive pointer analysis strategy that is more precise than other analysis strategies for JAVA programs. Further, this approach enabled the speed up of other data-flow analysis tools, such as their own tool FLOWDROID, which are using the IFDS-framework as well. However, to reach this result they did modify the used IFDS solver. But we are confident, that this pointer analysis strategy might be applicable on top of the parsing infrastructure TYPECHEF as well.

Medeiros et al. did compare ten different sampling algorithms for configurable systems [MKR+16]. In their study, they found that with an increasing number of sampling configurations, the fault-detection rate increases and at some point detects all faults. In contrast, simpler sampling sets, such as a configuration with most features enabled, are significant more efficient in a timely manner while being more imprecise in detecting faults.

The work of Nadi et al. addresses the remaining initial burden for setting up a new software system as case study for the TYPECHEF infrastructure [NH13, NBKC14]. They developed a static code analysis strategy which extracts *configuration knowledge* from a configurable software system. Their implemented heuristic is able to extract a very high percentage (> 90 %) of configuration constraints from the source code. Unfortunately, not all configuration constraints are located in the source code. Configuration constraints, which are caused by external dependencies, are only covered by a low percentage. Nevertheless, it is a step towards reducing the very high upfront investment using TYPECHEF.

8. Conclusion

In this thesis, we evaluated the potential of lifting variability-unaware interprocedural data-flow analysis formulated in the IFDS-framework on configurable software systems in C. To do so, we did successfully combine the parsing infrastructure TypeChef with the IFDS solver for configurable software systems SPL$^{\text{LIFT}}$. We learned, that the concept of the variational control-flow graph provided by the TypeChef infrastructure is a sophisticated and mature concept, which helped us to simplify the general lifting process of SPL$^{\text{LIFT}}$ while improving its precision and eliminating existing limitations.

We were not the first ones in the attempt of combining both tools, TypeChef and SPL$^{\text{LIFT}}$, together. To our knowledge, both previous attempts did not succeed. Most likely due to the fact, that they were unaware of the major difference in the concept of the underlying control-flow graphs and the technical challenges (such as the distinct identification between equal instances of nodes within the AST for different source code statements) in the combination process of two independent academic-based static code analysis tools.

To show the efficiency and correctness of this approach, we developed an assignment-based information-flow analysis problem for the data-flow framework IFDS. This *proof-of-concept* analysis we did apply as taint checking approach on the implementation of the AES-cipher in two real-world cryptography libraries: mbedTLS and OpenSSL. While our analysis was not able to detect any information leaks, we were able to show the correctness and efficiency of the adapted lifting technique of Bodden for configurable software systems written in the programming language of C.

Although some minor conceptual limitations remain, we did contribute an feasible and scaleable interproducural data-flow analysis framework for configurable software systems in the C programming language.

9. Acknowledgements

I thank Prof. Sven Apel and Prof. Christian Lengauer for supervising this thesis. Furthermore, I would like to thank the Chair of Software Engineering at the University of Passau for hiring me as a student research assistant since my Bachelor's Thesis and for giving me the opportunity of participating in current research. I want to thank all staff members who helped me with their suggestions and time, especially Dr. Alexander von Rhein and Andreas Stahlbauer for their substantial support and constant guidance.

9.1 Tool Availability

The presented tool combination of TypeChef with SPL$^{\text{LIFT}}$ is available as a subproject of TypeChef, called CSPL$^{\text{LIFT}}$, and can be retrieved at:

`https://github.com/aJanker/CSPLlift`

During the development process we had to modify the parsing infrastructure TypeChef in a custom fork. We are planning to integrate these changes back into the original TypeChef project as these modifications address some general issues. This fork can be found at:

`https://github.com/aJanker/TypeChef`

We included a convenient build script into the project of CSPL$^{\text{LIFT}}$ for an easy setup along with all required dependencies. Both case studies and the used evaluation setup for reproducibility can be downloaded here:

mbedTLS:	`https://github.com/aJanker/CSPLlift-mbedTLS-Analysis`
OpenSSL:	`https://github.com/aJanker/CSPLlift-OpenSSL-Analysis`

Finally, since the C Standard Library is compiler- and platform-specific, the concrete version used in our experiments is documented here:

`https://github.com/aJanker/TypeChef-GNUCHeader`

All mentioned tools, case studies and raw results of the experiments are provided on the disc attached with this thesis.

Bibliography

[ABKS13] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer, 2013. (cited on Page 7)

[ABW14] I. Abal, C. Brabrand, and A. Wąsowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Study. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 421–432. ACM, 2014. (cited on Page 64)

[All70] F. Allen. Control-flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19. ACM, 1970. (cited on Page 11)

[ARF$^+$14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the International Conference on Programming Languages Design and Implementation (PLDI)*, pages 259–269. ACM, 2014. (cited on Page 13, 28, and 65)

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008. (cited on Page 9)

[Bod12] E. Bodden. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *Proceedings of the International Workshop on the State of the Art in Java Program Analysis (SOAP)*, pages 3–8. ACM, 2012. (cited on Page 41, 43, and 65)

[BRT$^+$13] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013. (cited on Page 42 and 64)

[BTR$^+$13] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL$^{\text{LIFT}}$ – Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the International Conference on Programming Languages Design and Implementation (PLDI)*, pages 355–364. ACM, 2013. (cited on Page v, 2, 14, 28, 29, 41, 42, 43, 53, and 64)

[CW07] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, First edition, 2007. (cited on Page 8)

[DD77] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977. (cited on Page 13)

[DKA+14] Z. Durumeric, J. Kasten, D. Adrian, A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The Matter of Heartbleed. In *Proceedings of the International Conference on Internet Measurement (IMC)*, pages 475–488. ACM, 2014. (cited on Page 2)

[DR99] J. Daemen and V. Rijmen. AES Proposal: Rijndael. National Institute of Standards and Technology (NIST), 1999. (cited on Page 59)

[FKF+10] J. Feigenspan, C. Kästner, M. Frisch, R. Dachselt, and S. Apel. Visual Support for Understanding Product Lines. In *Proceedings of the 18th International Conference on Program Comprehension (ICPC)*, pages 34–35. IEEE, 2010. (cited on Page 14)

[FKPA15] G. Ferreira, C. Kästner, J Pfeffer, and S. Apel. Characterizing Complexity of Highly-Configurable Systems with Variational Call Graphs: Analyzing Configuration Options Interactions Complexity in Function Calls. In *Proceedings of the Symposium and Bootcamp on the Science of Security (HotSoS)*, pages 17:1–17:2. ACM, 2015. (cited on Page 16)

[FMK+16] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 65–73. ACM, 2016. (cited on Page 64)

[Hin01] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the Internationl Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61. ACM, 2001. (cited on Page 16)

[HZS+16] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. Preprocessor-based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*, pages 449–482, 2016. (cited on Page 8)

[Jan13] A. Janker. *Implementing Conditional Compilation Preserving Refactorings on C Code*. Bachelor's thesis, Department of Computer Science and Mathematics, University of Passau, Germany, 2013. (cited on Page 6)

[Jon03] J. Jones. Abstract Syntax Tree Implementation Idioms. In *Proceedings of the International Conference on Pattern Languages of Programs (PLOP)*, pages 1–10. Hillside, 2003. (cited on Page 9)

[Käs10] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Magdeburg, Germany, 2010. (cited on Page 14 and 24)

[KBBK10] C. Kim, E. Bodden, D. Batory, and S. Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *Proceedings of the International Conference on Runtime Verification (RV)*, pages 285–299. Springer, 2010. (cited on Page 54)

[KGO11] C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial Preprocessing C Code for Variability Analysis. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 127–136. ACM, 2011. (cited on Page 9)

[KGR$^+$11] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM, 2011. (cited on Page 2 and 9)

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall Software, Second edition, 1988. (cited on Page 5, 6, and 9)

[KSS09] U. Khedker, A. Sanyal, and B. Sathe. *Data-flow Analysis: Theory and Practice*. CRC Press, 2009. (cited on Page 12)

[LAL$^+$10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010. (cited on Page 2)

[Lan92] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992. (cited on Page 16)

[Lap07] P. Laplante. *What Every Engineer Should Know About Software Engineering*. CRC Press, 2007. (cited on Page 14)

[Lat08] C. Lattner. LLVM and Clang: Next Generation Compiler Technology. In *The BSD Conference*, pages 1–2, 2008. (cited on Page 25)

[LHBM14] J. Lerch, B Hermann, E. Bodden, and M. Mezini. FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 98–108. ACM, 2014. (cited on Page 65)

[Lie15] J. Liebig. *Analysis and Transformation of Configurable Systems*. PhD thesis, University of Passau, Germany, 2015. (cited on Page 2, 14, 40, 45, 54, and 64)

[LJG+15] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-Aware Refactoring in the Wild. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 380–391. IEEE, 2015. (cited on Page 2 and 45)

[LKA11] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011. (cited on Page 8 and 24)

[LST+06] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (SIGOPS)*, pages 191–204. ACM, 2006. (cited on Page 8)

[LvRK+13] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013. (cited on Page 2, 8, 21, 45, and 64)

[MKR+15] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 495–518. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. (cited on Page 8 and 64)

[MKR+16] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–654. ACM, 2016. (cited on Page 65)

[MRR04] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering*, 11(1):7–26, 2004. (cited on Page 12 and 16)

[NBKC14] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 140–151. ACM, 2014. (cited on Page 65)

[NH13] S. Nadi and R. Holt. The Linux Kernel: A Case Study of Build System Variability. *Journal of Software: Evolution and Process*, 2013. (cited on Page 65)

[NL11] C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011. (cited on Page 54)

[NMRW02] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 213–228. Springer, 2002. (cited on Page 25)

[NNH15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2015. (cited on Page 1, 8, 9, 11, 12, 14, 27, 29, 40, and 42)

[noa90] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610. 12-1990*, pages 1–84, 1990. (cited on Page 1)

[oT-FAA15] U.S. Department of Transportation - Federal Aviation Administration. *Considerations for Evaluating Safety Engineering Approaches to Software Assurance*, 2015. (cited on Page 1 and 9)

[PPB$^+$15] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *Proceedings of the International Conference on Modularity (MODULARITY)*, pages 81–92. ACM, 2015. (cited on Page 5)

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995. (cited on Page v, 26, and 28)

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the Symposium on Security and Privacy (SSP)*, pages 317–331. IEEE, 2010. (cited on Page 13)

[Sea05] R. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, First edition, 2005. (cited on Page 8)

[SLSA14] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment. In *Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE)*, pages 65–74. ACM, 2014. (cited on Page 7)

[SM03] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. (cited on Page 13)

[SNAB16] J. Späth, L. Nguyen, K. Ali, and E. Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2016. (cited on Page 65)

[SRH96] M. Sagiv, T. Reps, and S. Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, 167(1):131–170, 1996. (cited on Page 41)

[SSF+12] J. Siegmund, N. Siegmund, J. Fruth, S. Kuhlmann, J. Dittmann, and G. Saake. Program Comprehension in Preprocessor-Based Software. In *Proceedings of the International Workshop on Digital Engineering (IWDE)*, pages 517–528. Springer, 2012. (cited on Page 7)

[TLD+11] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-scale System Software. *ACM Operating Systems Review*, 45(3):10–14, 2011. (cited on Page 54)

[vR16] A. von Rhein. *Analysis Strategies for Configurable Systems*. PhD thesis, University of Passau, Germany, 2016. (cited on Page 5 and 45)

[WKE+14] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 213–226, 2014. (cited on Page 9)

[Zha98] S. Zhang. *Practical Pointer Aliasing Analysis*. PhD thesis, Rutgers, The State University of New Jersey, 1998. (cited on Page 16 and 18)

[ZRL96] S. Zhang, B. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step Toward Practical Analyses. *ACM SIGSOFT Software Engineering Notes*, 21(6):81–92, 1996. (cited on Page 16)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 5. Dezember 2016

Andreas Janker