Master’s Thesis
for the course “Applied Computer Science”

Assisted Discovery of Vulnerabilities in Source Code by Analyzing Program Slices

Alwin Maier

Computer Security Group

Bachelor’s and Master’s Theses of the Center for Computational Sciences at the Georg-August-Universität Göttingen
September 30, 2015
Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
✉ +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
✉ www.informatik.uni-goettingen.de

First Supervisor:     Prof. Dr. Konrad Rieck
Second Supervisor:   Dr. Ansgar Kellner
I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, September 30, 2015
Master’s Thesis

Assisted Discovery of Vulnerabilities in Source Code by Analyzing Program Slices

Alwin Maier

September 30, 2015

Supervised by Prof. Dr. Konrad Rieck
University of Göttingen
Computer Security Group
Abstract

Since our daily life is strongly oriented towards the use of computer systems, everyone is affected by security incidents. Consequently, securing computer systems is of everyone’s interest. Many vulnerabilities in source code can be put down to insufficient validation of input data. Not long ago, a method was introduced to support the auditor in finding and fixing such vulnerabilities. In this thesis, we contribute to this method by proposing alternative approaches for improvement.

The idea is based on a combination of unsupervised machine learning and static code analysis. More precisely, missing checks are expressed as anomalous patterns found in conditions. To separate relevant from irrelevant conditions, the scope of data sources or data sinks is determined. Analyzing these scopes, with respect to checks, makes it possible to expose sources/sinks that deviate from normality. Our method differs from the original method by the definition of the scopes and the similarity assessment of sources/sinks which is required to define normality. Interprocedural program slices are used as scopes for sources/sinks instead of intraprocedural lightweight taint analysis. Furthermore, the notion of similar sources/sinks is based on cluster analysis which is used instead of the identification of nearest neighbors.

We illustrate the theoretical advantages of our contributions. Moreover, we are able to demonstrate the new capabilities of our prototype in the evaluation of multiple controlled experiments.
Zusammenfassung

Da unser tägliches Leben stark von der Nutzung von Computersystemen abhängig ist, ist jeder von sicherheitsrelevanten Ereignissen betroffen. Daher ist das Verbes-
sern von Computersystemen für jeden von Interesse. Viele Schwachstellen im Pro-
grammcode können auf unzureichende Überprüfung von Eingabedaten zurückge-
führt werden. Vor kurzem wurde eine Methode vorgestellt, die den Auditor beim
Auffinden und Beheben solcher Schwachstellen unterstützen soll. In dieser Ab-
schlussarbeit wird diese Methode nochmals aufgegriffen und alternative Ansätze
zur Verbesserung vorgestellt.

Die Idee basiert auf unüberwachten maschinellen Lernen und der statischen
Analyse von Programmcode. Dabei werden fehlende Überprüfungen auf irregu-
läre Muster in Bedingungen zurückgeführt. Um relevante Bedingungen von irrele-
vanten zu trennen, wird für eine Datenquelle oder Datensenke zuerst ein Bereich
festgelegt. Die Analyse dieser Bereiche in Hinsicht auf Überprüfungen ermöglicht
es Quellen/Senken aufzudecken, welche von der Norm abweichen. Unsere Metho-
de unterscheidet sich von der ursprünglichen Methode durch die Definition dieser
Bereiche und durch die Beurteilung der Ähnlichkeit von Quellen/Senken, welche
benötigt wird um die Normalität zu definieren. Dazu wird interprozedurales Pro-
gram Slicing anstelle von intraprozeduraler Lightweight Taint Analyse eingesetzt.
Außerdem wird eine Clusteranalyse für die Erkennung ähnlicher Quellen/Senken
anstelle der Bestimmung der nächsten Nachbarn verwendet.

Wir veranschaulichen die theoretischen Vorteile dieser neuen Ansätze und de-
monstrieren die neuen Möglichkeiten unseres Prototyps in der Auswertung meh-
rerer kontrollierter Experimente.
# Contents

1 Introduction ................................................. 3  
    1.1 Overview ............................................... 3  
    1.2 Related Work .......................................... 5  
    1.3 Outline ................................................. 6  

2 Static Analysis ............................................ 9  
    2.1 Robust parsing of source code ......................... 9  
    2.2 Control flow analysis .................................. 11  
    2.3 Data flow analysis ..................................... 13  
    2.4 Dependence analysis .................................. 16  
        2.4.1 Data dependencies ................................. 17  
        2.4.2 Control dependencies ............................. 19  
        2.4.3 Program dependence graph ....................... 26  
    2.5 Program slicing ....................................... 27  
    2.6 JOERN ................................................... 28  

3 Unsupervised machine learning ......................... 31  
    3.1 Features and feature spaces ......................... 31  
    3.2 Supervised and unsupervised machine learning ....... 32  
    3.3 Complete–linkage clustering ......................... 33  
    3.4 Latent semantic analysis ............................. 34  
    3.5 Anomaly detection ..................................... 36  

4 Method ....................................................... 37  
    4.1 CHUCKY ................................................ 37  
    4.2 Improving CHUCKY ..................................... 39  
    4.3 CHUCKY–NG ........................................... 42  
        4.3.1 Source code parsing .............................. 43  
        4.3.2 Neighborhood discovery ......................... 44  
        4.3.3 Checks selection .................................. 48
Acknowledgements

I want to take the opportunity to thank my supervisors who supported me patiently during this time. My special thanks goes to Fabian who helped me with words and deeds at any time. Thanks a lot Fabs!

Last but not least, I want to say thanks to my friends and fellow students who beared my changing moods and gave me good advice.
1 Introduction

Since our daily life is strongly oriented towards the use of computer systems, everyone is affected by security incidents even if the person does not directly use the vulnerable system. Consequently, securing computer systems is of everyone’s interest. The root cause for most insecure systems is located in the underlying code: even the slightest error can result in a security critical malfunction of the whole system. For instance, missing or insufficient validation of input data, incomplete permission control, or incorrect usage of programming interfaces can have severe consequences. Many such vulnerabilities are a result of carelessness of programming and easy to fix once they are discovered. Others are more hidden or distributed in the code and, thus, harder to detect and to eliminate. Since to err is human software will not be perfect from scratch as long as humans write code. Thus, supporting techniques that help to increase the quality of software are in the focus of the current research area. In this sense, we agree with Heelan (2011) that one should keep in mind that supporting means to assist the auditor and not to replace the human component in the auditing process. The prototype developed for this thesis is not just intended to produce theoretical results of our work, but also to be usable and indeed valuable in practice.

Several approaches have been proposed in the past years using static and dynamic analysis or a combination of both to help securing computer systems. A list of static analysis tools is given in Section 1.2. In this thesis, we revisit CHUCKY, an assisting tool for exposing missing checks in source code. The method was originally developed by Yamaguchi et al. (2013) at the University of Göttingen. It utilizes techniques from static analysis and unsupervised machine learning to pinpoint potential missing checks in source code.

1.1 Overview

The basic idea of CHUCKY is to detect missing checks by comparing functions with respect to carefully selected conditions. Potential missing checks are reported to the
actor. Based on the assumption that the majority of functions perform checks correctly, anomaly detection is adapted to detect irregularities with respect to checks in a group of similar functions. We revisit this method motivated by its potential and the possibility for further improvements. We begin by summarizing this method and list its limitations and weaknesses. On this basis, we present our contributions.

CHUCKY starts by determining the neighborhood, i.e. the $k$ “closest” functions, such that each function in the neighborhood employs the same source or sink as the selected target. Based on the neighborhood, anomalies are detected in the last step. CHUCKY proceeds by separating irrelevant checks from relevant ones by performing lightweight taint analysis, starting at each input source or data sink. Subsequently, all “tainted” conditions are split into checks (sub expressions) and processed mathematically to be suitable for anomaly detection.

We will see that this method has some shortcomings. Since the method is based on anomaly detection, a representative neighborhood is essential. However, the size of the neighborhood is unknown and varies for each function. Consequently, choosing the neighborhood by its size is ineffective. Moreover, unique functions have no neighborhood at all and must be detected as such, since CHUCKY cannot be applied in this scenario. As a countermeasure, we propose cluster analysis in combination with latent semantic analysis for the neighborhood discovery to avoid these limitations. Another weakness is the check selection strategy. Since CHUCKY uses intraprocedural taint analysis, checks outside the boundary of functions are missed, even so they are relevant in some cases. Additionally, we show that lightweight tainting is inaccurate in many cases. Therefore, we implement interprocedural program slicing as an alternative to intraprocedural lightweight tainting. A further (technical) restriction of CHUCKY is that the analysis is based on the entity of functions. This is problematic since a source or sink that is employed multiple times in the same function, cannot be analyzed separately. We will teach CHUCKY to work on sources and sinks directly. Our goal is to reuse the idea of this method by providing alternative techniques to tackle its weaknesses while preserving its strengths. In summary, our contributions to this method are the following:

- We show that basing the analysis on sources and sinks directly is beneficial as it allows to handle multiple occurrences of the same source or sink independently. That way, it is possible to identify checks that are only missing for some occurrences. Additionally, each occurrence is used as a potential neighbor.
• We provide the option to detect similar functions on the basis of latent semantic analysis and clustering. We assume to obtain a representative grouping of functions accordingly to their contexts. Additionally, latent semantic analysis is used to learn the meaning of and the relation between API symbols contained in the functions.

• We implement interprocedural program slicing as an alternative check selection technique. Interprocedural analysis enables us to identify checks outside the boundary of the target function. Additionally, program slicing is the cleanest way to identify dependent statements (including conditions).

• We distribute an implementation of our method that is licensed under the GNU General Public License version 3 (GNU GPLv3). We refer to this implementation as CHUCKY-NG.

We evaluate our new implementation in a controlled experiment. We demonstrate the impacts of cluster analysis for neighborhood detection and interprocedural analysis. In summary, CHUCKY-NG is able to detect all missing checks and most of them without false alarms. The detection rate of our new approach is on par with the original method. Some experiments benefit from the new approach, while others disclose new problems.

1.2 Related Work

Since our method combines different techniques it is related to different research areas: Code clone detection, Static code analysis, and program maintenance.

Code clone detection Since programmers often tend to copy code, similar or even identical code fragments (clones) are not rare in source code. Many approaches exist to detect code clones in software. Identifying and eliminating code clones can help to increase the readability and maintainability of software. The well-known tool CCFinder developed by Kamiya et al. (2002) uses a token-based approach, i.e. source code is processed by a lexical analyzer producing a sequence of tokens. Another token-based approach is CP-Miner (Li et al., 2006) which has the pretension to detect more copy-paste code while being fast. Other approaches are based on the structure of code, i.e. using abstract syntax trees as the basis of comparison (e.g. Baxter et al., 1998; Jiang et al., 2007). A different approach to detecting similar frag-
ments was published by Komondoor and Horwitz (2001). They use the program dependence graph and program slicing to find isomorphic sub graphs.

The problem with code clones in our setting is copying of vulnerabilities. Since our method assumes that missing checks are not the normality, checks missing in general cannot be detected.

**Static Program Analysis**  Different supporting tools for finding vulnerabilities in source code are used in practice. Some tools provide an interface to scan for functions like `gets` or `strcpy` that are still used even if they are marked insecure (e.g. Viega et al., 2000; Wheeler, 2011). Tools like Microsoft’s PREfast (Larus et al., 2004) perform local analysis in C/C++ code by traversing the abstract syntax tree accordingly to a set of patterns in search for idioms. By design, such tools are limited to prescribed vulnerabilities. Unknown vulnerabilities to these tools cannot be found.

Tan et al. (2008) infers security rules from source code and detect violations of these specifications. Therefore a set of security check functions must be specified in advance. If one of these check functions is frequently used to protect the same sensitive operation a rule expressing this relation is inferred. A similar approach (Livshits et al., 2009) is based on propagation graphs, i.e. a model of information flow between functions. Nodes are classified in source, sink, and sanitizing nodes. On this basis, information flow specifications are inferred. Other exploit the existence of multiple independent APIs by detection differences in the respective implementations (Srivastava et al., 2011).

Static taint analysis has been successfully used to detect format string vulnerabilities in C source code (e.g. Shankar et al., 2001) or SQL injections and crosssite scripting in web applications and Java programs (e.g. Jovanovic et al., 2006; Livshits and Lam, 2005).

Different from these approaches, **CHUCKY-NG** does not require code annotations or additional information about sensitive code regions or functions.

**1.3 Outline**

This thesis is organized as follows: Chapter 2 deals with techniques from the field of static analysis of source code. This includes different code representations in form of graphs, e.g. abstract syntax trees, control flow graphs, and program dependence graphs, a joint representation of data and control dependencies. By means of these graphs, static analysis techniques for tracking the propagation of data in pro-
grams and extraction of connected fragments from source code are described and implemented. Basics of unsupervised machine learning are considered in Chapter 3. The actual method, revisited in this work, is described in Chapter 4. The original method is described first and our contribution to this method is presented afterwards. The impact of each contribution is closely observed in a theoretical sense. Finally, the improved method is explained in detail. The results are discussed in Chapter 5. To this end, different approaches of the method are evaluated and confronted, subsequently. This work is concluded in Chapter 6 which summarizes our results.
2 Static Analysis

Static program analysis is a software testing mechanism which is performed without running the program. An often forgotten example for static analysis tools that are indirectly used by programs are compilers that conduct basic code analyses, e.g. syntax validation as well as semantic analysis, before translating source code into machine code. Errors found by a compiler in this phase are called compile time errors. Static analysis attempts to find mistakes in programs by examining the underlying code in detail. On the contrary, dynamic program analysis involves the execution of the program where the behavior and the internal state of the program is monitored. An often used technique in dynamic analysis is taint tracking. This work deals with static analysis techniques for detecting vulnerabilities in source code. This includes parsing source code (see Section 2.1). Section 2.2 deals with the control flow in a program, i.e. the execution order of its statements. Section 2.3 gives an introduction to data flow analysis, a general method to determine how generic data is distributed throughout a program. Data flow analysis is then applied to solve two data flow problems that are essential to compute dependencies between statements, i.e. data and control dependencies (see Section 2.4). Both dependence types are used to isolate independent execution threads. More precisely, algorithms for static taint analysis and program slicing benefit from the dependence information. Details about both techniques are given in Section 2.5.

2.1 Robust parsing of source code

The first problem encountered when performing static analysis is parsing the source code. This is even more complicated by two factors: in many cases the source code is incomplete or dependent on external libraries which cannot be resolved due to unavailability or other reasons. Consequently, relying on a working build environment is not a practical option. Subsequently, available tools that can parse code, e.g. compilers for the specific programming language, cannot be reused. To overcome this obstacle building a robust parser is the way to go. Robust
parsers gracefully handle incomplete or unknown code fragments or other irregularities in the code. One way to realize a robust parser is to use the concept of island grammars. Island grammars usually focus on specific language constructs rather than the complete language. These constructs of interest (the "islands") are covered by detailed productions (rules) of the grammar. All unknown or irrelevant constructs are derived to a special terminal symbol called "water". In other words, where a ordinary grammar fails to find the next matching production, an island grammar can be used to avoid this scenario by providing a special production to water. A formal definition of island grammars can be found in Moonen (2001). By the use of a robust parser, we are able to extract as much information as possible from the code. The remaining task is to construct the rules of the grammar. This is a difficult and time consuming task, especially for complex languages as programming languages. Luckily, free software exists, including a robust parser based on an island grammar for C/C++ code (see JOERN in Section 2.6).

The desired result of source code parsing is a structured representation of the syntax, called abstract syntax tree (AST). The inner nodes of the abstract syntax tree of an expression represent operators, the children the operands, and the leaf nodes atomic expressions, i.e. literals or identifiers. See Figure 2.1 for an example. Since operations are not symmetric in general, child nodes are ordered from left to right to preserve the correct order. Abstract syntax trees are a common intermediate code representation. In contrast to parse trees, which capture the sequence of applied productions of the underlying grammar, abstract syntax trees abstract from the details of the syntax. Statements are represented as abstract syntax trees as well. An example is given in Figure 2.2 showing the AST of a generic if–then–else construct. A consecutive sequence of statements can again be grouped by a common root node. In this manner, whole functions can be expressed by a single AST.

Figure 2.1: The abstract syntax tree of the expression \( (\text{len} \geq 8) \ || \ (\text{len} < \text{sizeof(buf)}) \).
2.2 Control flow analysis

Control flow analysis is a static analysis technique to determine the execution order of statements in a program. We only consider control flow between statements and predicates (expressions representing the conditions in conditional statements) within a function. The control flow between functions, often referred to as interprocedural control flow, is a challenging problem, especially for functional programming languages, since the called function is not necessarily known at compile time. The flow of control within a function is controlled by the conditional statements and unconditional jumps of the programming language. It is best expressed as a graph, called the control flow graph (see Allen, 1970).

Beside the statement and predicate nodes, each control flow graph contains two special nodes: the entry node and the exit node. The entry node is the unique entry point which is connected to the first statement (or predicate) of the associated function. Hence, it is the only node without a predecessor. The exit node is reached from all statements upon which execution the function terminates and the control flow is returned to the caller. It is the only node without a successor, but can have multiple predecessors.

In the discipline of compilers nodes contain basic blocks, rather than single statements and predicates. A basic block is a linear sequence of statements having a single entry and exit point. We expand basic blocks such that each node represents a single statement or predicate and use the concept of basic blocks only for visualization purposes.

Formally, the control flow graph (CFG) is a directed graph in which the nodes represent statements or predicates and the edges the flow of control. Each CFG is associated with a unique function. From a graph theoretic perspective the control flow graph is defined as follows:
Definition 2.2.1 (Control flow graph). The control flow graph of a program \( \mathcal{P} \), denoted by \( \text{CFG}_\mathcal{P} \), is a 4-tuple \((V, E, \text{entry}, \text{exit})\) where the pair \((V, E)\) is a directed graph. In addition, each control flow graph must satisfy the following constraints:

- entry, exit \( \in V \) are two special nodes that represent the entry and exit point of \( \mathcal{P} \)
- \( \deg^- (\text{entry}) = 0 \). i.e. entry has no predecessor — entry is the unique entry point of \( \mathcal{P} \)
- \( \deg^+ \text{exit} = 0 \). i.e. exit has no successor — exit is the unique exit point of \( \mathcal{P} \)
- For all nodes \( v \in V \) there exists a path from entry to \( v \)
- For all nodes \( v \in V \) there exists a path from \( v \) to exit

The last two conditions ensure that each node lies on some path from the entry node to the exit node. As described above, the set \( S := V \setminus \{\text{entry}, \text{exit}\} \) is associated with the statements and predicates of \( \mathcal{P} \). Predicates can be distinguished from statements by their outdegree: \( s \in S \) is a predicate if and only if \( \deg^+ (s) > 1 \). Edges represent the flow of control: For two nodes \( s_1, s_2 \in S \) there exists an edge from \( s_1 \) to \( s_2 \) \( ((s_1, s_2) \in E) \) if there exists a possible execution of \( \mathcal{P} \) such that \( s_2 \) is executed immediately after \( s_1 \). Outgoing edges of predicates are labeled either true or false to signal which path is taken upon the evaluation of the predicate.

A specific execution of a program can be expressed by a path in the corresponding CFG as formalized in the following definition:

Definition 2.2.2 (Execution path). Let \( \text{CFG}_\mathcal{P} = (V, E, \text{entry}, \text{exit}) \) be the control flow graph of program \( \mathcal{P} \). An execution path is a path in \( \text{CFG}_\mathcal{P} \) from entry to exit.

By the definition of control flow graphs, there exists at least on execution path. Note that not all execution paths are realizable in reality, because some predicates might always evaluate to true or false independent of the program input. For example, a program with an endless loop might not have a realizable execution path at all.

The control flow graph is an important program representation since it serves as a cornerstone for other representations. Furthermore, it serves as an abstraction layer of the underlying programming language. Like abstract syntax trees, control flow graphs are commonly used as an intermediate source code representation by compilers for code generation.
Control flow graphs can be created from the AST. Each statement in the AST can be converted into a CFG representing only the control flow of this statement. Composite statements are created by recursively connecting the CFGs of its nested statements accordingly to the control flow of the statement. Connecting CFGs is not complicated since each CFG has a single entry and exit point. Unconditional jumps, where the target is not immediately available are patched afterwards.

### 2.3 Data flow analysis

Data flow analysis is a technique for gathering information about how data is distributed throughout a program. The desired result is an assignment of data flow values to each statement (and predicate) of the program that potentially arrives at this program point by considering all possible flows of control. The following introduction to data flow analysis is based on the introduction given by Aho et al. (2002).

The way how data is propagated through a program \( P \), or its control flow graph, is restricted by two different constraints: The first constraint arises from the semantics of the statements, the second constraint is an immediate result from the possible control flows. Let \( \text{CFG}_P = (V, E, \text{entry}, \text{exit}) \) be a control flow graph of \( P \). The semantic of a statement (or predicate) \( s \in V \) is expressed by a transfer function, denoted by \( f_s \). This function transfers the incoming data flow value, denoted by \( \text{in}[s] \), to a outgoing data flow value, denoted by \( \text{out}[s] \). How this function modifies the incoming data flow value in detail is based on the semantics of the statement in the context of the particular data flow problem. Hence, the transfer function for one and the same statement differ for different data flow problems. The transfer of the data flow value of statement \( s \) is formally described by,

\[
\text{out}[s] = f_s(\text{in}[s]).
\]

In many cases, the data flow values are sets. In this case, transfer functions can be expressed as an operation involving killing of set elements, i.e. removing elements from the incoming data flow value, and generating of set elements, i.e. adding elements to the incoming data flow value. That is mathematically expressed as:

\[
f_s(\text{in}[s]) = (\text{in}[s] \setminus \text{kill}[s]) \cup \text{gen}[s],
\]

where \( \text{kill}[s] \) are the data flow value elements killed by \( s \) and \( \text{gen}[s] \) are the data
flow value elements generated by $s$.

The definition of transfer functions can be extended to paths. To do so, the transfer functions of the nodes in the path are chained together. Let $p = \langle s_1, \ldots, s_{n-1}, s_n \rangle$ be a path in $\text{CFG}_P$. The transfer of data along this path is given by

$$F(p) = f_{s_{n-1}} \circ \cdots \circ f_{s_1} (\text{in}[s_1]) = f_{s_{n-1}} (\cdots f_{s_1} (\text{in}[s_1]) \cdots).$$

Given a specific execution of $\mathcal{P}$ assume that $s$ is executed. That is formally expressed as execution path $p = \langle \text{entry}, \cdots, s, \cdots, \text{exit} \rangle$ in $\text{CFG}_P$. The data flow value that flows into $s$ and out of $s$ for this specific execution of $\mathcal{P}$ is given by:

$$\text{in}[s] = F(\langle \text{entry}, \cdots, s \rangle)$$
$$\text{out}[s] = f_s(\text{in}[s]).$$

However, the aim of data flow analysis is to calculate the data flow value that potentially arrives at a statement. More specific, one is interested in the maximal or minimal possible data flow value. Subsequently, each possible execution of $\mathcal{P}$ has to be considered. For statement $s$ let $P_s$ be the set of all paths from entry to $s$. The set

$$V_s := \{ F(p) : p \in P_s \}$$

contains all potential incoming data flow values of $s$. The elements are combined by applying the so called confluence operator $\land$ (or $\lor$) to obtain the accumulated data flow value that potentially flows into $s$:

$$\text{in}[s] = \bigwedge_{x \in V_s} x = \bigwedge_{p \in P_s} F(p) \quad \text{for all } s \in V$$

(MOP)

The solution of (MOP) is called the meet–over–all–path solution. In the presence of loops the direct solution of (MOP) is impossible due to infinite size of $P_s$ for some statements $s$. To overcome this problem it is often possible to obtain the same solution by applying the confluence operator after each statement rather than at the end of each path. This idea yields the following (simultaneous) data flow equations:

$$\text{in}[s] = \bigwedge_{r \in \text{pred}(s)} \text{out}[r]$$

(2.1)

$$\text{out}[s] = f_s(\text{in}[s]).$$

Figure 2.3 shows the interplay between the confluence operator and the transfer

14
function.

The maximum (or minimum) fixed point of (2.1) coincide with the MOP solution if the domain of data flow values together with the confluence operator are semilattice.

**Definition 2.3.1 (Semilattice).** A set $S$ together with a binary operation, the meet $(\wedge)$, is called a meet-semilattice, denoted by $(S, \wedge)$, if for all $x, y, z \in S$ the following holds:

- $x \wedge x = x$ (Idempotency)
- $x \wedge y = y \wedge x$ (Commutativity)
- $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (Associativity)

A semilattice $(S, \wedge)$ is called bounded if it has an identity element $1$, i.e. $x \wedge 1 = x$ for all $x \in S$. An element $0 \in S$ is called zero element if $x \wedge 0 = 0$ for all $x \in S$.

A semilattice with operator $\vee$, $(S, \vee)$, is called join-semilattice.

The distinction between join- and meet-semilattices results from the connection between semilattice and partial orders:

\[
x \leq y \iff x \wedge y = x \quad \text{for meet-semilattices, or}
\]
\[
x \geq y \iff x \vee y = x \quad \text{for join-semilattices.}
\]
Consequently, a partial ordered set \((S, \leq)\) is a meet–semilattice if for each two elements \(x, y \in S\) the greatest lower bound of \(\{x, y\}\) exists. It is bounded if it has a greatest element. That is, there exists an element \(x \in S\) such that for all \(y \in S\), \(y \leq x\). Its smallest element, given that such an element exists, corresponds to the zero element of \((S, \wedge)\). Both definitions of semilattices are equivalent and can be used interchangeable, depending on which one is more convenient in the given setting. In addition to the semilattice property, each transfer function must satisfy the homomorphism or distributive property:

\[
f(x \wedge y) = f(x) \wedge f(y) \quad \text{for all } x, y \in S \text{ and } v \in V.
\]

The lattice–theoretic approach to data flow analysis was first studied by Kildall (1973) and then formalized by Kam and Ullman (1977).

In summary, if the domain of data flow values \(S\) together with the confluence operator \(\wedge\) are a semilattice and all transfer functions are distributive then the solution of (MOP) coincides with the solution of (2.1). In this case the data flow problem is an instance of a **distributive data flow framework**, that is a triple \((S, \wedge, \mathcal{F})\), where \((S, \wedge)\) is a semilattice and \(\mathcal{F}\) is a set of distributive transfer functions. An instance of a data flow analysis framework is formally defined as a tuple \((\text{CFG} = (V, E, \text{entry}, \text{exit}), M)\), where \(M : V \to \mathcal{F}\) is an assignment of transfer functions to nodes of \text{CFG}.

A more general approach of data flow analysis is based on **monotone data flow analysis frameworks** (Kam and Ullman, 1977) where the distributive property of the transfer functions is weakened by requiring only monotonicity of each function. That is, for all transfer functions \(f\) holds \(f(x \wedge y) \leq f(x) \wedge f(y)\). Many data flow problems do not satisfy the distributive property, but are monotone. Unfortunately, the monotonicity property of the transfer functions is not sufficient to guarantee that the solution of (2.1) will coincide with the MOP solution. However, monotone data flow frameworks ensure a conservative estimation of data flow values, that is \(\text{in}[s] \leq \bigwedge_{p \in P_s} F(p)\) for the maximum fixed point solution of \text{in}[s].

### 2.4 Dependence analysis

To determine whether dependencies exist between two statements one can ask the question whether it is safe or not to interchange both statements. If the order in which the statements are executed does not matter no dependency exists. In the
other case, the statements are either data or control dependent. More specific, two statements are data dependent if the rearrangement results in changed semantic. A statement is control dependent on a predicate if the execution of the statement depends on the evaluation of the predicate. In the following, data and control dependencies are formalized on the basis of control flow graphs.

Dependence analysis is important to compiler engineering, especially for program optimization. For more details see Cooper and Torczon (2011); Muchnick (1997).

2.4.1 Data dependencies

Let $CFG_P = (V, E, \text{entry}, \text{exit})$ be the control flow graph of some function $P$. Data dependencies exist between two nodes $v, w \in V$ if the following two conditions are fulfilled:

1. The statement represented by $v$ assigns a value to some variable, say $x$, which is referenced by $w$.
2. There exists at least one path $p = \langle v = v_0, v_1, \ldots, v_n = w \rangle$ such that each $v_i, i = 1, \ldots, n-1$ in $p$ does not overwrite the value of $x$, i.e. $x$ is not redefined.

The calculation of data dependencies are reduced to the calculation of reaching definitions. Reaching definition is a canonical example for data flow analysis. The challenging part is to determine whether a statement defines or references a variable. While this is obvious for simple assignments, the decision whether a function call defines one of its arguments is not trivial. The necessary information is contained in the abstract syntax trees of the statements and predicates. However, the concrete computation depends on the programming language.

We notate a definition of some variable $x$ at node $v \in V$ by the pair $(v, x)$. Note that we are not interested in the actual value of $x$. Let $\text{def}[v]$ be the set of variables defined at $v$ and $\text{use}[v]$ be the set of variables referenced at $v$.

**Definition 2.4.1 (Reaching definition).** Let $CFG = (V, E, \text{entry}, \text{exit})$ be a control flow graph. A definition $(u, x)$ reaches a node $w \in V$ if there exists a path $p = (u, v_0, v_1, \ldots, v_n, w)$ from $u$ to $w$ and $x$ is not reassigned on this path, that is:

$$x \in \text{def}[u],$$

and

$$x \notin \text{def}[v_i] \quad \text{for all } 0 \leq i \leq n.$$
Figure 2.4: The flow of definitions through a node in the control flow graph. Incoming data flow values are first combined by the confluence operator ($\cup$) and then updated by the transfer function.

An assignment, for example, generates a definition, but may also kill prior definitions. Let
\[ \text{gen}[n] := \{(n, x) : x \in \text{def}[n]\} \]
be the set of definitions generated at $n$ and
\[ \text{kill}[n] := \{(m, x) : x \in \text{def}[n] \text{ and } x \in \text{def}[m]\} \]
be the set of definitions killed at $n$. The problem of reaching definitions fits into a distributive data flow analysis framework. The data flow values are sets of definitions, the confluence operator is set union and the transfer function of node $v \in V$ is dependent on $\text{gen}[v]$ and $\text{kill}[v]$:
\[ f_v(x) := (x \setminus \text{kill}[v]) \cup \text{gen}[v]. \]

Figure 2.4 illustrates the flow of definitions analog to the data flow entity in Figure 2.3.

After all reaching definitions are known, data dependencies are obtained by checking an additional condition:

**Definition 2.4.2 (Data dependencies).** A node $n$ is *data dependent* on node $m$ with respect to variable $x$ if

1. there exists a definition of $x$ at $n$ that reaches $m$ and
2. $x \in \text{use}[m]$. 

18
void func1(int x) {
    if (x) {
        foo();
    } else {
        bar();
    }
}
Listing 2.1: Example code that is well structured.

void func2(int x) {
    if (x) {
        foo();
        return;
    }
    bar();
}
Listing 2.2: Example code with a unconditional jump.

The data dependencies inside a function can again be represented by a graph, called the data dependence graph:

**Definition 2.4.3 (Data dependence graph).** Let $\text{CFG}_P = (V_{\text{CFG}}, E_{\text{CFG}}, \text{entry}, \text{exit})$ be the control flow graph of the function $P$ and let $X$ be the set of all valid variable names. The data dependence graph of $P$ is a directed graph $\text{DDG}_P = (V_{\text{DDG}}, E_{\text{DDG}}, \mu)$ where $V_{\text{DDG}} = V_{\text{CFG}} \setminus \{\text{entry}, \text{exit}\}$. For all edges $(v, w) \in E_{\text{DDG}} \subset V_{\text{DDG}} \times V_{\text{DDG}}$ holds that $w$ is data dependent on $v$. The mapping $\mu : E_{\text{DDG}} \rightarrow \mathcal{P}(X)$ maps edges to a set of variables that are involved in the corresponding dependency.

### 2.4.2 Control dependencies

In well structured programs, i.e. programs that do not contain unconditional jumps like `break` or `continue` and `goto` or `return`, control dependencies can simply be obtained by examining the program’s structure. The control dependencies of the code in Listing 2.1 are obvious: the predicate $x$ of the `if`–statement directly controls the execution of the statements containing the calls `foo()` and `bar()`. This is due to the simple structure of the corresponding source code. On the contrary, the slightly modified code shown in Listing 2.2 contains a hidden control dependence between the statement `bar()` and the condition $x$ which is not as obvious as for the well structured code. Note that both programs are equivalent.

In programs that are composed of multiple nested statements and unstructured jumps, determining control dependencies become more and more advanced. Nevertheless, control dependencies are hidden in the control flow. The key to determining control dependencies are to find an alternate description that does not directly rely on the program’s structure. Therefore, the following graph theoretic terms are needed.
**Dominators**

Let $\text{CFG}_P = (V, E, \text{entry}, \text{exit})$ be a control flow graph. The *immediate dominator* of a node $n \in V$ is the closest predecessor $m \in V, m \neq n$, such that every path from entry to $n$ passes through $m$. This means that the statement represented by node $m$ is executed before the statement represented by node $n$ in any possible execution of the program.

The immediate dominator is a binary relation, denoted by $\text{idom}$, over the set of statements and predicates. Starting from the definition of immediate dominators, we define related terms: A node $n \in V$ *strictly dominates* another node $m \in V$ if $n$ is transitively $\text{idom}$–related to $m$. This relation can be expressed by the transitive closure of $\text{idom}$ ($\text{idom}^+$). Further, the reflexive–transitive closure results in the following notion: a node $n \in V$ *dominates* another node $m \in V$ if $n \text{idom}^* m$, i.e. $n$ strictly dominates $m$ or $n = m$. In this case we use the shorter notation $\text{dom}$ instead of $\text{idom}^*$. The dominator relation defines a partial order over the nodes of the control flow graph. The induced mapping $\text{DOM} : V \rightarrow \mathcal{P}(V)$, where $\text{DOM}(v) := \{ u : u \text{ dom } v \}$ defines the dominating set of a node $v \in V$. Similarly, the *immediate post–dominator* of a node $m \in V$ is the nearest successor $n \in V, n \neq m$, such that every path from $m$ to exit passes through $n$. The terms of (strict) post–dominators are defined in the same way as their counterparts. Post–dominators in a control flow graph are dominators in the reversed graph and vice versa.

The nodes of the control flow graph together with edges, representing the $\text{idom}$–relation (or the immediate post–dominator relation), yield a tree which is called the *(post–)dominator–tree*. The entry node dominates every node. Similarly, the exit node post–dominates every node. Hence, entry and exit are the root nodes of the dominator and post–dominator tree, respectively.

With these notions control dependence can be defined as follows:

**Definition 2.4.4** (Control dependence). Let $\text{CFG}_P = (V, E, \text{entry}, \text{exit})$ be the control flow graph of $P$. A node $y \in V$ is (direct) control dependent on a node $x \in V$ if:

1. there exists a path $p$ from $x$ to $y$,
2. $y$ is a post–dominator for all nodes in $p$ except $x$, and
3. $x$ is not post–dominated by $y$.

Note that, with this definition, a node is never control dependent on itself. Further, observe that there must exist a second path branching at $x$ since $y$ is no post–dominator of $x$. Thus, $x$ is a predicate as expected. In other words: $x$ has two
outgoing edges resulting in paths that join at the immediate post–dominator of \( x \). All nodes of both paths, except the start and end node, are control dependent on \( x \).

The dominance frontier of a node is the set of nodes where the dominance of this node ends. It is formally defined as follows:

**Definition 2.4.5 (Dominance Frontier).** Let \( CFG = (V, E, entry, exit) \) be a control flow graph. Let \( v \in V \) be an arbitrary node. The set

\[
DF_{CFG}(v) := \{ c \in V : \exists (n, c) \in E : v \text{ dom } n, v \text{ dom } c \}
\]

is called the **dominance frontier** of \( d \).

The definitions of control dependencies and dominance frontiers are quite similar, as stated by the following theorem:

**Theorem 2.4.1.** Let \( CFG = (V, E, entry, exit) \) be a control flow graph. Let \( CFG^R := (V, E^R := \{ (v, u) : (u, v) \in E \}, exit, entry) \) be the corresponding reversed control flow graph. A node \( v \in V \) is control dependent on a node \( c \in V \) if and only if \( c \in DF_{CFG^R}(v) \).

**Proof.** Let \( v \) be control dependent on \( c \). There exists a path \( p \) from \( c \) to \( v \) in \( CFG \), say \( p = (c, n_1, n_2, \ldots, n_k = v) \). By definition \( v \) post–dominates all \( n_i, 1 \leq i \leq k \), especially a successor of \( c \) (but not \( c \) itself). Consequently, \( v \) dominates a predecessor of \( c \) in \( CFG^R \) and \( v \) does not dominate \( c \). Thus, \( c \in DF_{CFG^R}(v) \).

Let \( c \in DF_{CFG^R}(v) \). First, \( v \text{ dom } c \). Further, there exists indeed a path from \( c \) to \( v \) in \( CFG \). Since there exists an edge \( (n, c) \in E^R \), such that \( v \text{ dom } n \), there exists a path \( p = v, \ldots, n, c \in CFG^R \). Clearly, the reversed path exists in \( CFG \). Since \( v \) dominates \( n, v \) dominates all other nodes in \( p \) between \( v \) and \( n \) (inclusive) as well.

It is often favored to have a (indirect) control dependence between the entry node and all statements and predicates of a function. This effect can be achieved by augmenting the control flow graph:

**Definition 2.4.6 (Augmented control flow graph).** The **augmented control flow graph** of a control flow graph \( CFG = (V_{CFG}, E_{CFG}, entry, exit) \) is defined as \( augCFG := (V_{CFG}, E_{CFG} \cup \{(entry, exit)\}, entry, exit) \).

The additional edge from \( entry \) to \( exit \) can be interpreted as follows: This edge makes the entry post–dominated by \( exit \) rather than the first statement of the function. Hence, the entry node of the function operates as an external condition which controls whether the function is executed or not.
Definition 2.4.7 (Control dependence graph). Let \(\text{CFG}_P = (V_{\text{CFG}}, E_{\text{CFG}}, \text{entry}, \text{exit})\) be the control flow graph of the function \(P\). The control dependence graph of \(P\) is a directed graph \(\text{CDG}_P = (V_{\text{CDG}} := V_{\text{CFG}} \setminus \{\text{exit}\}, E_{\text{CDG}})\) where \(E_{\text{CDG}} := \{(x, y) \in V_{\text{CDG}} \times V_{\text{CDG}} : y\text{ is control dependent on } x\text{ in } \text{augCFG}_P\}\).

The control dependence graph shares the nodes with the control flow graph with the exception that the exit node is dismissed, since it is not control dependent on any other node nor is any node control dependent on the exit node. The entry node is the only node that is not control dependent on another node. In the special case of a well structured function, the corresponding control dependence graph is in fact a tree. In general, control dependence graphs are at least fully connected, but it is possible that nodes are control dependent on more than one node.

Computation of Dominators

Dominators are traditionally computed using the algorithm from Lengauer and Tarjan (1979). Their algorithm has a low asymptotic complexity but its implementation is complex. In addition, Cooper et al. (2001) showed that the low complexity is rather of theoretical relevance. They presented an algorithm based on a forward data flow analysis that is faster in practice. Moreover, it is easy to understand and to implement. In addition, they describe an algorithm for computing dominance frontiers that uses the same data structures developed for the computation of dominators. We show in the following how this problem fits into a data flow analysis framework. To get started, we first express the dominating set of each node as the solution of a meet–over–all–path instance. Let \(P_v\) be the set of all paths from entry to a node \(v\). For a path \(p = (\text{entry} = v_1, v_2, \ldots, v_n, v)\) let \(F(p) := \{v_1, \ldots, v_n, v\}\) be the set of all nodes in \(p\). Now, by the definition of dominators, the MOP solution can be expressed as

\[
\text{DOM}(v) = \bigcap_{p \in P_v} F(p) \quad \text{for all } v \in V.
\]

Before the data flow equations are presented, each component of the framework, i.e. the domain of data flow values, the meet operation and the transfer functions, are listed. The data flow values are the dominating sets, i.e. sets of nodes. We will refer to the nodes by their reverse postorder number. Let \(\text{ord}(v)\) denote the reversed postorder number of node \(v\). Furthermore, observe that each dominating set contains the entry node. The postorder number of the entry node is always 1.
(ord(entry) = 1). In short, the domain $S$ of the framework is formally given by

$$S = \{x \in \mathcal{P}(\mathbb{N}) : 1 \in x\}.$$  

The meet operation is set intersection, as in the MOP solution. The identity element of the corresponding semilattice are the natural numbers $\mathbb{N}$ and the zero element is \{ord(entry)\} = \{1\}. The transfer function of node $v \in V \setminus \{\text{entry}\}$ is given by $f_v(x) = x \cup \{v\}$. The initial input value is the zero element, i.e. \{1\}. Since $f_v(x \cap y) = (x \cap y) \cup \{v\} = (x \cup \{v\}) \cap (y \cup \{v\}) = f_v(x) \cap f_v(y)$ for all $v \in V$ the transfer functions are distributive and the MOP solution coincides with the minimal fixed point of

$$
\begin{align*}
\text{DOM(entry)} &= \{\text{entry}\}, \\
\text{DOM}(n) &= \bigcap_{p \in \text{pred}(n)} \text{DOM}(p) \cup \{n\}.
\end{align*}
$$  

(2.2)

The data flow equations (2.2) can be solved by an arbitrary data flow solver. However, the structure of dominators permits a significant performance increase when using a specialized algorithm. First observe that all dominators of each node are uniquely defined by their immediate dominators:

$$
\text{DOM}(n) = \{n\} \cup \text{IDOM}(n) \cup \text{IDOM}^2(n) \cup \cdots \cup \{\text{entry}\}
$$

for all nodes $n \in V$. This fact makes the dominator tree a compact data structure for storing dominating sets. Moreover, the dominator tree is best suited for performing the intersect operation. Therefore the following intuitive definition is useful.

**Definition 2.4.8 (Common dominator).** Let $\text{CFG} = (V, E, \text{entry}, \text{exit})$ be a control flow graph. The common dominator of two nodes is the nearest node that dominates both of them. More formally: The mapping $\text{CDOM} : V \times V \to V$, where

$$(a, b) \mapsto c \iff \left( (\text{idom } a) \land (\text{dom } b) \right) \text{ or } \left( (\text{dom } a) \lor (\text{idom } b) \right)$$

maps two nodes to their common dominator.

The definition of common dominators can be easily extended to more than two nodes. Using this notion the data flow equation 2.2 can be rewritten as:

$$
\begin{align*}
\text{DOM(entry)} &= \{\text{entry}\}, \\
\text{DOM}(n) &= \text{DOM} \left( \bigcap_{p \in \text{pred}(n)} \text{CDOM}(p) \right) \cup \{n\}.
\end{align*}
$$  

(2.3)
The dominator tree is stored as key–value pairs \((a, b)\) which are interpreted as \(b idom a\). In this representation the immediate dominator of a node can be retrieved in constant time. The common dominator of nodes \(a, b\) is the root of the smallest subtree which contains \(a, b\). Observe that \(\text{ord}(v) > \text{ord}(\text{IDOM}(v))\) for all nodes \(n \in V \setminus \{\text{entry}\}\). Algorithm 1 shows the computation of the common dominator operation.

### Algorithm 1 Common Dominator

1: procedure COMMON–DOMINATOR\((a, b)\)
2:     \(\text{finger1} \leftarrow a\) \(\triangleright \text{Place first finger at node } a\)
3:     \(\text{finger2} \leftarrow b\) \(\triangleright \text{Place second finger at node } b\)
4:     while \(\text{finger1} \neq \text{finger2}\) do
5:         while \(\text{ord}(\text{finger1}) < \text{ord}(\text{finger2})\) do
6:             \(\text{finger1} \leftarrow \text{IDOM}(\text{finger1})\) \(\triangleright \text{Traverse tree upwards}\)
7:         end while
8:         while \(\text{ord}(\text{finger2}) < \text{ord}(\text{finger1})\) do
9:             \(\text{finger2} \leftarrow \text{IDOM}(\text{finger2})\) \(\triangleright \text{Traverse tree upwards}\)
10:        end while
11:     end while
12: return \(\text{finger1}\) \(\triangleright \text{Both finger agree}\)
13: end procedure

The data flow equations (2.3) are solved by an iterative approach. The nodes are iterated in reversed postorder which is a typical iteration order for forward data flow problems. The pseudocode is given in Algorithm 2. Since the nodes are traversed in reversed postorder the dominator tree is created top–down. Hence, the immediate dominators required to compute the common dominator are already known and, thus, Algorithm 1 can be used in line 8 of Algorithm 2.

The dominating set \(\text{DOM}(v)\) of node \(v\) is implicitly given by returning all nodes on the (unique) path from \(v\) to the root of the dominator tree.

**Computation of Dominance frontiers**

The algorithm to compute dominance frontiers makes advantage of the following fact: Each node \(c\) that is contained in the dominance frontier of node \(v\) has at least two successors. In terms of reversed control flow graphs \(c\) is a predicate node. For that reason, the algorithm starts at nodes with more than one successor and identifies nodes whose dominance frontiers contain this node. To this end, the algorithm iterates over all nodes \(c\) with \(\text{deg}(c) > 1\) and adds \(c\) to the dominance
Algorithm 2 Immediate Dominators

1: procedure IMMEDIATE–DOMINATORS($\text{CFG} = (V, E, \text{entry}, \text{exit})$)
2:     for $n \in V$ do
3:         $\text{IDOM}(n) \leftarrow \text{undefined}$
4:     end for
5:     $\text{IDOM}(\text{entry}) = \text{entry}$
6:     repeat
7:         for $n \in V \setminus \{\text{entry}\}$ in reversed postorder do
8:             $\text{IDOM}(n) \leftarrow \text{CDOM}_{p \in \text{pred}(n)}(p)$
9:         end for
10:     until nothing changed
11: end procedure

Computation of Control dependencies

We now summarize how the dominance information of a control flow graph is used to compute control dependencies. Let $\text{CFG} = (V, E, \text{entry}, \text{exit})$ be a control flow graph. By Theorem 2.4.1 control dependencies are given by the dominance frontiers of the reversed augmented control flow graph $\text{augCFG}^R$. To this end, the
dominator tree of $\text{augCFG}^R$ is calculated, which is the post–dominator tree of the augmented control flow graph $\text{augCFG}$.

The following is a step by step instruction how control dependencies are determined:

1. Start with the control flow graph $\text{CFG}_P$ of program $P$.

2. Create the post–dominator tree $T$ of the augmented control flow graph $\text{augCFG}_P$
   a) Create the augmented control flow graph $\text{augCFG}_P$ of $\text{CFG}_P$.
   b) Create the reversed augmented control flow graph $\text{augCFG}_P^R$ by reversing $\text{augCFG}_P$.
   c) Calculate the post–dominator tree $T$ of $\text{augCFG}_P$ by using Algorithm 2 with $\text{augCFG}_P^R$ as input

3. Calculate the dominance frontiers of each node by using Algorithm 3 with $\text{augCFG}_P^R$ and $T$ as input.

4. By Theorem 2.4.1, statement $v$ is control dependent on predicate $c$ of $P$, if $c \in \text{DF}_{\text{augCFG}_P^R}(v)$.

2.4.3 Program dependence graph

The program dependence graph of a function $P$ contains all dependencies inside this function. It is basically a joint representation of the data dependence graph and the control dependence graph of $P$.

Definition 2.4.9 (Program dependence graph). Let $\text{DDG}_P = (V_{\text{DDG}}, E_{\text{DDG}}, \mu)$ be the data dependence graph and $\text{CDG}_P = (V_{\text{CDG}}, E_{\text{CDG}})$ be the control dependence graph of a function or procedure $P$. The program dependence graph of $P$ is a directed, edge–labeled graph

$$\text{PDG} = (V_{\text{PDG}} = V_{\text{DDG}} \cup V_{\text{CDG}}, E_{\text{PDG}}, \mu),$$

where

$$E_{\text{PDG}} = \bigcup_{(u,v) \in E_{\text{DDG}}} (u, v, \text{DDG}) \cup \bigcup_{(u,v) \in E_{\text{CDG}}} (u, v, \text{CDG})$$

is the set of labeled edges, accordingly to the data and control dependence graphs.
2.5 Program slicing

Program slicing is a technique that can be used to reduce a program to a certain point of interest. Program slicing was introduced by Weiser (1981). He proposed program slicing to decompose programs in several pieces (slices), each of them producing a certain behavior identical to the original program. Weiser lists debugging and parallel processing of programs, e.g. parallel compilation or execution, as potential applications. Since then many other uses of program slicing have arisen from the generic form of slices. Today, slicing is applied in many different fields: debugging and program analysis (e.g. Weiser, 1981; Choi et al., 1991; Fritzson et al., 1992), software maintenance (e.g. Gallagher and Lyle, 1991), reverse engineering (e.g. Beck and Eichmann, 1993; Jackson and Rollins, 1994). An extended and detailed list of applications can be found in Tip (1995). Since slices can be computed automatically, program slicing is well suited to extract a specific part in a complex program.

Formally, program slicing solves the following question: Given a program point \( s \), e.g. a statement and a set of variables \( X \). Which are the statements and predicates that might affect the value of one of the variables from \( X \) at point \( s \)? The pair \( (s, X) \) is called slicing criterion. Program slicing can be classified into two variants. The problem stated above is called backward slicing, since it seeks for statements that affect the values of variables. Hence, the statements are executed prior the program point \( s \). Sometimes it is required to determine the statements that might be affected by one of the values of variables from \( X \) at \( s \). This problem is called forward slicing. We restrict the computation of slices to a single variable, i.e. the slicing criterion consists of a statement \( s \) and a distinct variable \( x \).

To compute slices, knowledge about the dependencies of statements and predicates comes in handy. As proposed by Horwitz et al. (1990) we use the program dependence graph to compute intraprocedural slices, i.e. slices that end at function boundaries. In this case slicing is modeled as a reachability problem in the dependence graphs. When computing interprocedural slices we slightly deviate from the method presented by Horwitz et al. (1990). We use a heuristic in which we assume that the return value of a function call is (indirectly) defined by all of its arguments.
With this heuristic we adapted the two phases used by Horwitz et al. (1990) to compute interprocedural program slices. Static taint analysis can be performed analog to the program slicing algorithm by excluding control dependence edges, i.e. by operating exclusively on the data dependence graph.

2.6 JOERN

JOERN is a platform for static analysis of C/C++ source code. It includes a robust parser driven by an island grammar. Furthermore, JOERN computes the control flow graph and program dependence graph for each function contained in the source code. All code representations, namely the abstract syntax tree, control flow graph, and dependence graphs are stored in a property graph of a NEO4J graph database. Additionally, JOERN provides tools and so-called traversal to navigate through the code. Property graphs are an often used data structure for graph databases. The use of property graphs for information storage is described in Rodriguez and Neubauer (2010). They can be defined as:

Definition 2.6.1 (Property graph). A property graph is a directed, edge-labeled, attributed multigraph, where sets of properties are attached to nodes and edges. Formally, a property graph is a 3–tuple $G = (V, E, \mu)$, where

- $V$ is a set of nodes,
- $E \subseteq V \times V \times \Sigma$ is a set of directed edges, where an edge $(v, w, \sigma) \in E$ connecting nodes $v$ and $w$ is typed by a label $\sigma \in \Sigma$
- $\mu: (V \cup E) \rightarrow \mathcal{P}(K \times L)$ is a function mapping nodes and edges to sets of attributes. Attributes are key–value pairs $(k, l) \in K \times L$.

The property graph generated by JOERN is called (interprocedural) code property graph (see Yamaguchi et al., 2014, 2015). Before we formalize code property graphs, the code representations described in the previous sections must be modified to fit the definition of property graphs. The abstract syntax tree of a function is extended by attaching attributes to the nodes. Those attributes include a code property for statements and expressions. For instance, each node of the AST in Figure 2.1 contains the code value. Moreover, each node has a type property accordingly to the code construct represented by the node. Further, a statement property is introduced to mark nodes representing statements or predicates, i.e. nodes shared with the control flow graph (and dependence graphs). The root node represents
the function, its children are the return type, the parameter list and the function body, which is divided into statements. Formally, an abstract syntax tree is a triple $\text{AST} = (V_{\text{AST}}, E_{\text{AST}}, \lambda_{\text{AST}})$, where $V_{\text{AST}}$ is the set of nodes, $E_{\text{AST}}$ edges representing the parent child relation, and $\lambda_{\text{AST}}$ a mapping that assigns attributes to nodes. Informally, code property graphs consist of the nodes of the AST. Nodes representing statements or predicates are connected by control flow edges as well as dependence edges. The formal definition is stated below.

**Definition 2.6.2 (Code property graph).** Let $\mathcal{P}$ be some function. The code property graph of $\mathcal{P}$ contains the following code representations in a joint property graph:

- $\text{AST}_\mathcal{P} = (V_{\text{AST}}, E_{\text{AST}}, \lambda_{\text{AST}})$, the abstract syntax tree of $\mathcal{P}$, where $\lambda_{\text{AST}}$ maps the nodes in $V_{\text{AST}}$ to their properties,

- $\text{CFG}_\mathcal{P} = (V_{\text{CDG}}, E_{\text{CFG}}, \text{entry}, \text{exit}, \lambda_{\text{CFG}})$, the control flow graph of $\mathcal{P}$, where $\lambda_{\text{CFG}}$ maps the edges in $E_{\text{CFG}}$ to a label property in $\{\text{true}, \text{false}, \epsilon\}$,

- $\text{PDG}_\mathcal{P} = (V_{\text{PDG}}, E_{\text{PDG}}, \lambda_{\text{PDG}})$ the program dependence graph of $\mathcal{P}$.

Let $V := V_{\text{AST}} \cup \{\text{entry}, \text{exit}\}$, $E := \{(v, w, \text{AST}) : (v, w) \in E_{\text{AST}}\} \cup \{(v, w, \text{CFG}) : (v, w) \in E_{\text{CFG}}\} \cup E_{\text{PDG}}$, and $\lambda: V \cup E \rightarrow \mathcal{P}(K \times L)$, with

\[
v \in V \mapsto \begin{cases} 
\lambda_{\text{AST}}(v) & \text{if } v \in V_{\text{AST}}, \\
\emptyset & \text{otherwise}
\end{cases}
\]

and

\[
e \in E \mapsto \begin{cases} 
\{(\text{LABEL}, \lambda_{\text{CFG}}(e))\} & \text{if } e \in E_{\text{CFG}}, \\
\bigcup_{x \in \lambda_{\text{DDG}}(e)} \{(\text{VAR}, x)\} & \text{if } e \in E_{\text{DDG}}, \\
\emptyset & \text{otherwise.}
\end{cases}
\]

The property graph $(V, E, \lambda)$ is called code property graph of $\mathcal{P}$, denoted by $\text{CPG}_\mathcal{P}$.

The auditor can formulate rules to traverse the code property graph using the edge types and properties of nodes and edges as basis for the decision which path should be taken. Since the main information is contained at the nodes, traversal start at a set of nodes and results in a set of nodes. Such traversals can be formally defined as:

**Definition 2.6.3 (Traversal).** Let $\text{CPG} = (V, E, \lambda)$ be a code property graph. A traversal is a function $T: \mathcal{P}(V) \rightarrow \mathcal{P}(V)$ that maps a set of nodes to another set of nodes accordingly to $\text{CPG}$.
The composition of two or more traversals yield another traversal. This is useful, since complex traversals can be created by combining known traversals. Basic traversals for generic property graphs are listed below:

- **IN** and **OUT** traverse from a set of nodes to all nodes reachable by a single step in direction or against the direction of the connecting edge, respectively. Related traversals are **IN\textsubscript{t}** which follow only edges of type \( t \) and **IN\textsubscript{\textbf{k,l}}\textsuperscript{t}** which additionally requires that the key–value pair \((k, l)\) is attached to the incoming edge. Traversals for outgoing edges are defined analog.

- **FILTER\textsubscript{p}** returns all nodes matching the predicate \( p \).

Examples of traversals provided by JOERN for code property graphs are:

- **CHILDREN** and **PARENT** which traverse from an AST node to its children and parent, respectively.

- **STATEMENT** which traverses from an AST node to the enclosing statement, i.e. traversing the tree upwards by repeatedly using the **PARENT** traversal until a statement node is reached.

The interprocedural code property graph is created by linking the code property graphs of together, i.e. connecting arguments with parameters and function calls with the root node of the called function’s AST.

Performing taint analysis or computing slices with JOERN is a great relief and as easy as defining a corresponding traversal.
3 Unsupervised machine learning

Machine learning is a sub domain of computer science influenced by many other fields like biology, neurology, and physics. In the broadest sense, machine learning algorithms are intended to learn dependencies or structures in data. The learned dependencies are then generalized in model which is used to make predictions on unseen data. This chapter gives a brief overview of terms related to machine learning and the algorithms used in our method. We begin by introducing features and feature spaces, used to describe raw objects, e.g. elements of source code, mathematically. Our method uses two different feature spaces, each representing a different aspect of source code: One to identify similar patterns and the other to express conditions found in a specific slice of code. We proceed by presenting the general concepts of machine learning in Section 3.2. Subsequently, three unsupervised machine learning algorithms required for the understanding of CHUCKY-NG are explained. Latent semantic analysis in combination with cluster analysis is adapted by our method to find similarities in source code. They are discussed in Section 3.3 and 3.4, respectively. Anomaly detection, the key component of our method, is used to expose missing checks by detecting deviations from an expected behavior. It is explained in Section 3.5.

For the interested reader, we refer to further literature. Many good books are available for detailed introduction to machine learning (e.g. Bishop, 2006; Duda et al., 2012).

3.1 Features and feature spaces

Many machine learning algorithms operate on geometrical data and are defined over a vector space. Since most data objects are non-numeric in their natural representation, they have to be transformed first. This transformation can either be implicit or explicit. Implicit transformations are based on kernel functions which measure the similarity between raw objects. Kernel functions induce a scalar product in some vector space. This property can be exploited to calculate distances be-
between objects without explicitly mapping the objects into this space. This is known as the *kernel trick*. For the explicit mapping, a set of features, denoted by $\mathcal{F}$, is defined first. Subsequently, the raw data objects are represented by these features. Therefore, each object is *embedded* into the vector space spanned by $\mathcal{F}$ by using a feature map:

$$\phi: \mathcal{X} \to \mathbb{R}^{|\mathcal{F}|},$$

where $\mathcal{X}$ is the input domain of the learning setting. The expressiveness of features is important for effective learning. The feature set and the extraction of features depends on the specific application.

### 3.2 Supervised and unsupervised machine learning

In general, learning can be divided into two different problem settings. In the *supervised learning* setting, labels ($y \in \mathcal{Y}$) are available for each data point. The learning algorithm tries to infer dependencies between the data points and their labels. The dependencies are expressed by model. Based on this model, predictions are made on unseen data. Mathematically, this can be expressed as follows: The learning algorithm implements a *learning function*

$$g: \mathcal{P}(\mathcal{X} \times \mathcal{Y}) \to \Theta,$$

where $\Theta$ is the space of models and $\mathcal{P}$ denotes the power set. The learning algorithm searches this space and returns a (hopefully expressive) model. In contrast, no labels exist in the *unsupervised learning* setting. The learning algorithm tries to generalize the structure of the data, e.g. by extracting clusters or groupings from the data. The extracted structure is again compressed into a model. In this case, the learning function can be described by

$$g: \mathcal{P}(\mathcal{X}) \to \Theta.$$

The learned model $\theta \in \Theta$ parameterizes a *prediction function*

$$f_\theta: \mathcal{X} \to \mathcal{Y}$$

which is used to make predictions on unseen data on the basis of the learned model $\theta$. The terms and notations are taken from Rieck (2009).

The data, i.e. the source code, available to our method is assumed to be unla-
beled, such that our method can be applied on any code without requiring additional information. In other words, we will focus on unsupervised learning.

### 3.3 Complete–linkage clustering

Clustering is an unsupervised learning method used for exclusive classification or grouping of finite sets of data. The general result of clustering is a partition of the data set $X$. Complete–linkage clustering is an instance of agglomerative hierarchical clustering. We give a brief overview of this class of clustering methods. A more detailed introduction to hierarchical clustering and cluster analysis in general can be found in Jain and Dubes (1988) and Duda et al. (2012).

Hierarchical clustering is a common method often used in cluster analysis. As the name suggests, hierarchical clustering seeks for a hierarchy of clusters rather than computing disjoint or flat clusters. This means that each cluster can have subclusters of its own. We distinguish two strategies for computing a hierarchy of clusters: agglomerative (or bottom-up) and diverse (top–down). In case of agglomerative clustering each data point of a data set starts in a cluster of its own. On the contrary, the diverse strategy starts with a single conjoint cluster embracing the whole data set. While diversive algorithms recursively split one cluster in each step, agglomerative algorithms link two clusters in each step. Both strategies yield a tree diagram, called dendrogram. The dendrogram is an intuitive way to represent a hierarchy of clusters. In each iteration an additional level is added to the dendrogram, representing a new partition of the data. In other words, a hierarchical clustering is a sequence of partitions in which two successive members differ only in the fusion of two elements.

Multiple agglomerative clustering algorithm exist. Each of them is specialized by a linkage function, which defines the similarity of clusters. One example is complete–linkage clustering. In this case, cluster $X$ and $Y$ are merged if the maximal distance between two points $x \in X$ and $y \in Y$ is minimal over the set of all clusters. Mathematically, this can be expressed as follows: Let $d$ be some distance function, e.g. the euclidean distance. The complete linkage function is defined as follows:

$$D(X, Y) := \max_{(x, y) \in X \times Y} d(x, y).$$

Linkage clustering can be easily implemented by using a proximity matrix $P = (p_{XY})$, where rows and columns correspond to clusters and $p_{XY} = D(X, Y)$. In
each step the minimal entry is located and the clusters corresponding to the row and column are merged. The proximity matrix is updated by removing the row and column of the merged clusters and adding a row and column representing the new cluster with entries accordingly to the used linkage function. This simple algorithm has time complexity of $O(n^3)$. More clever algorithms exist that perform complete–linkage clustering in time complexity of $O(n^2)$ (see Defays, 1977).

Afterwards, the hierarchy of clusters can be flattened by different criterions, i.e. rules how to prune the dendrogram. For example, the distance criterion forms flat clusters such that each two points within the same cluster have no greater distance than a given threshold.

### 3.4 Latent semantic analysis

Latent semantic analysis (LSA) is a method used for comparing text documents. It was originally developed for the field of information retrieval by Deerwester et al. (1990) where it is known as latent semantic indexing (LSI). The main problem addressed in the domain of information retrieval is the selection of documents semantically related to a given query. LSA arises from the observation that choosing documents on the basis of plain word matching is insufficient since the same semantic can be expressed by multiple words. Based on the observation that the meaning of text can be expressed by few latent variables, called concepts, LSA tries to find the most relevant ones. Subsequently, by the use of LSA, each document is represented by a set of concepts. The induced vector space is called the semantic space. Pairs of documents are compared by computing the similarity of their vectors. An often used similarity measure in this context is the cosine similarity. In summary, LSA extends the vector–based representation of text by expressing each document by sets of concepts rather than raw bag of words to make information retrieval systems robust against synonyms. Moreover, the dimensionality is reduced implicitly. In the following, we describe the theoretical foundation of LSA and explain how LSA is realized.

The association between documents and terms is expressed as a (weighted) term–document matrix. Rows in this matrix correspond to terms and columns to documents. Each column contains the vector–based representation of the corresponding document. Since the same situation can be expressed with different words, similar documents can look quite different. LSA transforms each document vector into a space of lower dimensionality spanned by the concepts. To this end, LSA seeks
a low–rank approximation of the term–document matrix. This approximation is based on singular value decomposition (SVD), a technique from linear algebra: Let $M \in \mathbb{R}^{m \times n}$ be the term–document matrix and $r \leq \min(m, n)$ be the rank of $M$. The matrix $M$ can be decomposed into two matrices $U \in \mathbb{R}^{m \times r}$ and $V \in \mathbb{R}^{n \times r}$ with orthogonal columns and a diagonal matrix $\Sigma \in \mathbb{R}^{r \times r}$ containing the positive and ordered singular values of $M$, such that

$$M = U \Sigma V^T.$$  

Singular values are closely related to eigenvalues: The columns in $U$ and $V$ correspond to the eigenvectors of $MM^T$ and $M^TM$, respectively. They are ordered by dominance, i.e. the first column corresponds to the most dominant direction in the respective vectorspace. The matrix $MM^T$ is called the term–to–term correlation matrix, since each entry expresses the correlation of two terms over all documents. Similarly, the matrix $M^TM$ is called document–to–document correlation matrix. Both correlation matrices share the same eigenvalues and the singular values of $M$ are the square roots of the corresponding eigenvalues. The decomposition of the term–document matrix allows to find an approximation $M_k$ to $M$ of rank $k < r$ by applying principal component analysis. That is, setting the smallest singular values to zero and retaining only the $k$ largest. This approximation is optimal in terms of the least squares error. Since the lower diagonal is set to zero, only columns of $U$ and $V$ corresponding to the $k$ largest singular values contribute to the approximation. Hence, all other columns can be removed, i.e.:

$$M_k = U_k \Sigma_k V_k^T \in \mathbb{R}^{m \times n},$$  

where $U_k$ and $V_k$ contain the first $k$ columns of $U$ and $V$, respectively. The correlation of two documents is independent of $U_k$

$$M_k^T M_k = (U_k \Sigma_k V_k^T)^T (U_k \Sigma_k V_k^T)$$  

$$= V_k \Sigma_k U_k^T U_k \Sigma_k V_k^T$$  

$$= (V_k \Sigma_k)(V_k \Sigma_k)^T.$$  

As a results, documents can be compared in a lower dimensional space. Furthermore, each document can be explicitly mapped into this space by representing documents as linear combinations of their terms accordingly to the $k$ most dominant
directions in the term–to–term correlation space, i.e. the columns of $U_k$:

$$U_k^\top M_k = U_k^\top U_k \Sigma_k V_k^\top = \Sigma_k V_k^\top = V_k \Sigma_k.$$ 

LSA is often used in combination with clustering to group documents by topics, known as document clustering. We will use LSA in the same way to cluster functions by their source code. More details of LSA can be found in Wiemer-Hastings (2004); Dumais (2004); Landauer (2006). An introduction to information retrieval in general is given by Baeza-Yates et al. (1999).

### 3.5 Anomaly detection

Anomaly detection is an unsupervised machine learning technique that is used to expose data objects which do not conform to an expected pattern. The expected pattern is often expressed by model of normality. One natural and simple way to expose anomalies is based on geometry. On that account, each object must be mapped into a feature space first. Subsequently, objects that are assumed to be “normal” are chosen to contribute to the model of normality. A simple model is based on the mean value of these points. Unknown points are compared to this model, e.g. by computing the deviation. Based on a distance threshold, these points can be categorized as normal or anomalous. In this representation, it is also possible to rank objects on a scale from normal to anomalous based on the magnitude of the distance.
4 Method

Missing checks in software can quickly lead to vulnerabilities, which in turn can have serious consequences for a user or system running the vulnerable application. Examples where missing checks immediately lead to security critical issues are applications implementing access control to restrict access to information, functionality, or other resources. In this setting, a missing check can lever out the access control mechanism of the application. Operations on buffers usually require some kind of range or size sanitation to protect from overflows. Programs using project specific APIs must ensure correct usage of the provided interfaces, especially the validation of arguments and return values. Thus detecting missing security checks is vital for securing software.

In this chapter, we revisit CHUCKY, a method for exposing missing checks in source code. The remainder of this chapter is as follows: We first recap CHUCKY in Section 4.1 and proceed by stating the weak points of this method while pointing out possible solutions and improvements to make the method more robust (Section 4.2). We conclude this chapter by presenting our improved version of CHUCKY in Section 4.3.

4.1 CHUCKY

Within a program, data propagates from an input source to a data sink. If the source or sink is sensitive in terms of security, validation of the data is mandatory. Accordingly, checks are placed after the source or before the sink. The aim of CHUCKY is to identify cases where a security check is missing in a semi-automatic manner. Therefore, CHUCKY utilizes taint analysis to statically determine how data may propagate through a program. Starting from a source or sink, CHUCKY collects checks found on the encountered paths. Combined with unsupervised machine learning, CHUCKY is able to expose potential missing checks. Based on the assumption that each source and sink is employed multiple times within the code base, CHUCKY automatically infers the importance of a check by learning program-
ming patterns, found in other functions, involving an identical source or sink. If a function does not contain a check that is present in the majority of the other functions, CHUCKY reports a missing check to the auditor, who can manually verify if this check is necessary or security relevant. The method is structured in five steps. Each of them is described in the following.

1. **Robust parsing.** The source code is processed by a robust parser first. The resulting abstract syntax trees are used to extract sources and sinks and all API symbols of functions, i.e. data types used in parameters and local variable declarations as well as the names of all called functions (see Yamaguchi et al., 2012). Examples of sources are function parameters and return values of function calls. Arguments of a function call are instances of sinks. Further, all assignments and conditions are required for lightweight tainting and to extract checks from conditions.

2. **Neighborhood discovery.** In this step, the API symbols are used to determine the neighborhood, i.e. similar functions using the same source or sink as the function under investigation (the target function). This is essential, since the necessity of a specific check is often dependent on the context of a function. The mere presence of the same source or sink is not sufficient. Only if a function resides in a similar context, its checks contribute to the model of normality. The API symbols are used as features of the context. To this end, all functions are embedded into a feature space spanned by the extracted API symbols. In this space, each function is represented by a weighted use of API symbols. The neighborhood of the target function is then determined by first selecting all functions that contain the same source or sink as the target function. Subsequently, the $k$ closest functions are selected as neighbors.

3. **Lightweight tainting.** Once the neighborhood is discovered, CHUCKY proceeds by lightweight tainting each function with respect to the source or sink under inspection. To this end, CHUCKY models dependencies between identifiers of a function by creating a directed graph. The nodes represent identifier names (including the names of called functions) and the edges two types of dependencies. In detail, an edge from identifier $x$ to identifier $y$ is added

   a) if $y$ is defined by an assignment and $x$ is used in the expression of the assignment, e.g. $y = x$, or

   b) if $x$ is an argument of a function call and $y$ the called function, e.g. $y(x)$.
The taint analysis starts at the identifier corresponding to the source or sink and propagates to all connected identifiers. Therefore, the graph is traversed top–down, i.e. in direction, and bottom–up, i.e. in reverse direction of the edges.

4. **Embedding of functions.** The target function and the functions in the neighborhood are embedded into another vector space spanned by checks. More precisely, all expressions contained in tainted conditions, i.e. conditions that involve at least one tainted identifier, are used as features for the respective function after a slight normalization.

5. **Anomaly detection.** Under the assumption that the majority of the functions of the neighborhood perform checks correctly, the embedding of the previous step is used as the basis for anomaly detection. The checks present in the neighborhood are combined representing a model of normality. More precisely, the center of mass is computed over all functions of the neighborhood. On the basis of this representation, a missing check is indicated by any deviation of the target function from this model. Thus, missing checks are detected by large distances from the center of mass. Moreover, the maximum norm of the deviation vector allows to compute an anomaly score and the dimension with the highest deviation gives a hint at the missing check.

### 4.2 Improving CHUCKY

While CHUCKY was successfully employed to uncover missing check vulnerabilities, we think that there is still room for improvement. One major weakness is the way in which the neighborhood is discovered. While CHUCKY defines the neighborhood of a function by its $k$–nearest neighbors, we think that a method based on clustering is more suitable. As already remarked by Yamaguchi et al. (2013), there exists neither an optimal choice of the number of neighbors nor is it possible to compute. To avoid choosing the neighborhood size directly, we choose a distance criterion, i.e. an upper bound of the maximum distance between two neighbors. This has several advantages: The actual neighborhood sizes can vary in size. This may sound self–defeating at first, but accounts for different sources and sinks which, of course, can have different neighborhood sizes. As a direct result, this helps to identify cases where not a single neighbor or only few neighbors exist. In such cases, CHUCKY should not (or even cannot) be applied. Furthermore,
int foo(void) {
    raw_input_t x;
    input_t input1;
    input_t input2;
    x = read_input();
    // missing check to validate input
    input1 = parse_input(x);
    x = read_input();
    if (!is_valid(x)) {
        return FAILURE;
    }
    input2 = parse_input(x);
    process(input1, input2);
    return SUCCESS;
}

Listing 4.1: Inaccuracy resulting from lightweight tainting. Lightweight tainting the source read_input in line 5 results in over tainting. Tainted identifiers are underlined.

the number of neighbors can be used as a measure of reliability: A high anomaly score for a source/sink with a large neighborhood looks promising, whereas high scores for small neighborhood sizes often turn out to be false–positives. Another side benefit is that the neighborhood relationship is symmetric. This property can be exploited to reduce the amount of computation required to determine the neighborhood for each element in a set of sources/sinks. However, clustering comes at a price: Measuring distances on the basis of raw API symbols turns out to be unreliable due to semantic dependencies between API symbols. Therefore, we apply latent semantic analysis to uncover relations between API symbols and to reduce the dimension of the embedding.

Furthermore, the check selection procedure used by CHUCKY can be improved. In detail, lightweight tainting is not as accurate as possible as shown in the following. Recap that the lightweight tainting technique used by CHUCKY reaches all identifiers that are connected to the source/sink by assignments or argument passing. Listing 4.1 shows a small code snippet where lightweight tainting is performed. Figure 4.1a shows the corresponding graph created by CHUCKY. Starting from the source read_input in line 5, all identifiers are tainted disregarding the fact that the return value of read_input does not affect lines 8 – 12. In this case, CHUCKY cannot detect the missing check in line 6. As a countermeasure, we restrict taint propagation between identifiers to statements connected by data dependency. Additionally, taint analysis is limited to one direction, i.e. in forward direction (top-
(a) Dependencies representing assignments and argument passing modeled for lightweight tainting.

(b) Data dependence graph generated by JOERN. Nodes contain the line number of the statements.

Figure 4.1: Different graph approaches for tainting the code in Listing 4.1. The data dependence graph generated by JOERN (b) has separate nodes for each statement. Consequently, tainting the sources in line 5 and 7 yield different results. The graph used by CHUCKY (a) is based on the identifiers. Unfortunately, both sources define the same identifier. Subsequently, all identifiers are reached when tainting the source read_input.

down) for sources and in backward direction (bottom–up) for sinks. Applying this method in our example yields the correct lines 5, 7, and 13 which do not contain the spurious condition in line 9. At the same time, by selecting the second call to read_input (line 8) as source, the condition would be correctly identified, since the lines 8, 9, 12, and 13 can be reached from this source by data dependency. The corresponding data dependence graph is shown in Figure 4.1b.

On the contrary, this tainting approach is too restrictive. Some checks are missed as demonstrated in Listing 4.2. Assume we want to examine the argument of parse_input in line 6, i.e. a sink. Tainting the argument x yields the statement that contains the argument itself and the statement in line 4 that assigns a value to x. However, the condition in line 5 is missed, even though the execution of line 6 depends on the evaluation of the corresponding predicate. For this reason, our final method considers data and control dependencies equally to extract relevant checks. In other words, we propose program slicing to select all dependent conditions. Note that the checks found this way are a superset of those found by sole tainting.

Another, more severe, restriction of CHUCKY is that the taint propagation stops at function boundaries. While CHUCKY is restricted to checks contained within the boundary of a single function, our implementation is able to find checks outside of the function’s boundary as well. To this end, we use a tailored form of interprocedural slicing.
int bar(void) {
  raw_input_t x;
  input_t input;
  x = read_input(x);
  if (is_valid(x)) {
    input = parse_input(x);
    process(input);
    return SUCCESS;
  }
  return FAILURE;
}

Listing 4.2: Inaccuracy by taint analysis restricted to data dependencies only. Tainting the argument x in line 6 yields the lines 4, 6.

Finally, our implementation is able to work on sources and sinks directly rather than being limited to the analysis of whole functions. This is important in cases where a source or sink occurs more than once in a single function. For instance, both calls to read_input in Listing 4.1 are represented by the same node in the dependence graph of Figure 4.1a. This limitation is removed by our implementation as well.

4.3 CHUCKY–NG

Our refined method reutilizes the five step procedure used by CHUCKY. Each step is explained in the following sections in greater detail and deploys the improvements mentioned in the previous section. In particular, we explain how the different techniques and tools from static code analysis and unsupervised machine learning are combined to expose missing checks in source code. The prototype of our method, called CHUCKY–NG, is realized as a family of command line tools that harmonize well with the tools provided by JOERN. Each step uses one or more tools which are designed to be used in a pipeline. Figure 4.2 shows some of these tools and their interaction in the auditing process. Each component of CHUCKY–NG is realized as UNIX–style filters. Solid lines represent the pipe operator that connects output with input, dashed lines represent auxiliary input. The communication of the auditor with the source code analysis platform JOERN is represented by dotted lines. Since the functionality is separated into independent tools different setups are possible. CHUCKY–NG is open source software licensed under the GNU General Public License version 3 (GPLv3) and can be used freely. This section contains all the details required for the understanding of our method.
4.3.1 Source code parsing

The source code is processed first, primarily by parsing the functions of the code base. This step does not intend to verify syntactical correctness of the code, but to obtain graph structures like the abstract syntax tree and the control flow graph of each function. Each of them providing a different kind of information about the source code required in the following steps. This task is performed independently by the source code analysis platform JOERN (Section 2.6). Like CHUCKY, the derivative CHUCKY–NG uses the abstract syntax trees to identify sources and sinks, to extract API symbols, and to access the checks contained in conditions. Sources and sinks are the starting point of the auditing process. They are denoted by $S$ in the following. The API symbols are used as features in the neighborhood discovery step (see Section 4.3.2). Sources and sinks are embedded into a feature space using extracted checks (see Section 4.3.4). Moreover, CHUCKY–NG utilizes the program dependence graph to compute slices and to perform taint analysis. Both are used to select relevant conditions (see Section 4.3.3).

Once the source code is imported into the graph database provided by JOERN, the auditor begins by selecting targets, i.e. a set $T \subset S$ of sources and sinks that should be examined for missing checks. Of course, not all sources and sinks are interesting in terms of security. Instead, it is the auditor’s expertise to select security sensitive sources and sinks. This requires profound knowledge of the code base and an experienced auditor. Each target is then examined independently for missing checks as explained in the following steps.
4.3.2 Neighborhood discovery

The prediction made by CHUCKY-NG, whether a given source or sink lacks a security check, is based on checks found in the scope of other sources or sinks. Since security checks are often context dependent, sources and sinks employed in a similar context, e.g. parsing network packets or processing audio or video streams, are selected for comparison only. This is crucial for the success of our method. To ease this step for the auditor, unsupervised machine learning is applied as a supporting technique to minimize the amount of manual effort. To this end, we split this step into two successive stages. The first stage incorporates the human knowledge: The auditor begins by manually crafting a preselection \( C(t) \subset S \). This set contains potential neighbors, i.e. candidates for the neighborhood of \( t \), denoted by \( \mathcal{N}(t) \). The preselection of a target usually contains identical or at least similar sources or sinks. To accomplish this task, the auditor can use all the information contained in the code property graph generated by JOERN. For instance, for a given parameter the auditor can select other parameters of the same type and name by running a simple database query. Since the neighborhood is a subset of the neighborhood candidates, an irrational choice of the candidates cannot yield reasonable neighbors. On the contrary, a well chosen preselection can compensate possible weaknesses or limitations in the second stage of this step. The auditor has the control to guide CHUCKY-NG in a meaningful direction. It is even possible to choose the neighborhood entirely by hand.

After the selection of neighbor candidates, CHUCKY-NG tries to automatically reduce this preselection by dismissing sources or sinks that operate in a different context. To this end, CHUCKY-NG utilizes proven techniques from the fields of natural language processing, information retrieval, and unsupervised machine learning. We begin by mapping each source and sink to its function. The function acts as a representative for the source or sink. It is used as the resource of information about the context. Therefore, we proceed by adapting the classic bag of words model over the language of API symbols, denoted by \( \mathcal{A} \), to extract terms for each function. We use the same API symbols previously used in the implementation of CHUCKY, i.e. types of parameters, types of local variable declarations, and the names of called functions, but extend the list by target types of explicit type castings, the return type of the function, and the names of all member variables of structured types. In summary, we make the following definitions: Let \( \mathcal{X} \) be the set of all functions contained in the code base. The mapping \( \text{FUNCTION} : S \rightarrow \mathcal{X} \) maps...
sources and sinks to their functions. This mapping is technically a simple traversal already provided by JOERN. Further, let BOW(x) be a multiset containing the API symbols of function x. We then define an indicator function id_x : A → {0, 1} where

$$id_x(a) = \begin{cases} 1 & \text{if } a \in \text{BOW}(x), \\ 0 & \text{otherwise}. \end{cases}$$

With those definitions, we create an explicit feature map φ that maps functions into a feature space over \( \mathbb{R} \). Each dimension in this space is associated with one API symbol. This mapping is given by:

$$\phi : \mathcal{X} \rightarrow \mathbb{R}^{|A|}$$

$$\phi_a(x) = id_x(a) \cdot \text{tf-idf}(a, x, \mathcal{X}) \quad \text{for all } a \in A,$$

where tf-idf is a common weighting term that assigns smaller weights to API symbols contained in many functions to lower their impact but also depends on the number of times it appears in a given function. In other words, API symbols are weighted accordingly to their relevance in a specific function. The feature map \( \phi \) produces a geometric representation for each function contained in the code base. However, the set of all API symbols is empirically larger than the set of functions (\(|A| > |\mathcal{X}|\)). That means few functions are scattered in a high dimensional space. Furthermore, many API symbols are often used together in the same context and thus are highly correlated. Other API symbols have a similar meaning and should be recognized as synonyms. Consequently, this preliminary representation is not yet suited as a basis for learning the neighborhood. Therefore, reducing the dimension on the feature space induced by the feature map \( \phi \) is obligatory. To this end, we apply latent semantic analysis (LSA) (see Section 3.4) on the term document matrix A, where terms are API symbols and documents are functions:

$$A = \left( \phi(x)^\top \right)_{x \in \mathcal{X}} \in \mathbb{R}^{|\mathcal{X}| \times |A|},$$

(4.1)
i.e. each row contains the feature vector of a function \( x \in \mathcal{X} \). We denote the row of A that corresponds to function \( x \) by \( A(x) = \phi(x)^\top \). While A is sparse, LSA is computationally expensive for code bases with many functions. In this case, LSA might be infeasible. An alternative is to perform the same analysis on a smaller portion of
the matrix containing only functions that belong to considered sources/sinks:

\[ \hat{\mathbf{A}}_t = \left( \phi(\text{FUNCTION}(s)) \right)_{s \in C(t)}^T \in \mathbb{R}^{\left| C(t) \right| \times \left| \hat{\mathbf{A}}_t \right|}, \]

where \( \hat{\mathbf{A}}_t = \bigcup_{s \in C(t)} \text{BOW}(s) \cup \text{BOW}(t) \). However, this requires that LSA is applied for each target independently. Of course, this is only applicable if the set \( C(t) \) is not too small. The result of LSA is an approximation \( \mathbf{B} \in \mathbb{R}^{\left| \mathcal{X} \right| \times d} \) of \( \mathbf{A} \) where \( 1 \leq d \leq \left| \mathcal{A} \right| \) is the number of components after the dimension reduction. While the rows in \( \mathbf{B} \) correspond to the same functions as in \( \mathbf{A} \), the interpretation of the columns is different. Each column represents a concept, i.e. a linear combination of API symbols. Only the \( d \) most significant concepts are retained while all others are dismissed. It is a known problem to choose the value of \( d \). As a rule of thumb \( 50 \leq d \leq 500 \) is often a reasonable choice. Note that, while all entries \( a_{ij} \) of \( \mathbf{A} \) are non-negative by definition, the entries \( b_{ij} \) of \( \mathbf{B} \) can be negative. In addition, the reduced term document matrix is dense, which can be a performance issue.

The tf–idf–weighting is computed by the tool SALLY developed by Rieck et al. (2012) which reads the bag of words of each function and produces an embedding in libsvm format accordingly to the matrix in equation (4.1). CHUCKY–NG provides a tool (chucky-reduce) that reads a feature matrix in libsvm format, performs latent semantic analysis, and returns the reduced feature matrix in libsvm format. Since the reduced matrix is dense loading it into memory can take some time. For that reason, all tools for reducing the neighborhood preselection (see below) can perform LSA in memory and also provide the option to perform LSA on the smaller matrix \( \hat{\mathbf{A}}_t \).

We quantify the difference in the contexts of two functions \( x, y \in \mathcal{X} \) by measuring the distance or similarity between \( \mathbf{B}(x) \) and \( \mathbf{B}(y) \). It is common practice to use the cosine similarity measure in applications where latent semantic analysis is involved. The cosine similarity between two vectors \( a, b \) equals the cosine of their inner angle. This can be formalized using a dot product:

\[ \text{sim}(a, b) = \frac{a \cdot b}{\|a\| \|b\|}. \]

This implies that the similarity between two functions is based on the orientation of their feature vectors rather than the magnitude. The cosine similarity ranges from \(-1\) (maximum dissimilarity), for points lying in complete opposite directions, to \(1\) (maximum similarity), for points lying in the exact same direction. We use
the cosine distance, \(1 - \text{sim}(a, b) \in [0, 2]\) for measuring the distances between two points. Note that the cosine distance is not a valid metric since the coincidence axiom and the triangle inequality are not satisfied.

We now describe how two different unsupervised learning methods are adapted to automatically reduce the preselection of a target to obtain the final neighborhood. The first method is based on the \(k\)-nearest neighbors classifier, the method already used by CHUCKY. The second method is based on clustering (see Section 3.3) and exclusively available in CHUCKY–NG. Both methods are implemented as members of the tool family. The \(k\)-nearest neighbors method is the natural choice for defining a neighborhood. It is formally defined as:

\[
\text{knn}_{M,k} : S \times \mathcal{P}(S) \to S \times \mathcal{P}(S)
\]

\[
(t, \{s_1, \ldots, s_n\}) \mapsto \begin{cases} 
(t, \{s_1, \ldots, s_n\}) & \text{if } n \leq k \\
(t, \{s_{\alpha(1)}, \ldots, s_{\alpha(k)}\}) & \text{otherwise}
\end{cases} \tag{4.2}
\]

where \(\alpha\) is a permutation of \(\{1, \ldots, n\}\) and \(s_{\alpha(i)}\) the \(i\)th closest point from \(\{s_1, \ldots, s_n\}\) to \(t\) based on the feature matrix \(M\).\(^1\) In our case, the feature matrix \(M\) is the term document matrix or the reduce term document matrix. In case of the term document matrix this function is equivalent to the neighborhood detection method of CHUCKY. This method is implemented in the tool chucky-knn. The cluster–based approach of CHUCKY–NG takes an alternative way to detect the neighborhood. The following function takes a set of sources or sinks and returns a partition representing clusters:

\[
\text{clust}_{M,t} : \mathcal{P}(S) \to \mathcal{P}(\mathcal{P}(S))
\]

\[
\{s_1, \ldots, s_n\} \mapsto \{C_1, \ldots, C_m\} \tag{4.3}
\]

where \(\{C_1, \ldots, C_m\}\) is a partition of \(\{s_1, \ldots, s_n\}\). The sets \(C_1, \ldots, C_m\) are the clusters, which are obtained by performing linkage clustering on the set \(\{s_1, \ldots, s_n\}\) using the distance threshold \(t\) to form flat clusters and the feature matrix \(M\).\(^2\) However, the domain and codomain of this function is not compatible with the input units (target–preselection pairs). Therefore, CHUCKY–NG provides the following two solutions: The first option is to dismiss all clusters but the one that contains the

\(^1\)We note that the mapping \(\text{knn}_{M,k}\) is not well defined, since the set of \(k\) nearest neighbors is not necessarily unique.

\(^2\)We note that the mapping \(\text{clust}_{M,t}\) is not well defined, since the clustering process is nondeterministic.
target, formally:

\[
\text{clust}^{\text{compatible}}_{M,t} : S \times \mathcal{P}(S) \to S \times \mathcal{P}(S)
\]

\[
(t, \{s_1, \ldots, s_n\}) \mapsto (t, \{c \in C \setminus \{t\} : t \in \text{clust}_{M,t}(\{t, s_1, \ldots, s_n\})\}).
\]

This is called the compatibility mode, because its input and output format is identical to the legacy \(k\)-nearest neighbors method of \textsc{CHUCKY}. The second solution takes advantages of the symmetry and the fact that in many auditing scenarios most elements of the preselection are examined as targets themselves. In this case, dismissing all other clusters is a waste of computation time. For this reason a rotation tool (\texttt{chucky-rotate}) is part of the tool family. The following mapping describes the functionality:

\[
\text{rotate} : \mathcal{P}(S) \to \mathcal{P}(S \times \mathcal{P}(S))
\]

\[
\{s_1, \ldots, s_n\} \mapsto \{(s_i, \{s_1, \ldots, s_n\} \setminus \{s_i\}) : 1 \leq i \leq n\}.
\]

Combining this tool with the cluster tool by rotating each cluster yields multiple target–neighborhood pairs in a single clustering step. Afterwards, unneeded targets can be removed by applying a simple filter:

\[
\text{filter} : \mathcal{P}(S \times \mathcal{P}(S)) \times \mathcal{P}(S) \to \mathcal{P}(S \times \mathcal{P}(S))
\]

\[
(\{(t_1, S_1), \ldots, (t_n, S_n)\}, T) \mapsto \{(t_i, S_i) : t_i \in T\}.
\]

As a side effect, the rotation tool can be used to generate target–preselection pairs from a set of sources or sinks.

### 4.3.3 Checks selection

To detect missing checks for a given target, \textsc{CHUCKY–NG} first gathers the checks, i.e. expressions contained in conditions, present in the scope of each source and sinks in the neighborhood as well as in the scope of target itself. We define the scope of a source or sink by the statements found in the interprocedural slice corresponding to the source or sink, respectively. Therefore, \textsc{CHUCKY–NG} derives the slicing criterion by determining the identifier defined or used by the source or sink, as well as the enclosing statement. Both are returned by simple traversals in the code property graph, i.e. the traversal \texttt{STATEMENT} yield the enclosing statement and the traversal \texttt{SYMBOL} the defined or used identifier.

The statements contained in the forward or backward slice with respect to the
inferred criterion \( C = (S, x) \) are the result of the traversals \( \text{SLICE}^\downarrow_x(S) \) and \( \text{SLICE}^\uparrow_x(S) \), respectively. As discussed in Section 2.5 slicing can be modeled as a reachability problem in the program dependence graph. The traversals are formally defined as follows:

\[
\text{SLICE}^\downarrow_U(V) := \begin{cases} 
V \cup \text{SLICE}^\downarrow_0 \left( \bigcup_{u \in U} \text{OUT}_{\text{DDG}}^\text{var}(V) \cup \text{OUT}_{\text{CDC}}(V) \right) & \text{if } U \neq \emptyset, \\
V \cup \text{SLICE}^\downarrow_0 \left( \text{OUT}_{\text{DDG}}(V) \cup \text{OUT}_{\text{CDC}}(V) \right) & \text{otherwise}
\end{cases}
\]

This traversal yields all statements in the intraprocedural forward slice by following the edges in the program dependence graph. The traversal \( \text{SLICE}^\uparrow_U(V) \) for backward slicing is defined analog. Subsequently, the relevant conditions for a source or sink are obtained by selecting all conditions contained in the forward or backward slice. This is achieved by appending a filter traversal \( \text{FILTER}_p \), where \( p(\cdot) \) is true if and only if the given node is a condition. Since the traversals \( \text{SLICE}^\downarrow \) and \( \text{SLICE}^\uparrow \) yield only statements in the boundary of a function, conditions outside the function boundary are still missed. For this reason we define new traversals for expanding a source or sink to parameters and return statements of called functions, respectively. More precisely, a source is expanded to the formal parameter of called functions, if the source is connected to the corresponding actual parameter via direct data flow. Expanding of sinks is defined analog. Both traversals can be executed recursively. The traversals are denoted by \( \text{EXPAND}^\downarrow_d \) and \( \text{EXPAND}^\uparrow_d \) for sources and sinks, respectively. The depth of recursion is limited by the value of \( d \).

In summary, relevant conditions of a source \( s \) determined by combining the introduced traversals in the following way:

\[
\text{COND}_d(s) = \left( \text{FILTER}_p \circ \text{SLICE}^\downarrow_{\text{SYMBOL}}(\{s\}) \circ \text{STATEMENT} \circ \text{EXPAND}^\downarrow_d \right)(\{s\}).
\]

The combined traversal for sinks is constructed analog using the counterparts of the slice and expand traversals. The set \( \text{COND}_d(s) \) contains all conditions extracted for the source or sink \( s \).

4.3.4 Source/Sink embedding

The information about the neighborhood and the relevant checks for each source or sink gained in the previous two steps is used in the last two steps to uncover potential missing checks. \textsc{Chucky-NG} is deployed under the assumption that the majority of the neighbors perform checks correctly. Thus, any deviation of the tar-
get from its neighborhood with respect to the checks performed is an indication for a missing check. To this end, this step proceeds by describing how the conditions of each source or sink are embedded in a feature space. The last step uses this embedding to detect anomalies.

Each source or sink is transformed into a feature vector that mathematically describes the checks found for the source or sink. Therefore, all expressions, including the sub expressions, contained in the abstract syntax tree of a condition are extracted and normalized. For an arbitrary expression $x$ of type $y$ the normalized expression of $x$, denoted by $EXPR(x)$, is defined as follows:

$$EXPR(x) = \begin{cases} 
\text{NORMALIZE}_y(x) & \text{if } x \text{ is a leaf node} \\
\text{NORMALIZE}_y(EXPR(c_1), \ldots, EXPR(c_n)) & \text{otherwise.}
\end{cases}$$ (4.7)

The expressions $c_1, \ldots, c_n$ are children, i.e. sub expressions, of $x$. For instance, the normalized expression of $((a - 1) \geq 0)$ is $((a - \text{NUM}) $CMP$ \text{NUM})$, where $a - 1$ and $0$ are sub expressions, which are normalized in advance, due to the recursive definition of $EXPR$. The mapping $\text{NORMALIZE}_y$ depends on type of the expression. For example, the normalization function for expressions of type EqualityExpression ($==$, $!=$) or RelationalExpression ($>$, $\geq$, etc.) replaces the operator with $\text{CMP}$. Most expressions do not undergo any kind of normalization, however at some points normalization is necessary:

1. **Removal of negations.** CHUCKY–NG does not evaluate conditions and pays no attention to the actions performed afterwards. Consequently, it does not matter whether a condition evaluates to true or false. For this reason negations are removed. See Figure 4.4 for an additional example.

2. **Unification of relational operators.** Likewise, operators of relational expressions are replaced by the unique symbol $\text{CMP}$.
3. **Unification of literals.** Numerical literals (including single characters) are replaced by $\$NUM$. This also affects arithmetic operations such that $1+2$ becomes $\$NUM$ instead of $\$NUM + \$NUM$. This way, the normalized expressions of $x < 1+2$ and $x < 3$ do not differ. String literals are replaced by $\$STR$.

4. **Normalization of source/sink symbols.** The identifier of the source or sink is replaced by $\$SYM$. This is important, since the identifiers bound to the sources or sinks are not identical in general.

Further, not all expressions contained in a condition are used as features:

1. **Pruning of call expressions.** The arguments of call expression are considered as not being checked, i.e. expressions of arguments are ignored. In case of interprocedural analysis, CHUCKY–NG descends in called functions nonetheless, given the source code is available.

2. **Pruning of array indexing expressions.** An array itself is considered as not being checked in the case one of its field elements is accessed.

3. **Pruning of member accesses.** The children of member accesses are not included in the expression set.

4. **Pruning of unary operations.** The children of unary operations are not included in the expression set.

Figure 4.3 shows the normalization of the condition contained in the AST of Figure 2.1. The normalization is performed with respect to the symbol $\text{len}$, which is substituted by $\$SYM$. As always, literals are normalized, comparison operators are replaced by $\$CMP$, and sub expressions are put within parenthesis. The normalized tree is created in postorder, where child nodes are normalized.
first. There are a total of six checks extracted from the condition: $SYM, NUM, SIZE, (SYM $CMP NUM), (SYM $CMP SIZE), and ((SYM $CMP NUM) \| ((SYM $CMP SIZE))$.

The set ExprSet($x$) denotes the set of all normalized expressions contained in the expression $x$. Formally:

$$\text{ExprSet}(x) = \{\text{Expr}(x)\} \bigcup_{c \in \text{Children}(x)} \text{ExprSet}(c),$$

where CHILDREN is a traversal yielding all children of an AST node.

Let $E$ denote the set of all normalized expressions contained in any condition. Similar to the function embedding we define an indicator function $\text{id}_s : E \rightarrow \{0, 1\}$ for each $s \in S$, where

$$\text{id}_s(e) := \begin{cases} 
1 & \text{if } e \in \bigcup_{c \in \text{Cond}(s)} \text{ExprSet}(c), \\
0 & \text{otherwise.}
\end{cases}$$

Each source or sink $s$ is then embedded into a binary feature space. The feature vector of $s$ covers all the information of the relevant checks. The feature map is formally given by:

$$\varphi : S \rightarrow \{0, 1\}^{|E|},$$

$$s \mapsto (\text{id}_s(e))_{e \in E}.$$  

**4.3.5 Anomaly detection**

The embedding of checks and the neighborhood of each target derived in the previous steps contain all the information required to detect missing checks. At first, we need to choose an appropriate model of normality that expresses how checks in the target's context should look like. To this end, we create a model representing the checks distributed in the neighborhood. The reasoning is that the sources or sinks in the neighborhood operate in a similar context as the target and, thus, contain representative checks. Moreover, checks found in many neighbors are expected to have high importance. Consequently, a check that is often used in association with the sources or sinks is probably required for the secure or correct usage. Therefore, we seek a single vector representation that unifies the extracted checks of all neighbors. The natural choice for a model that satisfies these requirements is the center
of mass $\mu_t$:

$$\mu_t := \frac{1}{|\mathcal{N}(t)|} \sum_{s \in \mathcal{N}(t)} \varphi(s) \in [0, 1]^{\mathcal{E}}.$$ 

The entries in the vector $\mu_t$ corresponding to checks with high importance are close to one, whereas entries corresponding to checks that occur occasionally are close to zero. That means, the importance of a check is expressed by a single numerical value. The vector $d_t = \mu_t - \varphi(t)$ is the deviation of the target $t$ from this model. Since $\varphi(t)$ is a binary vector, the entries in $d_t$ range from $-1$ to $1$. Entries are positive or negative if and only if the corresponding entry in $\varphi(t)$ is zero or one, respectively.

The interpretation is as follows: Positive entries in $d_t$ state the fraction of neighbors involving the corresponding checks while the target does not perform this check. Similarly, negative entries in $d_t$ state the fraction of neighbors not involving the corresponding checks while the target does perform this check. Since we are interested in missing checks, we focus on positive entries, especially the maximum entry:

$$\max_{e \in \mathcal{E}} (\mu_t - \varphi(t))_e.$$ 

We refer to this value as anomaly score. The corresponding check is given by

$$\arg \max_{e \in \mathcal{E}} (\mu_t - \varphi(t))_e$$

i.e. the expression that is present in one of the conditions deployed by most of the neighbors, but not in the target itself. For instance, if the anomaly score is 0.8 and the corresponding expression is $\$SYM$ $\$CMP$ $\$NUM$ the auditor can conclude that 80% of the neighbors compare the symbol associated with the source or sink with a numerical literal, while the chosen target does not contain a check of this kind. Figure 4.5 illustrates the anomaly detection process graphically. The sources or sinks of the neighborhood are close together while the target is far away. This deviation is quantified by calculating the distance vector from the target $t$ to the center of mass $\mu_t$. The source or sink labeled $x$ deviates from $\mu_t$ as well and may contain a missing check. However, it does not affect the center of mass too much, i.e. the model of normality is robust against some amount of noise.

The anomaly score is calculated by the tool $\text{chucky-score}$ that implements the
Figure 4.5: Simplified picture showing the anomaly detection step. The geometrical coordinates are based on the checks embedding that spans a high dimensional space in reality. The target $t$ deviates from the model of normality, $\mu_t$, derived from the neighborhood $\mathcal{N}(t)$. This is an indication for a potential missing check. CHUCKY-NG is able to calculate an anomaly score based on the difference and reports the missing expression to the auditor.

The following functionality:

$$\text{score}_M : \mathcal{S} \times \mathcal{P}(\mathcal{S}) \rightarrow [-1, 1] \times \mathcal{E}$$

$$(t, \{s_1, \ldots, s_n\}) \mapsto (\max_{e \in \mathcal{E}} d_t, \arg \max_{e \in \mathcal{E}} d_t),$$

where $d_t$ is the deviation vector and $M$ the feature matrix containing the check embedding.
5 Results

We evaluated CHUCKY–NG by performing controlled experiments analog to the experiments conducted in the evaluation of CHUCKY (see Yamaguchi et al., 2013). The aim of the experiments was to compare the detection performance of CHUCKY–NG using different strategies for neighborhood detection as well as check selection. Therefore, we created a set of experiments containing vulnerable code intentionally. This dataset is based on five open source projects providing a total of six independent code bases. Each project has been vulnerable due to a missing check in the past. Our dataset extends the dataset that has been used in the evaluation of CHUCKY by one additional vulnerability discovered lately in the Linux kernel by Yamaguchi et al. (2014). We created multiple experiments for each code base, similarly to the controlled experiments used in the evaluation of CHUCKY. The only difference is that we made use of the new ability to analyze sources/sinks directly rather than whole functions.

5.1 Setup of the evaluation

We restate the generation of the experiments for the sake of completeness: For each code base the vulnerability is patched first by adding the missing check to sanitize the data flow from or to the source or sink, respectively. Afterwards, for each code base, identical sources/sinks of related functions are identified. Each of them could have potentially suffered from the same flaw. Based on those sources and sinks several experiments are created by removing the security check in a rotating manner, leaving exactly one source/sink unsanitized at a time. These sources or sinks are labeled vulnerable or non–vulnerable, accordingly. All other sources/sinks remained unlabeled. This process yields multiple experiments with artificial as well as real vulnerabilities and enables us to evaluate our method on a larger dataset, while employing CHUCKY–NG in realistic scenarios only.

We briefly describe the vulnerabilities of each project and the involved source or sink. Table 5.1 provides an overview of all six code bases and lists the common vul-
nerabilities and exposures (CVE) identifier of the vulnerabilities for reference. Additionally, Table 5.2 provides a statistic about each code base including the number of functions and API symbols.

- **Firefox 4.0** The JavaScript engine of Firefox (version 4.0), one of the most used web browsers, contains a dangling pointer in the native code implementation of JavaScript functions. A missing validation of the number of arguments (i.e. parameter `uintN argc`) passed to the function enables an attacker to execute code. The same parameter exists 518 times in the code base. Each of them is included in the neighborhood preselection.

- **Linux 2.6.34** The missing validation of file ownership in the filesystem code of the Linux kernel 2.6.34 and earlier allows local users to bypass permission checks. The source is the parameter `struct dentry dentry`. It exists 1104 times in the code base. Each of them is considered as potential neighbor.

- **LibPNG 1.2.44** A missing check allows remote attackers to cause a denial of service via a crafted PNG image. The source under inspection is the parameter `png_uint_32 length`. It exists 29 times in the code base. Each of them is contained in the neighborhood preselection.

- **LibTIFF 3.9.4** A stack–based buffer overflow in the library LibTIFF allows an remote attacker to cause a denial of service via a crafted file. Checking the member variable `tdir_count` of the Parameter `TIFFDirEntry dir` can avoid this vulnerability. The same source exists 39 times in the code base. Each of them is included into the neighborhood preselection.

- **Pidgin 2.7.3** The popular instant messenger Pidgin contains a missing check vulnerability in version 2.7.3 of its messaging library. The return value of
the internal base64 decoding routine (purple_base64_decode) is used unchecked. This source exists exists 33 times in the code base. Each of them is included into the neighborhood preselection.

- **Linux 3.11** The missing validation of a length variable allows local users to trigger a buffer overflow via crafted write operations. The variable flows unchecked into the third argument of copy_from_user. This sink exists 287 times in the code base. Each of them is considered as potential neighbor.

For each of the 73 experiments CHUCKY–NG is employed to compute the anomaly score of each labeled source and sink while considering all identical sources or sinks as potential neighbors (this includes labeled as well as unlabeled sources and sinks). Each labeled source/sink is then ranked in two different settings. In the first setting, CHUCKY–NG is configured to use the classic \( k \)–nearest neighbors method in combination with intraprocedural program tainting. The number of neighbors are chosen from the range 1 to 50. The second setting uses the novel clustering method paired with interprocedural program slicing. Analog to the first setting, we use different thresholds for the maximal intra–cluster distance (the cophenetic distance). The interprocedural slicing algorithm stops after two transitive function calls. In both settings we use different parameters for the latent semantic analysis. For the experiments based on the Linux filesystem code and Firefox, we use the local version of LSA. Additionally, the \( k \)–nearest neighbors approach is performed without dimension reduction.

We computed the true–positive and false–positive rates for each experiment by applying a binary classifier on the rankings produced by CHUCKY–NG. All labeled sources and sinks with an anomaly score above a certain threshold are classified as vulnerable. In case of clustering, results for neighborhoods smaller than 5 are omitted, such that CHUCKY–NG is employed only if the prerequisite for a reasonable neighborhood is guaranteed. On the basis of the rankings created by CHUCKY–NG, we computed the receiver operating characteristic (ROC) curve and visualize the corresponding area under the curve (AUC) as a performance measure (see Bradley, 1997). In detail, the detection rate of CHUCKY–NG is plotted against the parameter used in the neighborhood detection step, i.e. the number of neighbors \( (k) \) or the maximal cophenetic distance \( (t) \) in the first and second setting, respectively. A detection rate of 1.0 means that the source/sink labeled vulnerable is ranked highest in all experiments. Detection rates below 1.0 indicate that some non–vulnerable sources/sinks are ranked higher or just as high as the vulnerable source/sink.
Table 5.2: Overview of the six code bases. The versatility of the dataset makes it possible to evaluate CHUCKY–NG under different conditions.

<table>
<thead>
<tr>
<th>Project</th>
<th>functions</th>
<th>experiments</th>
<th>neigh. candidates</th>
<th>API symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox 4.0</td>
<td>13780</td>
<td>10</td>
<td>518</td>
<td>25449</td>
</tr>
<tr>
<td>Linux 2.6.34.13</td>
<td>20882</td>
<td>8</td>
<td>1041</td>
<td>38630</td>
</tr>
<tr>
<td>LibPNG 1.2.44</td>
<td>460</td>
<td>19</td>
<td>29</td>
<td>1255</td>
</tr>
<tr>
<td>LibTIFF 3.9.4</td>
<td>705</td>
<td>9</td>
<td>39</td>
<td>1416</td>
</tr>
<tr>
<td>Pidgin 2.7.3</td>
<td>7909</td>
<td>20</td>
<td>33</td>
<td>11901</td>
</tr>
<tr>
<td>Linux 3.11</td>
<td>20807</td>
<td>7</td>
<td>287</td>
<td>42476</td>
</tr>
</tbody>
</table>

the following section, we focus on experiments that yielded deviations in the detection rates of both settings. In particular, we want to analyze the impacts of clustering and interprocedural program slicing on the detection rate of CHUCKY–NG.

5.2 Discussion of the results

The neighborhood discovery is an important part, since it forms the foundation of the success of our method. An inaccurate neighborhood can lead to a drop in the detection rate of the method, especially when the neighborhood becomes too large. At some point no more sources or sinks of the same context exist and thus increasing the neighborhood size beyond this point will mix sources or sinks of different contexts. Subsequently, the model of normality is not representative for the target’s context. This effect has already been observed in Yamaguchi et al. (2013) on the basis of the Linux filesystem code. We reproduced this effect (see Figure 5.1). When the neighborhood becomes too large the detection rate drops instantly. This is an indication that the necessity of the check is extremely context dependent. Therefore, the right choice of $k$ is important. A look at the code confirms our assumption: The source is of heavy use in the overall code base, especially in different contexts. While the security check is required for the labeled functions, it is irrelevant in most other cases. Therefore, a representative neighborhood is essential to detect this specific missing check.

The same experiment was performed with CHUCKY–NG using clustering. The detection performance is displayed in Figure 5.2a. For a large range of thresholds ($0.3 < t < 0.7$) the detection performance is optimal, i.e. CHUCKY–NG detects each missing check without a single false alarm. A look at the average neighborhood size (see Figure 5.2b) shows that the neighborhood is almost constant in this range.
Figure 5.1: The minimal detection rate of CHUCKY–NG using $k$–nearest neighbors as neighborhood discovery method and intraprocedural program tainting for check selection.

Even for larger neighborhood sizes the detection rate of this setting is superior. We have to note, however, that measuring distances is only expressive when latent semantic analysis is performed. Measuring distances without LSA is ineffective. This is a considerable limitation since the number of dimensions where CHUCKY–NG performs best is unknown in advance. In our experiments, we attained the best results for 50–100 dimensions, with exception of the Linux filesystem code. In the latter case the best results were obtained for 200–500 dimensions. On the other hand, not all sources/sinks are contained in a sufficiently large cluster. Consequently, CHUCKY–NG cannot be applied at all. The sizes of the neighborhoods obtained by clustering vary for different code bases. For example, Figure 5.3 shows the amount of discontinued experiments of the two code bases LibPNG and Pidgin. Dependent on the distance threshold and the LSA parameter experiments were aborted. However, the checks in those cases are performed to ensure the secure usage of APIs. Therefore, checks are usually required in most cases anyway, i.e. checks are independent of the context. Moreover, only few neighborhood candidates are available. Thus, neighborhood detection can be skipped, since the neighborhood preselection is sufficient. In this case CHUCKY–NG achieved perfect results. In addition, it is realistic that some sources/sinks are rather unique. Our method is able to detect these circumstances, in contrast to the $k$–nearest neighbors approach.

In terms of check selection, the experiments on basis of the staging code of Linux
Maximal Cophenetic Distance ($t$)

(a) Minimal Detection rate

(b) Average neighborhood size

Figure 5.2: The minimal detection rate (a) and the corresponding average cluster size (b) of CHUCKY-NG using clustering as neighborhood discovery method and interprocedural program slicing for check selection. For small cophenetic distances no neighborhood is found. After the distance reaches a first threshold ($t \approx 0.3$) the clusters yield a small but stable neighborhood. For $t \approx 0.7$ another threshold exists. Beyond this point the neighborhood size increases rapidly and the detection performance drops instantly.

LibPNG  

<table>
<thead>
<tr>
<th>$c = 50$</th>
<th>$c = 100$</th>
<th>$c = 200$</th>
<th>$c = 500$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pidgin

<table>
<thead>
<tr>
<th>$c = 50$</th>
<th>$c = 100$</th>
<th>$c = 200$</th>
<th>$c = 500$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: The ratio of skipped experiments of the code bases LibPNG and Pidgin for different numbers of dimensions ($c$) retained after the LSA. Experiments are skipped if the detected neighborhood contains less than 5 sources or sinks.
3.11 show that tainting is not sufficient. The statement containing the sink, i.e. the third argument of `copy_from_user`, is control dependent on the predicate that contains the important check. Consequently, no data dependency exists between the predicate and the argument. Hence, the check is only found by program slicing. On the other hand, experiments on basis of the library LibTIFF yield significantly worse results when using interprocedural slicing compared to intraprocedural tainting (see Figure 5.4). A closer look at the code reveals the following: The check is missing inside the boundary of the function, but present outside of the function boundary. However, the check should be located in the function nonetheless, since it is used to validate a function parameter. As a result, CHUCKY–NG does not report a missing check because it is found in the interprocedural scope of the source. Albeit it is performed at the wrong place. Our method is not capable to identify checks performed too late. This problem is a known limitation amplified by interprocedural analysis. Unfortunately, the dataset does not contain cases where interprocedural analysis is required. Thus the theoretical advantages of interprocedural analysis cannot be revealed by our experiments.

All other experiments yield similar results in both settings. Our results also re-
flect the results obtained in the evaluation of CHUCKY.

In summary, CHUCKY–NG is able to detect the absence of the manually removed checks in all cases. However, dependent on the neighborhood, other checks are ranked higher and, thus, detected as false positives. When comparing $k$–nearest neighbors with clustering, one should prefer clustering in cases where the neighborhood is relevant. For sources and sinks that should be validated independently of the context CHUCKY–NG performs best without neighborhood detection at all. While tainting is not sufficient, interprocedural slicing has downsides as well. In the best cases, we reached a detection rate above 90\%, averaged over all experiments. That means, CHUCKY–NG is able to detect most missing checks without false–positives.
6 Conclusion

In this work, we have revisited CHUCKY, a method for exposing missing check vulnerabilities in source code using anomaly detection. To account for the context-dependent validation of data flow, we have presented a new approach to extract similar usage patterns of data sources and sinks in C/C++ source code: Instead of relying on the \(k\)-nearest neighbors of a source or sink for neighborhood discovery we have successfully applied linkage clustering in combination with latent semantic analysis to group sources or sinks accordingly to their API usage patterns. While we have seen that clustering is not applicable in cases where neighborhood detection is negligible, i.e. in cases where sanitation of sources or sinks is mandatory independently of the context, we were able to demonstrate the advantages of clustering over the \(k\)-nearest neighbors method in all other cases. Unfortunately, the success of clustering is dependent on the choice of the dimension parameter used by the latent semantic analysis. Furthermore, we utilized interprocedural program slicing as a drop-in replacement for lightweight tainting to select the relevant checks of a given source or sink. Albeit the theoretical advantages of interprocedural analysis, a known limitation of the presented method is amplified: The described method is restricted to the plain detection whether a check is present or missing. Unfortunately, this implies that it is not possible to judge about the correct usage of checks. Especially it is not possible to decide if a check is performed in the right place. Interprocedural slicing enlarges the scope in which CHUCKY-NG searches for checks. Hence, the possibility that an irrelevant check is included grows by the use of the interprocedural slicing algorithm. However, in terms of intraprocedural analysis, we were able to demonstrate the theoretical advantages of slicing over tainting by our experiments. We have to note, however, that the tainting algorithm used in CHUCKY-NG is not directly comparable to the lightweight tainting implementation in CHUCKY, since it is based on taint propagation between statements rather than identifiers.

In our evaluation, we were able to reach a detection rate above 90% on average using clustering and interprocedural slicing. That means, CHUCKY-NG is able
to detect most missing checks without false-positives. To mitigate effect of false alarms, CHUCKY–NG reports the missing checks to the auditor who can quickly separate false positives from true positives. Our results are comparable to the performance of CHUCKY. However, we think that our refined method is superior to CHUCKY in practice.

Despite the current development status of the method implemented in CHUCKY–NG, we see further extensions that may help to improve the detection performance even more. The following suggestions are not intended to replace parts of CHUCKY–NG, but to provide alternatives or additional options to existent techniques.

- Instead of defining similar usages of a source or sink exclusively on API symbols, other features are possible in general. While structural features do not bring up the desired effect (see Yamaguchi et al., 2012) introducing features representing dependence relations between API symbols are worth a closer look.

- As an alternative to the learning based approach, the neighborhood of a source/sink can be based on pairs of sources and sinks. Such source–sink systems were successfully used in (Yamaguchi et al., 2015). Additionally, source–sink systems indirectly define the scope from which checks are selected. For example, it is possible to compute the slices for the source and sink. The scope of the source–sink system is then defined by the intersection of the slices.

- The scope of a source or sink can be limited by introducing a parameter specifying the maximum number of edges traversed by the slicing algorithm. Thereby, the maximum distance of a check from a source or sink can be controlled to mitigate the effect from checks performed in other places. This option would also eliminate the distinction between intra- and interprocedural analysis.

Due to the modular structure of the method, it is possible to use all techniques interchangeable. That way, the auditor is able to decide which method seems best in the current setting. Moreover, it is quickly possible to integrate one of the extensions listed above into the auditing process.
Bibliography


