Georg-August-Universität Göttingen

Pattern-Based Vulnerability Discovery

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im PhD Programmme in Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von
Fabian Yamaguchi
aus Bochum

Göttingen 2015

Betreuungsausschuss:	Prof. Dr. Konrad Rieck Georg-August Universität Göttingen Prof. Dr. Jens Grabowski Georg-August Universität Göttingen
Prüfungskommission:	
Referent:	Prof. Dr. Konrad Rieck Georg-August Universität Göttingen
Korreferenten:	Prof. Dr. Thorsten Holz Ruhr-Universität Bochum Prof. Dr. Jean-Pierre Seifert Technische Universität Berlin
Weitere Mitglieder der Prüfungskommission:	Prof. Dr. Dieter Hogrefe Georg-August Universität Göttingen Prof. Dr. Carsten Damm Georg-August Universität Göttingen Prof. Dr. Stephan Waack Georg-August Universität Göttingen
Tag der mündlichen Prüfung:	30. Oktober 2015

Abstract

With our increasing reliance on the correct functioning of computer systems, identifying and eliminating vulnerabilities in program code is gaining in importance. To date, the vast majority of these flaws are found by tedious manual auditing of code conducted by experienced security analysts. Unfortunately, a single missed flaw can suffice for an attacker to fully compromise a system, and thus, the sheer amount of code plays into the attacker's cards. On the defender's side, this creates a persistent demand for methods that assist in the discovery of vulnerabilities at scale.

This thesis introduces *pattern-based vulnerability discovery*, a novel approach for identifying vulnerabilities which combines techniques from static analysis, machine learning, and graph mining to augment the analyst's abilities rather than trying to replace her. The main idea of this approach is to leverage patterns in the code to narrow in on potential vulnerabilities, where these patterns may be formulated manually, derived from the security history, or inferred from the code directly. We base our approach on a novel architecture for robust analysis of source code that enables large amounts of code to be mined for vulnerabilities via traversals in a *code property graph*, a joint representation of a program's syntax, control flow, and data flow. While useful to identify occurrences of manually defined patterns in its own right, we proceed to show that the platform offers a rich data source for automatically discovering and exposing patterns in code. To this end, we develop different vectorial representations of source code based on symbols, trees, and graphs, allowing it to be processed with machine learning algorithms. Ultimately, this enables us to devise three unique pattern-based techniques for vulnerability discovery, each of which address a different task encountered in day-to-day auditing by exploiting a different of the three main capabilities of unsupervised learning methods. In particular, we present a method to identify vulnerabilities similar to a known vulnerability, a method to uncover missing checks linked to security critical objects, and finally, a method that closes the loop by automatically generating traversals for our code analysis platform to explicitly express and store vulnerable programming patterns.

We empirically evaluate our methods on the source code of popular and widely-used open source projects, both in controlled settings and in real world code audits. In controlled settings, we find that all methods considerably reduce the amount of code that needs to be inspected. In real world audits, our methods allow us to expose many previously unknown and often critical vulnerabilities, including vulnerabilities in the VLC media player, the instant messenger Pidgin, and the Linux kernel.

Dedicated to Jana and my parents.

Acknowledgments

I would like to take this as an opportunity to thank everyone who has supported me in this research during the last couple of years. All of you have made my time as a PhD student an experience that I would not want to miss.

First of all, I would like to thank Prof. Dr. Konrad Rieck for being an excellent advisor to me. Your valuable comments and suggestions as well as your critical questions are the foundation of this research. In addition, the time and energy you spend listening to and understanding each of your students to support them in their development is invaluable. Thank you for giving me the time to write a thesis that I am actually happy with, and finally, thank you for lending me your copy of *GTA5* to support my writing endeavor. Prospective PhD students should know that they will have trouble finding a better advisor.

I would also like to thank Prof. Dr. Thorsten Holz and Prof. Dr. Jean-Pierre Seifert for taking the time to read and referee this thesis. Given your valuable experience in applied computer security and your full schedules, it is an honor to have you on the thesis committee. In addition, I would like to thank all other members of the thesis committee for their valuable time: Prof. Dr. Dieter Hogrefe, Prof. Dr. Carsten Damm, and Prof. Dr. Stephan Waack.

Furthermore, I would also like to express my gratitude for my colleagues Daniel Arp, Hugo Gascon, Christian Wressnegger, and Alwin Maier from the Computer Security Group at the University of Goettingen, as well as Ansgar Kellner, Salke Hartung, and Hang Zhang from the Telematics Group. I am also grateful for having had the opportunity to work with highly motivated, skilled, and friendly researchers from other institutions throughout this work, including Jannik Pewny and Felix Schuster from Ruhr University Bochum, Malte Skoruppa from Saarland University, Aylin Caliskan-Islam and Rachel Greenstadt from Drexel University, Markus Lottmann from Technische Universität Berlin, and Nico Golde from Qualcomm Research Germany. Moreover, I would like to express my deepest appreciation for my friends Daniel Arp, Markus Lottmann, Bernhard Brehm, Nico Golde, and Gregor Kopf who have not stopped to inspire and encourage me whenever we find time to talk. I would also like to thank Felix Lindner for giving me a first job in vulnerability discovery ten years ago, and Sergey Bratus for first introducing me to latent semantic analysis.

Finally, I gratefully acknowledge funding from the *German Research Foundation* under the project DEVIL (RI 2469/1-1), and the *German Federal Ministry of Education and Research* under the project PROSEC (FKZ 01BY1145).

Contents

List of Figures

List of Tables

Publications

The research presented in this thesis combines and extends work performed in the course of a PhD program pursued by the author at the Georg-August-Universität Göttingen. As is customary in areas of applied computer science, individual results were published in the proceedings of scientific conferences throughout the program. This resulted in the following peer-reviewed publications that include work substantial for the completion of this thesis.

- *Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning. Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 5th USENIX Workshop on Offensive Technologies (WOOT) [163]*
- *Generalized Vulnerability Extrapolation using Abstract Syntax Trees. Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 28th Annual Computer Security Applications Conference (ACSAC).* ***Outstanding Paper Award****.* [164]
- *Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, Konrad Rieck. 20th ACM Conference on Computer and Communications Security (CCS)* [166]
- *Modeling and Discovering Vulnerabilities with Code Property Graphs. Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 35th IEEE Symposium on Security and Privacy (S&P)* [162]
- *Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 36th IEEE Symposium on Security and Privacy (S&P)* [165]

This thesis presents a comprehensive overview of pattern-based vulnerability discovery based on these publications as well as novel material, where the goal is to point out the underlying theme connecting the presented approaches. In particular, Chapter 2 presents and extends work on code property graphs and their storage in graph databases previously published in [162], and additionally describes an interprocedural extension of code property graphs first presented in [165]. The method for extrapolation of vulnerabilities described in Chapter 4 was presented in [164] and [163]. Moreover,

our method for the detection of missing checks outlined in Chapter 5 was previously published in [166], and finally, the work on automatic inference of search patterns via clustering in Chapter 6 was published in [165]. The author hereby assures that he is the lead author of all five aforementioned publications.

The effort to understand the relation between these individual pieces furthermore lead to the development of novel and previously unpublished material, in particular, the complete architecture for robust source code analysis presented in Chapter 2, and the general procedure for learning on code property graphs outlined in Chapter 3.

In addition, the insights gained into code analysis and machine learning allowed the author to contribute to the following peer-reviewed papers on vulnerability discovery, malware detection and anonymous communication.

- *Structural Detection of Android Malware using Embedded Call Graphs. Hugo Gascon, Fabian Yamaguchi, Daniel Arp, Konrad Rieck. 6th ACM Workshop on Security and Artificial Intelligence (AISEC)* [44]
- *Torben: A Practical Side-Channel Attack for Deanonymizing Tor Communication. Daniel Arp, Fabian Yamaguchi, and Konrad Rieck. 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)* [9]
- *De-anonymizing Programmers via Code Stylometry. Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 24th USENIX Security Symposium* [19]
- *VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. Henning Perl, Daniel Arp, Sergej Dechand, Fabian Yamaguchi, Sascha Fahl, Yasemin Acar, Konrad Rieck, and Matthew Smith. 22nd ACM Conference on Computer and Communications Security (CCS)* [108]
- *Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp and Konrad Rieck. 11th International Conference on Security and Privacy in Communication Networks (SECURECOMM)* [43]

While the results of these papers are not included in this thesis, references have been made throughout the text to highlight connections between this thesis and the papers.

1 Introduction

As we increasingly embrace the convenience of information technology, the security of computer systems is becoming a growing concern. To provide it, secure network protocols, architectures and cryptographic algorithms are crucial. Unfortunately, the success of these efforts is repeatedly undermined by subtle flaws in implementations. A particular prominent and absurd example of such a flaw is the *Heartbleed* vulnerability [29] found in the cryptographic library OpenSSL in April 2014. While the library provides the basis for encrypted transmission of Web pages on a large number of systems, a single missing sanity check in its code turned it into a gaping security hole. In effect, attackers gained the ability to read sensitive information from an estimated 24%-55% of the most popular one million websites serving encrypted pages [38], while ironically, servers not offering encryption remained immune. This highlights the central role the quality of the underlying program code plays for the security of computer systems.

In total, efforts for the discovery of these kinds of vulnerabilities result in the disclosure of between 4600-6800 vulnerabilities per year, as measured over the last eight years [140]. While this number may seem high at first, these vulnerabilities are distributed over the entire software landscape and are of varying severity. Attackers interested in compromising specific targets therefore find a much smaller amount of vulnerabilities at their disposal. For example, only 31 critical vulnerabilities were disclosed in the Firefox Web browser in 2014 [see 99], some of which are relevant only for few versions of the program. In effect, vulnerabilities have become a valuable good, leading to the establishment of vulnerability markets in recent years, where previously unknown flaws and accompanying exploits are sold for hundreds of thousands of dollars [47].

To date, the vast majority of critical vulnerabilities is found by manual analysis of code by security experts. This includes recent high impact vulnerabilities such as *Heartbleed* [see 121], the *Shellshock* vulnerability in the GNU bash shell [see 23, 105], as well as the recently discovered *Stagefright* vulnerabilities that allow attackers to remotely control Android phones by sending crafted multimedia messages to victims. In fact, the role professional security analysts play in the discovery of critical vulnerabilities by manually reviewing code cannot be overstated. In all of its stages, vulnerability discovery is a tedious task, requiring an intimate knowledge of the target software system to be gained, possible attack scenarios to be devised, and finally, flaws that can be leveraged to bypass security measures to be identified. The difficulty of these tasks creates a persistent demand for new methods to assist analysts in their daily work.

In the spirit of theoretical computer science and its roots in mathematics, academic work in the area has mostly focused on the development of formal and exact methods such as model checking [see 11] and symbolic execution [see 18, 132], which allow properties of the code to be verified in an automated deductive process in the flavor of a mathematical proof. While demonstrated to be powerful in the lab environment [see 17, 20, 133, 155], these approaches are both hard to scale to the large software projects we face today, and equally hard to integrate into the code auditing process [58]. Moreover, their success ultimately depends on exact modeling of programming language semantics, including effects dependent on the execution environment and compiler. Considering the complex composition of technologies and languages in today's systems, this is a daunting task. In addition, few methods developed in academic research strive to assist analysts in their work, and rather aim for full automation, a considerably more difficult, and possibly hopeless task. Overall, it may not come as a surprise that results of academic work in the area play only a limited role in real-world vulnerability identification to date [58, 168].

This work presents *pattern-based vulnerability discovery*, a contrasting approach for the identification of vulnerabilities that employs robust and inherently inexact pattern recognition and machine learning techniques to augment the analyst's abilities in day-to-day auditing rather than trying to replace her. We thus deliberately depart from exact analysis and instead adopt an engineering perspective to view vulnerability identification as a problem involving, metaphorically speaking, the discovery of a signal present in a noisy data stream. This change in illumination is performed to find how computers can assist analysts in settings where the sheer amount of code prohibit exact analysis due to lack of time and resources. Arguably, these settings constitute the rule rather than the exception. In this first chapter, we briefly introduce the reader to the problem of vulnerability discovery, as well as the techniques offered by machine learning that we leverage to assist analysts in this tedious process. With this background information at hand, we proceed to introduce *pattern-based vulnerability discovery*, the approach presented in this thesis. Finally, we give an overview of the contributions made and take the reader on a quick tour of the remaining chapters.

1.1 Vulnerability Discovery

We begin by introducing the reader to vulnerability discovery, the task that all work presented in this thesis ultimately aims to simplify. To this end, we first briefly introduce the concept of vulnerabilities and discuss their manifestation by example.

1.1.1 What are Vulnerabilities?

The Internet Security Glossary (IETF RFC 4949) [136, page 333] defines a vulnerability to be *a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy*, and goes on to state that *a system can have three types of vulnerabilities: (a) vulnerabilities in design or specification; (b) vulnerabilities in implementation; (c) vulnerabilities in operation and management.*

In this work, we focus on vulnerabilities in software, and hence, the systems considered are software systems. In addition, we exclude vulnerabilities in operation and management from our analysis to concentrate entirely on those flaws visible and fixable in program code, that is, vulnerabilities in a program's design and implementation. With these restrictions in mind, we note that vulnerabilities are defined to be a subset of flaws, making clear that discovering flaws can be considered a first step in vulnerability discovery. However, narrowing in on those flaws that violate security policies is equally important. Therefore, in stark contrast to methods for the discovery of defects developed in software engineering [e.g., 17, 53, 84], the focus lies on identifying flaws that are highly probable to provide the attacker with a certain gain, and that can in fact be triggered by an attacker.

Finally, the definition given by the Internet Security Glossary relies on that of *security policies*. The corresponding definition, however, is lengthy and abstract, making clear in particular that whether a flaw qualifies as a vulnerability or not is highly dependent on the program and its role in an organization. To focus on vulnerabilities typical for many programs, we therefore adopt an attacker perspective, resisting the temptation of defining security policies for our targets in detail, and instead, restricting ourselves to security policies that are a reasonable minimum for most systems. In particular, we consider the following policies.

- **Code execution.** An attacker capable of providing input data to be processed by the target program should not gain the ability to execute arbitrary code in the context of the program. For example, the client of a web server should not be able to execute arbitrary code in the context of the web server, and the user of a system call should not be able to execute code with kernel privileges.
- **Information disclosure.** Second, attackers should not gain the ability to read information the system does not intend to expose to them. For example, the attacker should not be able to read raw memory from a remote program or download arbitrary files from the system.
- **Denial of service.** It should not be possible for an attacker to terminate the execution of a system running on a remote host or serving multiple users. An exception is made for administrative users. As an example, it should not be possible for the user of an instant messenger to shut down the messaging server, nor should a non-privileged user of an operating system be able to crash the kernel.

While vulnerabilities constitute a non-trivial property of code, and thus, finding a general and effective procedure to detect these flaws is not possible as expressed in Rice's Theorem [see 115], we can at least provide a model that captures important properties of potentially vulnerable code, as we do in the following.

1.1.2 Characterizing Vulnerable Code

The vast majority of defects in code are not security relevant, and therefore, they are not vulnerabilities. To design methods specifically to assist in the discovery of vulnerabilities therefore requires us to identify those properties of code typical for these specific types of defects. We identify the following key properties of code that apply to a large number of vulnerabilities plaguing software today.

- **Sensitive operation.** First, a vulnerability must enable an attacker to carry out a sensitive operation with the potential of enabling her to violate a security policy, whether explicitly given, or made apparent only by successful attack. For example, reading a file from a system's hard disk is a sensitive operation.
- **Attacker control.** Second, an attacker must be able to trigger a vulnerability, that is, she needs to be able to provide input or influence the environment such that the sensitive operation is executed. With respect to our example, this may amount to the ability to control the file to read from the hard disk.
- **Insufficient validation.** Finally, the vulnerability must enable the attacker to actually cause a violation of the security policy by failing to restrict how the sensitive operation can be leveraged. In our example, a vulnerability may exist if the file-read operation can be used to extract arbitrary files from the system, but may not exist if only files from a certain folder can be read, designated to hold only non-sensitive information.

This model is influenced by work on taint analysis [see 88, 104, 132], and is generic enough to capture many types of vulnerabilities, including those typical for Web applications, but also many types of memory corruption vulnerabilities found in system code.

```
1  // src/misc/update.c
2  static bool GetUpdateFile( update_t *p_update )
3  {
4      stream_t *p_stream = NULL;
5      char *psz_version_line = NULL;
6      char *psz_update_data = NULL;
7
8      p_stream = stream_UrlNew( p_update->p_libvlc, UPDATE_VLC_STATUS_URL );
9      if( !p_stream )
10     {
11         msg_Err( p_update->p_libvlc, "Failed to open %s for reading",
12                  UPDATE_VLC_STATUS_URL );
13         goto error;
14     }
15
16     const int64_t i_read = stream_Size( p_stream );
17     psz_update_data = malloc( i_read + 1 ); /* terminating '\0' */
18     if( !psz_update_data )
19         goto error;
20
21     if( stream_Read( p_stream, psz_update_data, i_read ) != i_read )
22     {
23         msg_Err( p_update->p_libvlc, "Couldn't download update file %s",
24                  UPDATE_VLC_STATUS_URL );
25         goto error;
26     }
27     psz_update_data[i_read] = '\0';
28
29     stream_Delete( p_stream );
30     p_stream = NULL;
31
32     // [...]
33 }
```

FIGURE 1.1: Remote code execution vulnerability in the updater of the popular media player VLC.

As an example, Figure 1.1 shows a memory corruption vulnerability in an excerpt of the automatic updater of the popular VLC media player[1] (version 2.1.5), uncovered by

[1] http://www.videolan.org/vlc/

the author as part of this research. The listing shows around 30 lines of code of a total of around 556,000 lines of code in VLC. Within these 30 lines, the program reads the alleged size of the attacker-controlled data stream into the 64 bit integer `i_read` on line 16, and proceeds to call the allocation routine `malloc` with the argument `i_read + 1` on the next line. As on line 21, attacker-controlled data of up to `i_read` byte is copied into the so allocated buffer, it must be able to hold at least `i_read` bytes to not cause the buffer to overflow, and it first sight, the code seems to ensure this.

However, it has not been considered that the argument passed to the allocation routine `malloc` is of type `size_t`, which is only 32 bit wide on 32 bit platforms. Thus, if `i_read` is set to be $2^{32} - 1$, `i_read + 1` will be 2^{32}, a number that cannot be stored in a 32 bit integer. To handle this condition, a truncation is performed, such that the amount of memory requested for allocation is in fact zero bytes. The result is a buffer overflow that has been proven by the author to allow for arbitrary code execution even with modern mitigation techniques enabled [see 96, 143]. In this example, attacker control is established by allowing her to specify the alleged size of the input stream. The sensitive operation is a copy-operation, and the validation of input is insufficient as more bytes can be copied into the buffer than it is able to hold. In particular, this allows the attacker to execute arbitrary code, a violation of a security policy.

As we show in this work, we can mine for vulnerabilities conforming to the abstract description given thus far. In fact, we can even automatically extract descriptions for common combinations of attacker-controlled sources, sensitive sinks, and the associated validation. In addition, it is often possible to deal with cases where identifying both the attacker-controlled source and the corresponding sink is difficult, but missing security checks tied to the source or sink alone can be identified.

1.2 Machine Learning

Ultimately, we are interested in obtaining programs that help narrow in on potential vulnerabilities by exploiting patterns in code. Machine learning methods [see 16, 37, 57] provide a powerful tool in this setting. In fact, the fundamental idea connecting all of these methods is that of automatically generating programs from data [34], making them a natural fit for our problem. For example, one may wish to devise a program capable of determining for an arbitrary fruit whether it is more likely to be a pear or an apple. One possibility to achieve this is by writing a set of manual rules, e.g., if the fruit is red and round, it is an apple. A more elegant solution is to induce such theories automatically by examining a basket of examples, and subsequently creating a *model* that encodes the properties of apples and pears. This model can be used to instantiate a predictor that implements a generic decision rule as a function of the model. For example, the predictor may compare the input fruit's properties to those of apples and pears as specified by the model, and decide in favor of the fruit type that shares the larger number of properties. Figure 1.2 illustrates this process.

Applied to vulnerability discovery, we are interested in generating programs that determine for arbitrary constructs of code how likely it is that they are vulnerable. We do this by inferring models for vulnerable code *from code*, and in particular, from samples of vulnerable and non-vulnerable code.

FIGURE 1.2: Conceptual view on machine learning

Formally, we can express the predictor we seek as a *prediction function* $f : \mathcal{X} \to \mathcal{Y}$ where $\mathcal{X}$ is the set of program constructs and, $\mathcal{Y}$ is an output space. For example, for $x \in \mathcal{X}$, $f(x)$ may quantify the likelihood that x implements a vulnerability as a real number between 0 and 1, in which case the output space is $[0, 1]$. Alternatively, we can also choose the output space to be a finite set of numbers, e.g., $\mathcal{Y} = \{0, 1\}$, where 0 and 1 indicate non-vulnerable and vulnerable code respectively.

As previously mentioned, instead of finding the prediction function f directly, we approach this problem by generating a model θ that encodes the properties of vulnerable code, and choosing f to be given by f_θ, a member of a function family parametrized by θ. In this formulation, machine learning is an optimization problem where we seek a model θ from a (possibly infinite) model space Ω, with minimum *cost* according to a cost function $E : \Omega \to \mathbb{R}$.

1.2.1 Settings in Machine Learning

While the concrete mechanics of finding an optimal model are dependent on the machine learning method, in all cases, these procedures take into account sample data. At this point, two settings can be roughly distinguished in terms of the available data: the *supervised* and the *unsupervised* setting. In the supervised setting, we have access to *labeled* data, that is, a set of data points from the input domain $\mathcal{X}$ along with the desired output values from $\mathcal{Y}$. In contrast, in the unsupervised setting, only samples from the input domain $\mathcal{X}$ are available *without* corresponding labels.

Informally, supervised and unsupervised learning methods address two diverging objectives. On the one hand, supervised learning focuses on approximating the relation between the input space and the output space. On the other, since in the unsupervised setting, no knowledge of correct output values for given input values is available, unsupervised learning instead focuses on uncovering structure in the input space, using the output domain to express it.

For example, based on labeled apples and pears, a supervised algorithm may determine that shape is a good indicator to distinguish the two categories. An unsupervised algorithm does not have access to these labels, however, it can still determine that there seem to be two groups of objects that can be distinguished by shape, and indicate this structure by generating a predictor that maps objects of these groups to labels indicating their group membership.

While the idea of teaching a computer program to distinguish vulnerable and non-vulnerable code from *labeled* examples seems intriguing, it heavily relies on providing good examples of vulnerable code. Clearly, creating these examples is labor-intense and worse, many types of vulnerabilities are very program specific, relying on the concrete programming environment, application programming interfaces, and finally, the program's exposure to attacker-controlled input. Particularly, when considering the limited

time frame available for external security analysts when reviewing code for vulnerabilities, a method relying on carefully labeled, application-specific samples of vulnerable code is of little use in practice.

Therefore, our focus in this work is on methods, which are largely unsupervised, and exploit the structure of the code as-is to narrow in on vulnerabilities while requiring very little information from the analyst. These algorithms allow us to identify latent patterns in the data, providing us with means to find compact representations, point out anomalies, or group data points.

1.2.2 Unsupervised Methods

While there are a multitude of different unsupervised learning algorithms, most address one or more of the following three core problems.

- **Dimensionality Reduction.** These techniques can be used to find expressive features for a data set, denoise the data, and obtain a more compact representation of it. To this end, dependencies in the data are exploited to project it into a lower dimensional space where some of the data's properties are no longer considered, while others are preserved. In the context of vulnerability discovery, these techniques can be used to extract programming patterns and enable searching for functions employing similar programming patterns (see Chapter 4), and as a pre-processing step for subsequent anomaly detection and clustering.
- **Anomaly Detection.** Unsupervised algorithms for anomaly detection allow deviations from patterns in the data to be detected. This is achieved by calculating a model of normality for the data, or a sub set of the data points, and subsequently measuring the difference of individual data points to the model. In the context of vulnerability discovery, this allows us to identify unusual fragments of code that deviate considerably from an identified pattern. In particular, we review a successful application of anomaly detection for the identification of missing an anomalous checks in Chapter 5.
- **Clustering.** Finally, with clustering algorithms, data points can be grouped into so-called *clusters* according to their similarity. This can be useful in code analysis and vulnerability discovery in particular, to obtain summaries of the code base contents, that is, to decompose the code base into sets of similar code, thereby allowing entire groups of code to be excluded from analysis. Moreover, clustering is a first step towards signature generation. We highlight this application in Chapter 6, where descriptions are generated from clusters of code fragments.

With the necessary background on machine learning algorithms and vulnerability discovery, we are now ready to introduce the core idea that connects the methods proposed in this thesis: pattern-based vulnerability discovery.

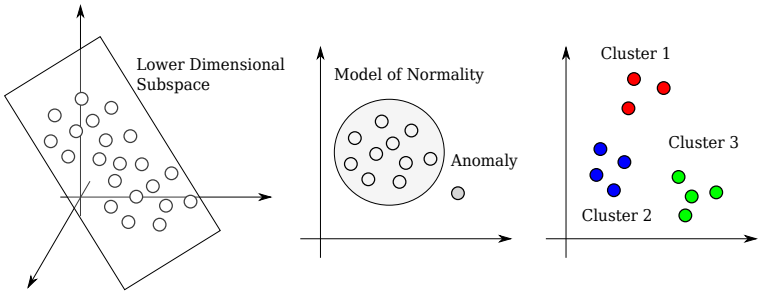


FIGURE 1.3: Schematic depiction of the three main techniques of unsupervised learning: dimensionality reduction (left), anomaly detection (middle), and clustering (right)

1.3 Pattern-Based Vulnerability Discovery

Precise methods of program analysis such as model checking and symbolic execution provide us with means to reason about confined regions of code in great detail. However, given the sheer mass of code that requires analysis to ensure secure operation of our computing infrastructure, and the vast amount of possible program properties to check, this ability is arguably secondary, in comparison to that of identifying interesting locations in the code for detailed inspection in the first place.

Taking on this perspective, vulnerability discovery becomes a search problem at scale, requiring us to expose needles in a haystack, with the additional difficulty that we cannot specify exactly what a needle looks like in advance. The idea of turning towards techniques for exploratory data analysis as provided by pattern recognition and machine learning, to assist in the discovery of vulnerabilities therefore arises naturally. *Pattern-based vulnerability discovery* implements this idea. We use the term to collectively describe methods that leverage patterns in code to narrow in on potential vulnerabilities. These patterns may be formulated by the analyst based on domain knowledge, derived from external data such as vulnerability histories, or inferred from the code directly.

Machine learning plays a crucial role for pattern-based vulnerability discovery. While these methods lack a deeper understanding of program semantics, they easily outperform the analyst when tasked with identifying patterns in large amounts of data, both in terms of speed and precision. However, we do not strive to fully automate the discovery of vulnerabilities using these methods, and instead employ them to augment the analyst's abilities rather than trying to replace her. We thus seek to optimally combine the strengths of the analyst and the machine, and in particular, allow the analyst to guide program exploration, and make final security critical decisions. Machine learning thereby becomes an *assistant technology* useful in different phases of the analysis.

The challenges for the design of pattern-based techniques for vulnerability discovery are threefold. First, tasks of the auditing process need to be identified that call for tedious manual analysis, and would benefit from pattern recognition. Second, a suitable interface needs to be provided to allow the analyst to interact easily with the tool. In particular, this interface should make the identified patterns explicit, to allow the analyst to judge, and possibly refine the results produced by the learner. Finally, efficient data structures and storage mechanisms need to be identified to allow the learning-based methods to execute in short time frames on commodity hardware, enabling the analyst to interact with the system during the auditing process.

1.4 Thesis Contribution

In this thesis, we explore how unsupervised machine learning techniques can assist in vulnerability discovery. Driven by common scenarios encountered in day-to-day auditing of source code, we propose different methods to augment the analyst's abilities. These methods not only share conceptual similarities but are also based on a common underlying framework for robust code analysis. In summary, the following key contributions make this possible.

- **An architecture for robust code analysis.** We present a novel architecture for robust code analysis, and pattern-based vulnerability discovery in particular. This architecture combines a novel parsing strategy (*refinement parsing*), a joint data structure of program syntax, control flow and data flow referred to as a *code property graph*, and a storage mechanism based on graph databases. In addition to allowing code to be mined for vulnerabilities using concise descriptions encoded as graph database queries, it forms the technical basis for all methods of vulnerability discovery presented in this thesis (Chapter 2).

- **Embedding of source code in vector spaces.** We proceed to develop several different feature maps to embed source code in vector spaces and thus enable it to be processed using machine learning algorithms. Moreover, we present a generic procedure to embed source code represented by code property graphs. This procedure plays a central role in all of the methods for vulnerability discovery presented in this thesis and thus connects our methods conceptually (Chapter 3).

- **Mining for instances of vulnerable programming patterns.** Based on the presented architecture for robust code analysis and the generic embedding procedure, we develop a method for discovering instances of programming patterns related to a known vulnerability. To this end, we employ dimensionality reduction to analyze code in terms of syntactical patterns, similar to the way latent semantic analysis finds text documents dealing with similar topics (Chapter 4).

- **Mining for deviations from inferred programming patterns.** We proceed to explore how our method for finding syntactically similar code can be extended to narrow in on vulnerable code by pointing out deviations from programming patterns via anomaly detection. Based on this idea, we derive a novel method for the identification of missing checks in source code and demonstrate its ability to assist in the discovery of missing security critical checks in particular (Chapter 5).

- **Explicit representation of programming patterns.** Finally, we show how search patterns that describe *taint-style vulnerabilities* can be automatically extracted from source code using clustering techniques. In these special types of missing check vulnerabilities, attacker controlled data is propagated to a sensitive operation without undergoing prior validation, a description that matches many high impact vulnerabilities as we show (Chapter 6).

Supporting source code for these contributions have been made available as open-source. This is particularly true for the robust code analysis platform *Joern* developed during this work, which has been made use of by security professionals in code auditing and enabled further scientific research on decompilation [161] and authorship attribution [19].

1.5 Thesis Organization

This thesis consists of seven chapters, six of which remain. The first two chapters provide the technical and methodological basis for vulnerability discovery via pattern-based techniques. It is therefore suggested to read these two chapters first. The following three chapters present concrete methods for pattern-based vulnerability discovery, each with a focus on one of the three primary problems addressed by unsupervised machine learning. These chapters can be read in arbitrary order, however, they are ordered such that the presented methods gradually increase in sophistication. Thus, if in doubt, reading chapters one after another assures the safest journey. The last chapter concludes.

Chapter 2 introduces our platform for robust code analysis along with background information on the techniques from compiler construction and graph mining this work is based on. Moreover, it introduces the code property graph, the primary data structure employed for robust code analysis in the remainder of this work.

Chapter 3 deals with the problem of embedding source code in vector spaces, a prerequisite for the application of machine learning algorithms for code analysis. In particular, we discuss a general procedure to embed code property graphs in vector spaces, which is instantiated by all methods presented in the remaining chapters.

Chapter 4 This chapter deals with the application of dimensionality reduction techniques to vulnerability discovery. In particular, we present a method to automatically extract programming patterns from source code and identify vulnerabilities similar to a known vulnerability. We implement this method based on the code analysis platform presented in Chapter 2 and by employing the embedding procedure developed in Chapter 3.

Chapter 5 We proceed to explore potential applications of anomaly detection to vulnerability discovery and present a method to uncover missing security critical checks in source code automatically. This second method makes use of the method presented in the previous Chapter but extends it to consider deviations from programming patterns.

Chapter 6 Finally, we present a method for learning explicit representations for vulnerabilities given in the form of database queries for our code analysis platform. This method hinges on clustering algorithms, and thus, it presents an application for the last of the three major types of unsupervised learning algorithms.

Chapter 7 In this final chapter, the presented work is summarized, its limitations are discussed, and conclusions are drawn. We close by discussing possible directions for future research in the area.

2 Graph Mining for Vulnerability Discovery

Discovering vulnerabilities in source code by exploiting meaningful patterns requires a comprehensive and feature rich representation of code. We cannot expect a system to learn these patterns automatically if its view on the code does not permit them to be discovered in the first place. Before we can develop methods for vulnerability discovery based on machine learning, we therefore need to devise a suitable representation of code that at least allows us to manually express patterns linked to vulnerabilities. In addition, we need to ensure that this representation can be robustly extracted from code, and stored in a suitable way to make mining of large amounts of code possible even on commodity hardware.

As a solution, this chapter presents our *platform for robust source code analysis*, which serves as a foundation for our approach to pattern-based vulnerability discovery and all the concrete methods proposed in this thesis. In essence, this platform enables large amounts of code to be analyzed with respect to syntax, control flow and data flow, and mined using an extensible query language. To this end, we combine classic ideas from compiler construction, lesser known techniques for analyzing code robustly, and the emerging technology of graph databases.

Ultimately, this system enables analysts to characterize vulnerabilities as *traversals in a code property graph*, a joint representation of a program's syntax, control flow, and data flow. These traversals serve as *search patterns* and can be expressed as queries for the graph database system. We show that this approach amounts to a powerful tool for vulnerability discovery by manually crafting search patterns for different types of vulnerabilities and uncovering 18 previously unknown vulnerabilities in the source code of the Linux kernel, a mature and frequently audited code base. Moreover, it provides a loosely defined, flexible language for encoding patterns in code, an idea we further explore in Chapter 6, where search patterns are derived automatically from code.

We begin by providing a broad overview of our architecture (Section 2.1) and highlight its key components as well as the way in which analysts can interact with the platform. We will see that this requires us to adapt exact techniques from program analysis to perform in a setting where we need to reason under uncertainty. In particular, we discuss how source code can be parsed robustly and subsequently transformed into intermediate graph-based program representations (Section 2.2). We continue to show how these representations can be combined to create the core data structure for pattern recognition

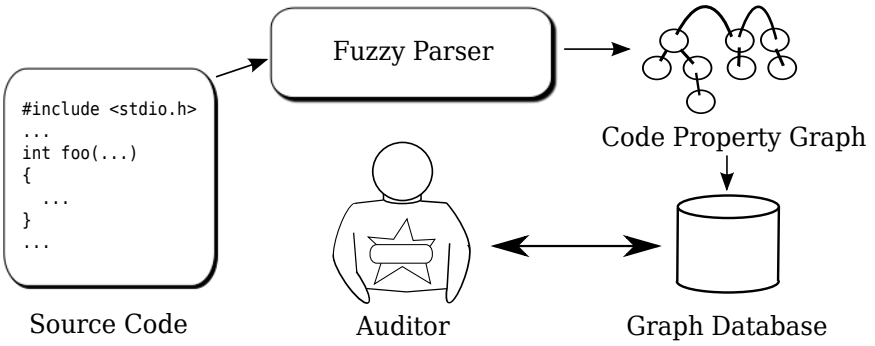


FIGURE 2.1: Overview of our architecture for robust code analysis

in large code bases as discussed in this work (Section 2.3). Finally, we illustrate how graph databases provide us with the machinery necessary to effectively store and process programs given in our intermediate representation (Section 2.4). Ultimately, this provides us with an environment for the development of the pattern-based approaches discussed in the remainder of the thesis.

2.1 A Code Mining System

Figure 2.1 gives an overview of the resulting architecture. In summary, it combines the following key techniques to augment the analyst's abilities.

- **Fuzzy Parsing.** The first step to robust code analysis is robust, best-effort parsing. We therefore perform approximate, *fuzzy* parsing of source code based on the concept of *refinement parsing* in order to allow analysis of code even when a working build environment cannot be configured, e.g., for incomplete code, legacy code or even fragments of code such as software patches (Section 2.2).
- **Code Property Graphs.** To allow complex patterns in code to be expressed that combine syntax, control flow and data flow properties, we employ a novel program representation, the *code property graph*. This representation can be easily constructed from the fuzzy parser output (Section 2.3).
- **Graph Databases.** Storing program representations of large software projects to make them accessible for pattern mining is challenging. To this end, our architecture makes use of graph databases, thereby allowing code property graphs to be queried interactively using expressive query languages (Section 2.4).

The analyst begins by passing source code to the fuzzy parser, which proceeds to generate a versatile intermediate representation of the code, the code property graph. This graph is then stored in a graph database, allowing the user to mine the code for vulnerabilities. Moreover, machine learning techniques for pattern recognition are implemented on the server side and can be leveraged by the analyst to support her analysis. In the following sections, we discuss each of these components in greater detail and provide the necessary background information where required.

2.2 Fuzzy Parsing

Automatic analysis of source code hinges on the availability of intermediate code representations that make program properties explicit. The compiler design and program analysis literature offer a wealth of such representations, created for different purposes. We ultimately seek to make as many of these representations accessible to the analyst as possible, in order to allow vulnerable code to be characterized using expressive and concise descriptions.

All of these representations are either directly or indirectly created from a program's *parse tree*, making the ability to properly parse source code beforehand a necessity. For compiled languages such as C or C++ in particular, compiler frontends can often be instrumented easily to achieve this. Unfortunately, while this approach is sufficient in the lab environment, it has major shortcomings that prohibit its application for robust code analysis. The main problem encountered is that compiler frontends are only capable of generating parse trees if the program's syntactical structure can be determined with absolute certainty. Unfortunately, for languages such as C or C++, this is only possible if it can be resolved whether identifiers refer to the name of a variable or that of a type [see 60, 73]. This is a reasonable requirement for code compilation, as source code can only be translated into machine code if it conforms to the language specification, however, it stands in stark contrast to the notion of *robust* code analysis as a single missing header file terminates the process.

The problem of parsing code with missing declarations has been previously studied by several authors, particularly in the field of reverse engineering [e.g., 10, 73, 75]. In contrast to the code compilation setting, in reverse engineering, one can assume that the code is syntactically correct, that is, there exists at least a single language dialect that the code conforms to. This assumption changes the role of the parser drastically. It is no longer necessary to check the code for syntactical correctness, instead, we are interested in determining as much of the code's structure as possible given incomplete information.

We solve this problem by developing a *fuzzy parser* based on a novel parsing called *refinement parsing* (see Section 2.2.1). In contrast to exact parsers, the parse trees generated by fuzzy parsers vary in the amount of detail they expose about program constructs depending on the parser's ability to recognize the code. However, as we will see in the following sections, useful representations can be generated from code even if not all details of the program are clear.

Figure 2.2 gives an overview of the representations we can generate based on the fuzzy parser output, and highlights their dependencies. We begin by analyzing program syntax

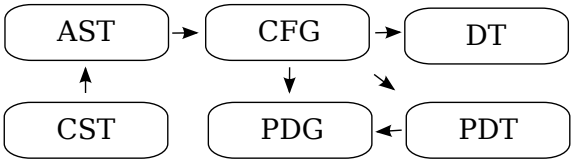


FIGURE 2.2: Dependencies between program representations.

by constructing the concrete syntax tree (CST) or parse tree (bottom left), which is first translated into an abstract syntax tree (AST) (upper left). To analyze the program's control flow, we generate a control flow graph (CFG) from the abstract syntax tree. Based on the information it contains, we can determine control- and data dependencies as expressed by the dominator tree (DT), the post-dominator tree (PDT), and finally, the program dependence graph (PDG), which is constructed by combining information from the control flow graph and the post dominator tree.

In the following, we discuss how syntax, control flow, and program dependencies are expressed by these representations and how they can be created from the fuzzy parser's output. However, before we do so, the concept of refinement parsing needs to be introduced in order to understand how parse trees can be generated even when code is only understood partially.

2.2.1 Refinement Parsing

The main idea of refinement parsing is to parse code in multiple stages, where each stage increases the level of detail considered. For example, in a first stage, only function and class definitions may be identified without parsing contents in detail. A second stage parser may then subsequently try to analyze function content in order to determine statements. The advantage of this strategy when parsing incomplete code is clear: while it may not always be possible to correctly parse every last detail of a function, we may at least be able to recognize its coarse structure, and if we can, refinement parsing will.

The exact parsing stages implemented by the refinement parser are language dependent. For the imperative languages C and C++, we implement the following three parsing stages as *island grammars* for the ANTRLv4 parser generator [106].

- **Module Level Parser.** The first stage parser only recognizes the coarse structure of a module, that is, grouping constructs such as functions, namespaces, and classes, as well as top-level variable declarations. Only little is demanded from the grouped contents namely that it is correctly nested, meaning that for each opening curly bracket, a corresponding closing curly bracket exists. To identify functions nested in class definitions or namespaces, we pass the contents of classes and namespaces to a new instance of the module-level parser, while function content is passed to the function parser.
- **Function Parser.** The function parser coarsely identifies program constructs that influence control flow within a function, which, in C/C++ are given by program statements. This includes jump statements such as *goto*, *continue*, and *break*, selection statements such as *if*-statements, and *switch*-statements, as well as iteration statements such as *for*-, *while*-, and *do*-loops. Analogously to the handling of function content by the module parser, the function parser demands little from statements, namely, that they are correctly terminated via semicolons or, in the case of conditions, correctly nested inside brackets.
- **Statement Parser.** Finally, the statement parser analyzes statements to decompose them into expressions, a prerequisite to reasoning about statement semantics in subsequent analysis. For example, we thus determine function calls for interprocedural analysis, as well as assignment operations for data-flow tracking. In

```
Code           =  [FunctionDef | Water]*;
FunctionDef    = ReturnType? FunctionName FunctionParamList CompoundStmt;
...
CompoundStmt   = "{" Content* "}";
Content        = ANY_TOKEN_BUT_CURLIES | CompoundStatement;
Water          = ANY_TOKEN;
```

FIGURE 2.3: An excerpt of an island grammar for recognition of C functions

practice, we have merged the grammars of the function parser and the statement parser, however, it is worth pointing out that control flow analysis can be performed based on a function parser alone.

Fuzzy parsers based on island grammars as proposed by Moonen [98] offer a particularly elegant solution to implementing parser stages. The main idea of his approach is captured in the definition of island grammars:

"*An island grammar is a grammar*[1] *that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water)*" [98].

This idea cannot be applied directly in our setting since we are actually interested in *all constructs* we can possibly recognize in the input stream, and hence, the island grammar degenerates into a full-fledged grammar in the limit. However, we can modify this idea slightly and instead create a chain of parsers, each of which focus on certain *aspects* of program constructs while being liberal with respect to all others. As an example, consider an island grammar for recognizing function definitions. Figure 2.3 shows its productions in Extended Backus-Nauer Form (EBNF), a standard notation for context free grammars [137].

The first line of this sample grammar already highlights that island grammars are ambiguous by definition. The production expresses that the designated start symbol `Code` may be expanded into a sequence of arbitrary length where each element is either a function definition (symbol `FunctionDef`) or `Water`. Since `Water` matches an arbitrary token (line 6), any production containing `Water` as one of its alternatives becomes ambiguous. This ambiguity is inherent to the approach and must be addressed by introducing external information, namely, that the non-terminal `Water` may only be considered once all other alternatives are known not to be applicable. Fortunately, the parser algorithm implemented by the ANTLRv4 parser generator resolves such ambiguities naturally, simply by matching non-terminals in the order, in which they are specified, and hence, the `Water` non-terminal is considered only if all other non-terminals cannot be matched. The production thereby implements a default "catch-all" rule.

Line 2 describes function definitions in detail, stating that it begins with an optional return type, followed by a mandatory function name and function parameter list, followed by a compound statement. However, in contrast to a complete grammar for C, only the bare minimum of requirements for compound statements are formulated (line 4), their correct nesting. This is achieved by defining function content to be a sequence of arbitrary length where each element is either an arbitrary non-curly token or another compound statement, guaranteeing that each opening curly is eventually closed before the final closing curly.

[1]The term *grammar* is used as a shorthand for *context-free grammar* here [see 1, chapter 2]

The advantage of using such island grammars for fuzzy parsing as compared to hand-written fuzzy parsers is (a) a clean and compact formulation of recognized language constructs and (b) enforced separation of parsing from all other components of the system[2]. A drawback is a possibly increased execution time when compared to hand-written parsers as language-specific optimizations are not as easy to introduce.

2.2.2 Exposing Program Syntax

Narrowing in on vulnerabilities is often already possible based on program syntax alone. In this context, syntax trees are a useful tool to characterize syntactical patterns, as they faithfully model how language constructs are nested and chained to form programs. Moreover, these trees are the direct output of the parser and hence, they form the basis for the creation of all other representations considered in this work. We now briefly illustrate how parse trees are generated by the parser and discuss their transformation into abstract syntax trees, a simplified and normalized syntax tree better suited for static analysis.

```
1  void foo()
2  {
3      int x = source();
4      if (x < MAX)
5      {
6          int y = 2 * x;
7          sink(y);
8      }
9  }
```

FIGURE 2.4: Running example of a code listing [162]

In the following, and throughout the rest of the chapter, let us consider the input program shown in Figure 2.4. While chosen to be as simple as possible, this example already allows the strengths and weaknesses of each representation to be made apparent. In particular, the example shows a function named `foo`, which reads input into a variable `x` by calling the function `source` (line 3). This variable is subsequently checked to be smaller than a constant `MAX` (line 4) before being used in an arithmetic calculation (line 6) and passed to the function `sink` (line 7).

2.2.2.1 Parse Trees

Concrete syntax trees, typically referred to simply as *parse trees*, can be easily generated as a by-product when parsing code according to a grammar, as for example, the grammar shown in Figure 2.5. This is achieved by executing the productions of the grammar to recognize the input, and creating a node for each encountered terminal and non-terminal. Connecting each node to that of its parent production, we obtain the desired tree structure.

As an example, Figure 2.6 shows the parse tree of the function `foo` obtained by applying our illustrative island grammar from Figure 2.5. The example shows that inner nodes and leaf nodes correspond to non-terminals and terminals respectively. Starting at

[2]This is a design in accordance with the principles of language-theoretic security [see 129], albeit the security of the analysis system is not of central concern in our work.

```
CompoundStatement  = "{" Stmt* "}";
Stmt               = CompoundStatement | Decl | IfBlock | ... | Water;
IfBlock            = "(" Pred, ")"Stmt;
...
Decl               = TypeName Expr;
Expr               = AssignExpr ["," Expr];
Water              = ANY_TOKEN;
```

FIGURE 2.5: Excerpt of an island grammar for parsing of C function contents (merged function and statement parser)

the root node (`Func`), the tree encodes which productions are executed to match the input program. In this case, a compound statement (`CompoundStatement`) consisting of an opening curly bracket followed by a declaration (`Decl`) and an if-block (`IfBlock`), followed by a closing curly bracket are matched. In particular, the example highlights that the input program is shown with no details omitted, even punctuation characters are preserved.

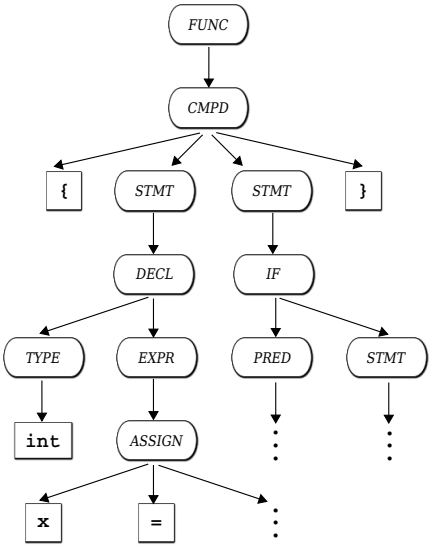


FIGURE 2.6: Parse tree for the running example

Applying this concept to generate parse trees from refinement parsers (see Section 2.2.1) is simple; each parser stage already generates a parse tree, which simply needs to be combined correctly with that generated by the previous state. For example, the module level parser generates a parse tree containing a designated node for function contents, however, only non-terminal are stored beneath this node, that is, the function content is not analyzed in detail. Upon executing the function parser, we can now simply replace the content node by the parse tree generated by the function parser, thereby obtaining a detailed representation of function content.

The parse tree is the only representation employed by our architecture that can be directly calculated from the source text and thus, it forms the basis for the generation of all other representations discussed throughout this section. However, the parse tree's verbosity and sensitivity to slight changes in program formulation are undesirable in the context of pattern recognition. We therefore immediately transform the parse tree into more robust representations of program syntax, the *abstract syntax tree* (AST).

2.2.2.2 Abstract Syntax Trees

In contrast to parse trees, abstract syntax trees omit details of program formulation that do not have an effect on the semantics of the program. For example, for program semantics, it is typically not relevant whether two variables are declared in a declaration list or using two consecutive declarations. While the concrete syntax trees differ, the abstract syntax trees are designed to be the same for both cases. Additionally, the abstract syntax tree usually does not contain punctuation symbols such as braces, semicolons or parentheses, as these are already implicitly encoded in the tree structure. Finally, the abstract syntax tree is often condensed by discarding inner nodes with a single non-terminal child node, making it a considerably more compact representation than the concrete syntax tree.

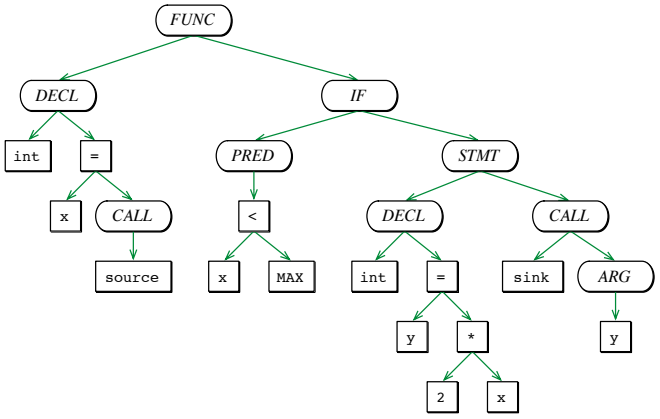


FIGURE 2.7: Abstract syntax tree for the sample listing. [162].

As an example, Figure 2.7 shows an abstract syntax tree for the function `foo`. In particular, we see that brackets present in the parse tree are no longer visible in the abstract syntax tree. Moreover, expressions have been collapsed as the call to `source` illustrates, which is directly connected to the assignment expression (denoted by an equality sign) as opposed to being connected to a chain of non-terminals that are traversed until the non-terminal `expr` is finally found to be a `call` expression by expansion.

Abstract syntax trees can be directly created from parse trees, and many parser generators offer built-in capabilities to achieve this [see 106]. In essence, this is achieved by defining translations of elementary parse trees into their corresponding abstract syntax trees, and recursively walking the parse tree. This operation is unaffected by our replacement of exact parsers by a fuzzy parsers, and hence, we do not discuss it in detail. It is noteworthy, however, that abstract syntax trees created by our fuzzy parser may contain terminal-nodes for water tokens, that is, tokens, which have not been recognized as parts of known language constructs.

2.2.3 Exposing Control Flow

The syntax tree allows all program statements to be easily located and examined, however, it is not well suited to study statement interplay. In particular, it does not allow to easily determine statement execution order, a key requirement for modeling of vulnerabilities. Where the abstract syntax tree highlights the code's syntactical structure,

the *control flow graph* (CFG) [2] exposes its *control flow*, i.e., the possible order in which statements may be executed and the conditions that must be met for this to happen. To this end, the control flow graph contains a node for each statement (both control- and non-control statements) as well as a designated entry and exit node. Transfer of control is indicated by directed edges between these nodes.

Figure 2.8 shows the control flow graph for the sample function `foo`. As is true for all control flow graphs, the graph contains an entry and an exit node denoted by `ENTRY` and `EXIT` respectively. Moreover, a node for each statement exist. In this case, there are three non-control statements, namely, the declarations of `x` and `y` and the call to `sink`. In addition, one control statement exists given by `if (x < MAX)`. Each non-control statement is connected to exactly one other node via an outgoing edge labeled as ϵ, while control statements have two outgoing edges labeled as `true` and `false` to indicate under which condition control is transferred to the destination block.

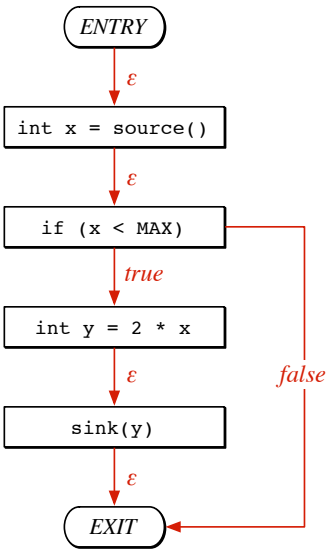


FIGURE 2.8: Control flow graph for the function `foo` [162].

Control flow graphs can be calculated directly from abstract syntax trees, however, it is necessary to provide information about all keywords the language offers to allow programmers to alter control flow, e.g., the keywords `if`, `for` or `goto`. Given this information, the following two step procedure translates abstract syntax trees into control flow graphs.

- **Structured control flow.** First, structured control flow statements such as `if`, `for` or `while` are handled to generate an initial version of the control flow graph. This can be easily achieved by defining for each structured control flow statement how the corresponding abstract syntax tree is converted into a control flow graph and recursively applying these rules to the complete abstract syntax tree.
- **Unstructured control flow.** Second, the control flow graph is corrected by considering unstructured control flow introduced by jump statements. Handling jump statements is easy after building the initial control flow graph as all loops, and hence the targets of `break` and `continue`, as well as all labels referenced by `goto` statements are now known. In effect, the second phase of the analysis simply

introduces additional control flow edges from jump statements to their targets, yielding the final control flow graph.

To calculate control flow graphs based on the abstract syntax trees generated by our fuzzy parser, we only slightly adapt this procedure to account for *water*. We simply create a statement node for each water token, which we connect via an incoming control flow edge to the last known statement, and via an outgoing edge to the next known statement. With control flow graphs at hand, we can now explore control- and data dependencies among statements.

2.2.4 Exposing Statement Dependencies

The availability of control flow graphs is a prerequisite for the generation of more specialized representations such as dominator trees and program dependence graphs, both of which play a key-role in our approaches to vulnerability discovery. *Data-flow analysis* forms the algorithmic basis for the creation of these representations. In the following, we briefly recapitulate essential data-flow analysis and proceed to discuss dominator trees, post-dominator trees, and program dependence graphs. For a more in-depth discussion, the reader is referred to Chapter 9 of the standard textbook on compiler design by Aho et al. [1] known as the "Dragon Book".

2.2.4.1 Data-Flow Analysis

Data-flow analysis offers a general framework for solving a number of related tasks in program analysis. To this end, determining data flow is formulated as an abstract problem, that, in essence, deals with finding *data flow values* for different points in the program, given a set of constraints that account for the structure of the CFG and the semantics of its statements. This abstract data flow problem is formulated as follows.

A program as represented by a control flow graph describes a (possibly unbounded) set of *paths*, i.e., sequences of statements that may be executed in order. As the program is executed, each statement finds the program to be in an input state, modifies this state to generate an output state, and passes this new state to the next statement on the path. This idea is expressed by modeling a statement as a program point *before* execution, where the program is in the input state, a *transfer function* that maps the input state to an output state, and finally, a program point *after* execution where the program has transitioned into the output state.

Formally, we can describe the data flow problem as follows. Let $p = [s_1, \ldots, s_n]$ denote a path in the control flow graph, i.e., a sequence of statements. Furthermore, for each statement s, let $\text{In}[s]$ and $\text{Out}[s]$ denote the set of data-flow values before and after statement execution respectively, following the notation in the classic Dragon Book on compiler construction [1]. Then, the transfer function f_s of s maps the input state to the output state, i.e., $\text{Out}[s] = f_s(\text{In}[s])$. This is true for all statements of the control flow graph, and it expresses the constraints imposed on our solution by statement semantics. Determining the data-flow value that reaches a statement s_n after executing a path to it can then simply be achieved by chaining the transfer functions of the individual statements, i.e., $\text{In}[s_n] = F_p(\text{In}[s_1])$ where $F_p = (f_{s_{n-1}} \circ f_{s_{n-2}} \circ \ldots \circ f_{s_1})$.

While this lays the foundation for reasoning about single paths in the control flow graph, to consider the effect all possible paths may have on the data-flow value of a statement, we need to find a suitable operation to combine the outgoing data-flow values of all preceding statements. The choice of this operator defines our notion of the abstract program state, and hence, it is problem-specific. In the abstract problem definition, we therefore merely demand that it is a binary operation denoted by the binary operator $\cup$, the so called confluence operator. For each statement s, the incoming data flow value is then defined to be equal to the combination of outgoing data-flow values produced by all of the paths leading to s, or formally

$$\mathrm{In}[s] = \bigcup_{p \in P_s} F_p(\mathrm{In}[s_1]) \tag{2.1}$$

where P_s is the set of paths to s and s_1 is the entry node. We thereby obtain additional constraints that model the structure of the control flow graph. It is important to understand in this context, that this so called *meet-over-all-paths solution* is already an approximation as it rests on the assumption that all paths in the control flow graph can actually be taken. Whether this is true is undecidable in general.

We have now fully defined an abstract data-flow problem that can instantiated by providing a set of possible data-flow values and initial values for each program point, transfer functions for each statement, and finally, a confluence operator. Unfortunately, the problem formulation does not immediately suggest an algorithm to obtain a solution. In particular, computing the left side of control flow constraints (Equation 2.1) from the right side is not possible for control flow graphs containing cycles, as the number of paths to consider is unbounded in this case.

In practice, the data-flow problem is usually solved by iterative approximation of data-flow values at each program point until an equilibrium is reached. Algorithm 1 expresses this idea in detail. In essence, this is the algorithm formulated by Aho et al. [1, Chp. 9] as adapted to our definition of the control flow graph.

Algorithm 1 Iterative Algorithm for a forward data-flow problem [1]

```
1: procedure DataFlow
2:     Initialize(Out[ENTRY])
3:     for v ∈ V_C \ {ENTRY} do
4:         Out[s] ← ⊤
5:     while (changes to any Out occur) do
6:         for s ∈ V_C \ {EXIT} do
7:             In[s] ← ⋃_{P a predecessor of s} Out[s]
8:             Out[s] ← f_s(In[s])
```

The algorithm proceeds by initializing the output state of the entry node (denoted by Out[ENTRY]) according to the problem definition. All other output states are initialized to be empty by assigning the data-flow value $\top$. Input values and output values are then recalculated for all nodes until no further changes in output values are observed. This algorithm is known to calculate the meet-over-all-path solution for the transfer functions and confluence operators used in many standard data flow problems [see 1, Chp. 9].

2.2.4.2 Dominator and Post-Dominator Trees

Many vulnerabilities arise from insufficient sanitization of user input, and thus the question naturally arises whether a sanitization routine is *always* executed before a sensitive operation. While this question involves analyzing control flow, the control flow graph is not immediately applicable to answer this question. The main problem is that the control flow graph tells us whether a statement *may* but not whether it *must* be executed before another. Fortunately, another representation can be calculated from the control flow graph that is well suited in this scenario without requiring additional semantics of the language to be clarified, the *dominator tree*.

As is true for control flow graphs, the dominator trees contain a node for each program statement. A directed edge connects one node to another if the source node *dominates*[3] the destination node in the control flow graph. A node s_2 dominates a node s_1 if all paths from the entry node to s_1 contain s_2. Similarly, a node s_2 *post-dominates* a node s_1 if all paths from s_1 to the exit node contain s_2. Finally, a node s_2 strictly dominates/post-dominates another node s_1 if the two nodes are unequal and s_2 dominates/post-dominates s_1.

While for a given node s_1, more than one dominator can exist, there is only one dominator s_2 that strictly dominates s_1 but does not strictly dominate any other dominator of s_1. This node is referred to as s_1's *immediate dominator*. Immediate post-dominance can be defined accordingly.

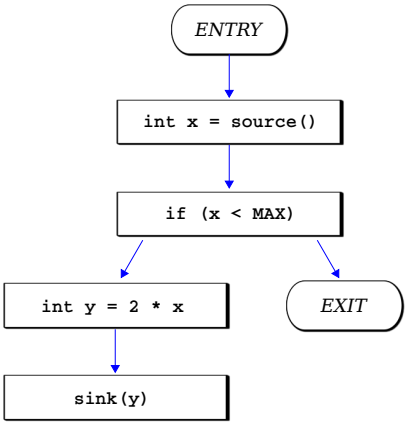


FIGURE 2.9: Dominator tree for the function `foo`

The definition of immediate dominance gives rise to the dominance tree, a hierarchical representation where each node is connected to its immediate dominator. As an example, consider Figure 2.9 showing the dominator tree for the function `foo`. We see that a node exists in the dominator tree for each of the control flow graph's nodes (see Figure 2.8), and that these are connected to form a tree rooted in the `entry` node. This is true for all dominator trees as the function entry is by definition not dominated by any other statement. While for the first three statements, the post dominator tree resembles the control flow graph, the outgoing edges of the `if`-statement differ. In particular, execution of the `exit` node need not be preceded by a call to `sink`. Instead, the last statement that must be executed before reaching the `exit` node is the `if` statement, and thus, the exit node is connected to the `if` statement to express this dominance relation.

[3]The notion of dominance was introduced in 1959 by Prosser [111]

Different algorithms for calculation of dominators have been proposed in the past [e.g., 25, 83]. In particular, determining dominators has been cast as a data-flow problem (see Section 2.2.4.1), allowing it to be solved using simple iterative schemes with a runtime quadratic in the number of control flow graph nodes. While Lengauer and Tarjan [83] show that lower asymptotic complexity can be achieved at the cost of a considerably more complicated algorithm, Cooper et al. [25] show empirically that, despite the theoretical disadvantage, careful choice of data structures allows simple data-flow approaches to perform favorably over Lengauer and Tarjan [83]'s algorithm in practice.

A simple algorithm to determine dominators using the data-flow approach (see Section 2.2.4.1) can be obtained by instantiating the data-flow problem as follows.

- Data-flow values are sets of nodes that dominate a program point, i.e., the domain of data-set values is given by the power-set of set of nodes V_C of the control flow graph. Moreover, Out$[s]$ is initialized to be V_C for all $s \in V_c$, i.e., we begin by assuming that a node is dominated by every other node, including itself.
- A successor may be dominated by any of a nodes dominators and by the node itself. Therefore, the transfer function is given by $f_s(x) = x \cup s$.
- A node s is dominated by a node $d \neq n$ if and only if d dominates all its predecessors. Choosing set-intersection as a confluence operator implements this rule.

As a result, we obtain the set of dominators Out$[s]$ for each statement s. While this highlights the core idea followed by all data-flow approaches to dominator calculation, several optimizations specific to dominator calculation can be implemented to achieve better performance in practice. For an in-depth discussion, we would like to refer the reader to the paper by Cooper et al. [25].

Once constructed, the dominator tree can be used to determine whether the execution of a statement s_2 is definitely preceded by the execution of another statement s_1. This is the case if s_1 dominates s_2. In particular, it can therefore be used to assure that a sanitization routine is always executed before a sensitive sink is reached.

2.2.4.3 Program Dependence Graphs

Finally, modeling data flow is vital to track how attacker-controlled data is propagated within the program. To this end, we employ the program dependence graph as introduced by Ferrante et al. [41]. While initially conceived to allow simple algorithms for program slicing [157] to be formulated, it has become a general purpose tool for the analysis of data flow and the effect of predicates on statement execution. Just like control flow graphs, program dependence graphs contain nodes for each statement and predicate and connect them to express their interplay. However, instead of making control flow explicit in its most basic form, two types of dependencies derivable from control flow are modeled, data and control dependencies.

- **Data dependencies.** Each program statement can *use* or *define* (i.e., modify) variables. Data dependencies indicate where values defined by a statement are used. This is the case if the destination statement uses a variable defined by

the source, and a path exists in the control flow graph from the source to the destination where none of the nodes on the path redefine the propagated variable. Determining these dependencies is a canonical problem of data-flow analysis known as *reaching definition analysis* [see 1, Chp. 9.2].

- **Control dependencies.** The execution of a statement may depend on the value of a predicate, for example, the call to `sink` in our example is only performed if `x < MAX` is true. Control dependencies indicate these kinds of dependencies. These dependencies can be calculated based on the control flow graph and the post-dominator tree.

As an example, Figure 2.10 shows the program dependence graph for the function `foo`. We see that it, just like the control flow graph, it contains a node for each program statement, however, from the program dependence graph, the exact order of statement execution can no longer be derived. Instead, we see edges from statements defining variables to users of these variables. For example, the variable x is defined by the top-most statement, and its values are used in the definition of y as well as the predicate. This predicate is itself connected to the sink-call by a control-dependence edge, indicating that the sink-call is only made if the predicate evaluates to true.

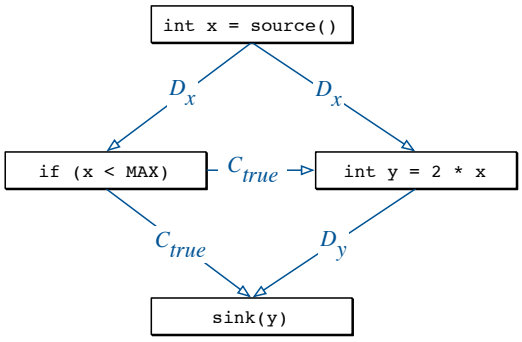


FIGURE 2.10: Program Dependence Graph of the sample function `foo` [162].

Both of these edges can be calculated from the control flow graph, and, in the case of control dependencies, the post-dominator tree.

Calculation of data dependencies. Calculating data dependencies can be achieved using the data-flow analysis framework discussed in Section 2.2.4.1. To this end, the abstract data-flow problem is instantiated as follows.

- Data-flow values represent the set definitions that reach the associated program point. The domain of data-flow values is thus given by the power set of all definitions. All data-flow-values are initialized to be the empty set.
- For each statement s, the transfer function f_s is given by $f_s = \text{gen}[s] \cup (\text{In}[s] \setminus \text{kill}[s])$ where $\text{gen}[s]$ is the set of definitions generated by the statement, and $\text{kill}[s]$ is the set of definitions killed by the statement. For example, if a statement defines the

variable x, it generates a definition for x and kills all definitions of x except for the generated definition.

- Set-union is used as a confluence operator, meaning that the definitions at a program point are equal to the union of the definitions of its predecessors. In effect, it is sufficient for a definition to reach a statement on one of the incoming control flow edges for it to be propagated.

As a result, we obtain a set of definitions for each program-point. In particular, for each statement, this gives us an approximation of the set of definitions that reach it.

Calculation of control dependencies. As shown by Cytron et al. [30], for a given node v, the nodes it is control dependent on are given by the so called *dominator frontier* of v in the reverse control flow graph. Intuitively, the dominance frontier of a node v is the set of nodes where v's dominance ends, that is, while these nodes are not dominated by v themselves, they are immediate successors of nodes dominated by v. Determining dominance frontiers requires a control flow graph and its corresponding dominator tree. As the dominator tree of the reverse control flow graph corresponds to the post dominator tree of the control flow graph, the calculation of control dependencies thus ultimately hinges on the availability of the post dominator tree.

While control dependencies can be calculated without introducing additional information about language semantics, data dependencies require for us to describe when a variable is *defined* and when it is used. For our robust code analysis platform, we simply consider a variable to be defined if it occurs on the left hand side of an assignment or as an argument to a function that is known to define its arguments. We consider it to be used if it appears in any other context.

2.3 Code Property Graphs

As the previous section shows, useful classic program representations can be generated from the output of a fuzzy parser, which opens up the possibility to leverage the information these structures contain for robust discovery of vulnerabilities. Unfortunately, it also becomes clear that none of these representations are suited to express all types of patterns in code equally well. While each representation highlights different aspects of the program, none can fully replace the others.

One cannot help but wonder whether it may be possible to merge these representations in order to obtain a single general structure that combines the strengths of its components. Not only would such a structure allow patterns to be analyzed that rely on combinations of syntax, control- and data flow, but it could also give rise to a general procedure for pattern recognition in static code representations, that is not tied to any particular representation (see Chapter 3).

We approach this problem by developing the *code property graph* [162, 165], a joint representation of syntax, control flow and data flow. Moreover, as an instance of a so-called *property graph*, this structure is ideally suited for storage and querying using graph database technology (see Section 2.4). In effect, the code property graph gradually advanced to become the core data structure of our platform for robust code analysis over the course of our work.

In this section, we first provide a definition of property graphs as an abstract data type, including basic operations important for the construction of code property graphs, and continue to discuss the notion of *traversals*, the primary tool for querying these graphs. In addition, we define *transformations* as a generalization of traversals that allow for modifications of the property graph as opposed to providing search functionality only. Finally, we show how the classic program representations introduced in the previous section can be formulated as instances of property graphs, and carefully merged to construct the code property graph.

2.3.1 Property Graphs

Graphs provide an intuitive tool to model data from a variety of different domains, including social networks, chemical compounds, or even, the structure of computer programs as we consider in this work. While in mathematical graph theory, the exact data represented by the graph is typically irrelevant, when considering graphs as a data structure for storage of information, encoding the actual meaning of nodes and the relationships edges represent becomes crucial.

In graph theory, a directed graph is typically defined to be a pair $G = (V, E)$, where V is a set of nodes and $E \subseteq (V \times V)$ is a set of edges, that is edges, are ordered pairs of nodes that represent the source and destination node respectively. This definition captures the notion of relationships between entities, however, it deliberately abstracts from the actual objects and types of relationships encoded by the graph to strip away unnecessary detail when reasoning about graph structure.

The definition of property graph extends the traditional graph definition to make the data stored in nodes and edges explicit as is required to store graphs in graph databases (see Section 2.4). This is achieved by (a) allowing key-value pairs to be attached to nodes and edges, and (b) allowing different types of relationships to be encoded in a single graph.

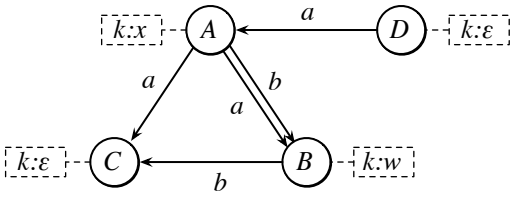


FIGURE 2.11: Example of a property graph [162]

Figure 2.11 shows an example of a property graph, highlighting its merits. The graph consists of four nodes connected by edges labeled as a and b, showing that the property graph can encode relationships of different type in a single graph. Moreover, the property graph is a multigraph, meaning that two nodes can be connected by more than a single edge, as is the case for the nodes A and B. Finally, nodes can *store* data. In the example, a value for the key k is stored in each of the nodes. This value may, however, also be empty as is the case for node C, which we denote by the symbol ϵ. These empty key-value pairs do not need to be stored in practice and merely simplify formal treatment, allowing us to express property graphs as follows.

Definition 2.1. A *property graph* $g = (V, E, \lambda, \mu, s, d)$ is a directed, edge-labeled, attributed multigraph where V is a set of nodes, and E is a set of directed edges where $s : E \mapsto V$ and $d : E \mapsto V$ assign source and destination nodes to edges respectively. Moreover, $\lambda : E \to \Sigma$ is an edge labeling function assigning a label from the alphabet Σ to each edge. Properties can be assigned to edges and nodes by the function $\mu : (V \cup E) \times K \to S$ where K is a set of property keys and S the set of property values.

By describing classical program representations in terms of property graphs, we allow them to be processed using graph database systems, a key idea implemented by our robust code analysis architecture.

2.3.2 Traversals for Property Graphs

Creating and storing property graphs is but a means to an end, and only becomes useful if methods to effectively retrieve data are provided alongside. The notion of traversals on property graphs provides this missing puzzle piece. Originally popularized by the query language Gremlin[4], and first formally defined by Rodriguez and Neubauer [124], they have become a fundamental tool for querying graph databases. We make heavy use of this formalism to obtain a framework for machine learning on static code representations that all methods for vulnerability discovery discussed in this work are based on (see Chapter 3).

Conceptually, a traversal is a program, which walks the property graph in order to test for the existence of sub structures of interest. Beginning with a set of start nodes or edges, the traversal sweeps over the property graph, taking into account the properties and labels to eventually terminate at a set of result nodes and edges. Formally, we can capture this idea in the following definition.

Definition 2.2. A *traversal* is a function $\mathcal{T} : \mathcal{P}(V \cup E) \to \mathcal{P}(V \cup E)$ that maps a set of nodes and edges to another set of nodes and edges according to a property graph g, where V and E are the node and edge sets of g respectively, and $\mathcal{P}$ denote the power set operation.

An important aspect of this definition is that the domain of traversals corresponds to its co-domain, and hence, traversals can be chained arbitrarily to yield new traversals. This makes it possibly to construct complex traversals that, for example, extract features from property graphs for machine learning tasks, by combining simpler traversals. To this end, we define a number of elementary traversals used throughout this work. We begin by defining the following traversal to filter graphs based on a predicate.

$$\textsc{Filter}_p(X) = \{x \in X : p(x)\}$$

This traversal returns all nodes and edges in the set X that match p, where p is a Boolean function. For example, p may be true for all nodes with a certain property and false for edges and all other nodes. For the special case of selecting all nodes with a property of a given key and value, we define the function Lookup as $\textsc{Lookup}(k, a, X) = \textsc{Filter}_p(X)$ where $p(x)$ is true if $\mu(x, k) = a$, and false otherwise. While the definition suggests that this lookup operation requires time linear in the number of nodes, in practice, graph

[4] `https://github.com/tinkerpop/gremlin/`

databases can create indices over node properties to allow these kinds of lookups to be carried out in constant time.

Upon selection of start nodes using a lookup operation, we can now move to the edges connecting them to their neighbors. To this end, we define the traversals

$$\begin{aligned}
\textsc{OutE}(X) &= \bigcup_{x \in X} \{e : e \in E \text{ and } s(e) = x\} \\
\textsc{OutE}_l(X) &= \bigcup_{x \in X} \{e : e \in E \text{ and } s(e) = x \text{ and } \lambda(e) = l\} \\
\textsc{OutE}_l^{k,a}(X) &= \bigcup_{x \in X} \{e : e \in E \text{ and } s(e) = x \text{ and } \lambda(e) = l \text{ and } \mu(e,k) = a\}
\end{aligned}$$

to determine outgoing edges of all nodes in a set X, where the first traversal returns all edges, the second returns only those edges carrying the label l, and the third additionally requires the edge property k to be a. Analogously, to determine incoming edges, we define the three traversals

$$\begin{aligned}
\textsc{InE}(X) &= \bigcup_{x \in X} \{e : e \in E \text{ and } d(e) = x\} \\
\textsc{InE}_l(X) &= \bigcup_{x \in X} \big\{e : e \in E \text{ and } d(e) = x \text{ and } \lambda(e) = l\big\} \\
\textsc{InE}_l^{k,a}(X) &= \bigcup_{x \in X} \big\{e : e \in E \text{ and } d(e) = x \text{ and } \lambda(e) = l \text{ and } \mu((e,k) = a\big\}
\end{aligned}$$

We refer to these elementary traversals that map nodes to a sub set of their edges as *expansions*. For each of these, we can define corresponding traversals that return neighboring nodes reachable via these edges. To this end, we introduce the traversals

$$\begin{aligned}
\textsc{VSrc}(X) &= \{s(e) : e \in X\}, \\
\textsc{VDst}(X) &= \{d(e) : e \in X\}
\end{aligned}$$

to obtain source nodes for a set of edges, and destination nodes respectively. These can be chained with expansions. For example, $\textsc{Out} = \textsc{VDst} \circ \textsc{OutE}$ obtains all nodes reachable via all outgoing edges by first determining outgoing edges and subsequently extracting their destination nodes. In general, for an expansion $\mathcal{E}$, the traversal that returns the corresponding set of reachable nodes $\hat{\mathcal{E}}$ is given by $\hat{\mathcal{E}}(X) = \bigcup_{x \in X} \bigcup_{e \in \mathcal{E}(x)} r(e,x)$ where

$$r(e,x) = \begin{cases} s(e) & \text{if } d(e) = x \\ d(e) & \text{otherwise} \end{cases}$$

that is, we simply expand each node $x \in X$ using $\mathcal{E}$ and return source nodes for edges where x is the destination node, and destination nodes for edges where x is the source node[5]. Finally, $\textsc{Not}(\mathcal{T})$ is given by the set of nodes in V that are not in $\mathcal{T}$, and we define the traversals

$$\begin{aligned}
\textsc{Or}(\mathcal{T}_1, \ldots, \mathcal{T}_N)(X) &= \mathcal{T}_1(X) \cup \ldots \cup \mathcal{T}_N(X) \\
\textsc{And}(\mathcal{T}_1, \ldots, \mathcal{T}_N)(X) &= \mathcal{T}_1(X) \cap \ldots \cap \mathcal{T}_N(X)
\end{aligned}$$

[5] In the case where an edge connects v to itself, this function returns v.

to logically combine the output of the traversals $\mathcal{T}_1$ to $\mathcal{T}_N$. With operations on property graphs and elementary traversals at hand, we now turn to the construction of the code property graph.

2.3.3 Constructing Code Property Graphs

Property graphs are a versatile data structure, which can be used to obtain intuitive representations of many types of data, and source code in particular. In the following, we define abstract syntax trees, control flow graphs and program dependence graphs as instances of property graphs, to finally merge them into a joint representation of syntax, control flow and data flow, the code property graph.

2.3.3.1 Transforming Classical Representations

We now proceed to formally define these representations as instances of property graphs, making it possibly to process them using graph database technology. We begin with the abstract syntax tree, the first of the representations to be generated from the parse tree. In terms of property graphs, we can define abstract syntax trees as follows.

Definition 2.3. An *abstract syntax tree* $g_A = (V_A, E_A, \lambda_A \mu_A, s_A, d_A)$ is a property graph where V_A is a set of nodes, E_A is a set of edges, and the labeling function λ_A labels all of these edges with the symbol $\mathcal{A}$ to indicate AST edges. Moreover, μ_A assigns the properties *code* and *order* to each node, such that the property value is the operator/operand and the child-number respectively [162]. Finally, we demand that the set S of property values contains at least the values `PRED` and `STMT` to denote control statements (*predicates*) and non-control statements respectively.

In contrast to typical definitions of the abstract syntax tree, our definition as a property graph binds nodes to the program text that they represent using the function μ_A and the key *code*. Moreover, we do not define syntax trees to be ordered trees as is typically done, and instead, account for order by annotating nodes with a child-number. Finally, the constraint imposed on S introduces the concept of statements at the syntax tree level, allowing us to make the relation to the control flow graph apparent.

Definition 2.4. A *control flow graph* is a property graph $g_C = (V_C, E_C, \lambda_C, \cdot, s_C, d_C)$, where the nodes V_C correspond to statements of the corresponding abstract syntax tree, that is, all nodes $v \in V_A$ where $\mu(v, \mathit{code})$ is *STMT* or *PRED*. Additionally, V_C contains a designated entry- and exit-node. E_C is a set of control flow edges, and finally, the edge labeling function λ_C assigns a label from the set $\Sigma_C = \{\mathit{true}, \mathit{false}, \epsilon\}$ to all edges of the property graph.

As is true for our definition of abstract syntax trees, this definition of control flow graphs deviates from standard definitions [e.g., 1, Chp. 8.4]. In contrast to standard definitions, our definition links control flow graphs to the abstract syntax trees they are generated from by highlighting that the set of nodes of the control flow graph is actually a subset of the nodes of the abstract syntax tree. This is essential for the construction of code property graphs, as we see in Section 2.3.3.2.

Finally, we can define the third data structure we base the code property graph on, the program dependence graph.

Definition 2.5. A *program dependence graph* of a function is a property graph $g_P = (V_P, E_P, \lambda_P, \mu_P, s_P, d_P)$ where the nodes V_P are the nodes of the corresponding CFG, E_P is a set of edges, and the edge labeling function $\lambda_P : E_p \to \Sigma_P$ assigns a value from the alphabet $\Sigma_P = \{C, D\}$ to each edge, where C and D denote control and data dependencies respectively. Additionally, the property *symbol* is assigned to each data dependence edge to indicate the propagated symbol, and the property *condition* is assigned to each control dependence edge to indicate the state of the originating predicate as *true* or *false* [162].

The crux of these definitions is that we define the three graph representations incrementally, thereby logically connecting the three representations. This ultimately makes it possible to merge the three representations as we explore in the following.

2.3.3.2 Merging Representations

The key observation that allows for the construction of code property graphs is that all classical program representations presented thus far are inherently linked, as they describe the same source code, albeit from different angles. It is therefore unsurprising that a language construct exists for which a corresponding node can be found in each of the representations, namely, the program statement. Realizing this correspondence, and by virtue of the definitions of classical program representations as property graphs given in the previous section, we obtain a joint representation by simply overlaying graphs. This is possible without causing confusion as each edge is labeled to indicate the relationship it expresses.

Figure 2.12[6] illustrates this idea. It shows the code property graph for the sample function `foo`, containing sub trees of the abstract syntax tree for each statement (green edges). Nodes of each statement are also present in the control flow graph (Figure 2.8) and the program dependence graph (Figure 2.10). We connect syntax trees by control flow edges (red) and dependence edges (blue) to indicate these relationships.

Formally, we can express the construction of code property graphs as follows. The code property graph contains all nodes of the corresponding abstract syntax tree as well as a designated entry and exit node. We define $V := V_A \cup \{\texttt{ENTRY}, \texttt{EXIT}\}$ to denote this set. Furthermore, we know that for each node in the control flow graph, there exists exactly one corresponding node in V, namely the node that represents the same statement, i.e., there exists an injection $f : V_C \to V$ with

- $f(\texttt{ENTRY}_C) := \texttt{ENTRY}$
- $f(\texttt{EXIT}_C) := \texttt{EXIT}$
- $f(v_c) = v$ for all $v_c \in V_C \setminus \{\texttt{ENTRY}, \texttt{EXIT}\}$, where v is the syntax-tree node for the statement associated with v_c.

Defining a corresponding function for nodes of the program dependence graph is not necessary, as, by Definition 2.5, the set of nodes of the program dependence graph V_P is equal to the set of nodes V_C of the control flow graph. In effect, f also fully illustrates the relationship between nodes of the program dependence graph and the code property graph.

[6] Syntax-tree edges above the statement level are omitted to improve presentation

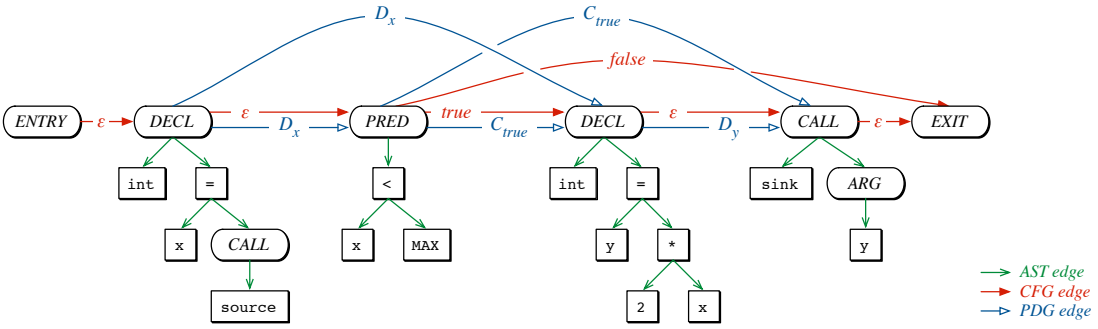


FIGURE 2.12: Code Property Graph for the function `foo` [162].

We proceed with the construction of edges for the code property graph. To this end, we can simply define the set of edges E to be given by $E := E_A \cup E_C \cup E_P$. This does not cause trouble even if two representations contain edges with the same endpoints, as E is the edge-set of a multigraph. These sets are best imagined as sets of identifiers or references, as opposed to sub-sets of all node-pairs typical for edge-sets in classical graph definitions. On the downside, we need to transfer sources, destinations and labels to edges of the code property graph by defining suitable functions s, d and λ. Fortunately, given the mapping f between CFG nodes and CPG nodes, this becomes easy. For example, we define s to be given by $s = s_A \cup s_{CFG} \cup s_{PDG}$ where

$$s_{\text{CFG}}(e) := f(s_C(e)) \text{ for all } e \in E_C, \text{ and}$$
$$s_{\text{PDG}}(e) := f(s_P(e)) \text{ for all } e \in E_P.$$

In other words, we employ f to map the start nodes of CFG and PDG edges to their corresponding nodes in the code property graph. This is not necessary for syntax edges as syntax nodes are fully contained in the set of CPG nodes. Similarly, we can define d, which associates endpoints with each edge, and the edge-labeling function λ.

Finally, the same procedure is employed transfer properties from the program dependence graph to the code property graph by defining μ to be given by $\mu := mu_A \cup \mu_{\text{PDG}}$ where $\mu_{\text{PDG}}(x, k) := f(\mu_P(x, k))$ for all $x \in (V_P \cup E_P)$ and all $k \in K_P$.

In summary, this leads to the following definition of the code property graph.

Definition 2.6. A *code property graph* is a property graph $g = (V, E, \lambda, \mu, s, d)$ constructed from the AST, CFG and PDG of a function, where $V = V_A \cup \{\texttt{ENTRY}, \texttt{EXIT}\}$, $E = E_A \cup E_C \cup E_P$, $\lambda = \lambda_A \cup \lambda_{\text{CFG}} \cup \lambda_{\text{PDG}}$, $\mu = \mu_A \cup \mu_{\text{PDG}}$, $s = s_A \cup s_{\text{CFG}} \cup s_{\text{PDG}}$, and finally, $d = d_A \cup d_{CFG} \cup d_{PDG}$.

The property graph thus combines abstract syntax trees, control flow graphs and program dependence graphs to make them available in a joint representation that encodes, syntax, control flow and data flow. Additionally, as the code property graph is a property graph, it can be directly stored in a graph database without modification.

The idea of merging representations at program statements is not limited to the three representations chosen for the code property graph and additional representations can be overlaid to extend the CPG's capabilities. However, unless otherwise noted, we are referring to this particular combination of structures when using the term *code property graph* throughout this thesis.

2.3.4 Extension for Interprocedural Analysis

The code property graph presented thus far only describes functions independently but fails to model their interplay. While this local view on the code is often already sufficient to discover critical vulnerabilities as we see in Section 2.5, interprocedural analysis allows us to uncover both patterns and vulnerabilities otherwise inaccessible. To conclude our discussion of code property graphs, we therefore discuss how our data structure can be extended for interprocedural analysis, an extension we make extensive use of in Chapter 6 for the generation of vulnerability descriptions for taint-style vulnerabilities (see Section 1.1.1).

To extend code property graphs for interprocedural analysis, we carry out a two-step procedure. We begin by introducing edges from arguments to parameters of their callees, and from return statements back to call sites, making data flow between call sites and callees explicit. The resulting preliminary graph already expresses call relations between functions. Unfortunately, it remains needlessly inexact as it does not account for modifications made by functions to their arguments, nor the effects these have as data flows back along call chains. We can therefore proceed to improve the preliminary graph by detecting argument modifications using post-dominator trees (see Section 2.2.4.2) and propagating this information through the graph to obtain the final interprocedural version of the code property graph. In the following, we describe this correction of data flow in greater detail.

2.3.4.1 Detecting Argument Modification

Data flow correction is bootstrapped by determining function calls that result in modifications of their arguments, that is, calls, which result in *definitions* of their arguments as picked up by reaching definition analysis (see Section 2.2.4.3). While for commonly used library functions such as `recv` and `read` from the POSIX standard, detection may not be necessary as these can be annotated to express argument modifications, this is not true for internal API functions such as the `n2s` macro that serves as a data source in the *Heartbleed* vulnerability.

For an arbitrary call to a library function, we therefore do not know whether it causes its arguments to be defined, and in effect, for all direct and indirect callers of library functions, the respective data flow edges in the code property graph may be incorrect. As an example, we consider the library functions `read` and `write` from the POSIX library: both receive a pointer to a memory buffer as their second argument. From the function signature alone, it is not possible to derive that `read` modifies the contents of the buffer while `write` does not, a vital difference that directly affects the data-flow edges of the code property graph.

We address this problem as follows. For each callee we cannot resolve, that is, each function called that comes without source code, we calculate a simple statistic based for each of its argument. We base this statistic on the following two checks.

1. First, we check whether a local variable declaration reaches the argument via data flow without passing through a easily recognizable initialization, e.g., a call to a constructor or an assignment.

2. Second, we ensure that the path from the function call to the local variable declaration in the post dominator tree does not contain another statement directly connected to the variable declaration via data flow.

We proceed to calculate the fraction of call sites that fulfill both conditions, and assume that the argument is defined by call to the function if this fraction is above a threshold. We fix this threshold to 10% in our experiments, a rather low value that expresses our preference of false positives over false negatives. Employing this simple heuristic, we recalculate the data flow edges of all affected functions.

```
int bar(int x, int y) {                    1
     int z;                                2
     boo(&z);                              3
     if (y < 10)                           4
       foo(x,y,&z);                        5
 }                                         6
                                           7
int boo(int *z) {                          8
     *z = get();                           9
}                                          10
```

FIGURE 2.13: Sample listing for argument definition [165]

The sample code shown in Figure 2.13 illustrates our heuristic for the detection of argument definition: the local variable z is declared on line 2 in the function `bar` without obvious initialization. A reference to `z` is subsequently passed as an argument to the functions `boo` on line 3 and `foo` on line 5. While it is reasonable to assume that the call to `boo` results in the definition of `z`, this is not true for the call to `foo` as it is called after `boo` that has already initialized `z`.

2.3.4.2 Propagation of Data-Flow Information

Argument detection as discussed in the previous section allows us to fix the code property graph of the library function's immediate caller, but corrections are not propagated along call chains to affect indirect callers. As an example, we reconsider the running example given in Figure 2.13. In this example, the argument `z` passed to the function `foo` is first defined on line 2 but redefined on line 9 inside the function `boo`. This results in a data flow from the source `get` to the function `foo`.

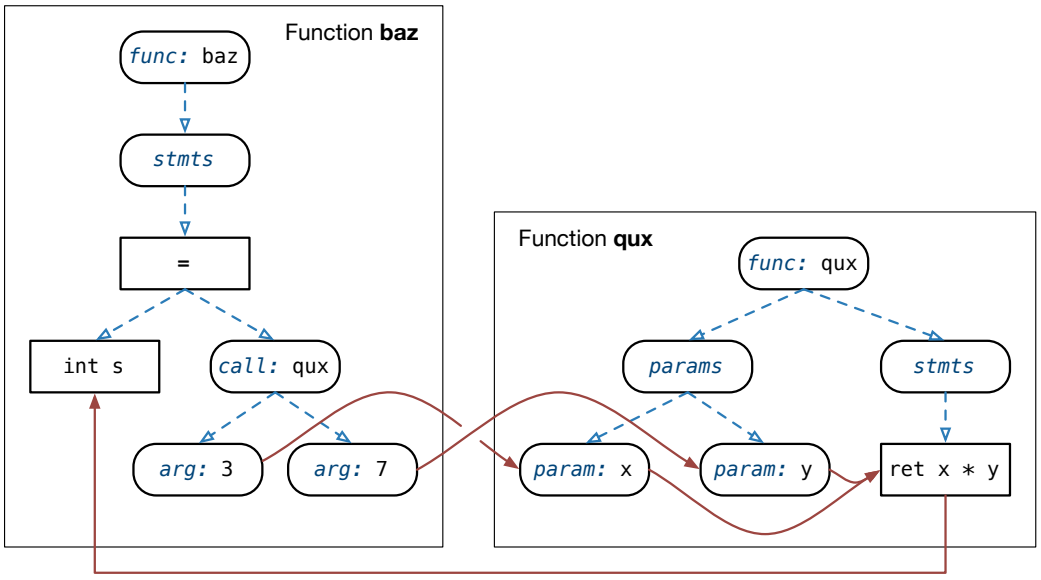
We take indirect argument definitions as seen in the example into account by propagating data-flow information along call chains. We achieve this by analyzing the source code of each available function to determine whether argument definition takes place by checking for each of its parameters whether (a) they are defined inside the function, and (b) the definition reaches the exit statement via control flow. However, this rests on the assumption that the data-flow edges of the function already take into account argument definitions performed by any of its callees. Therefore, before we analyze the function itself, we analyze all of its callees, and in particular, apply the heuristic for library function detection presented in the previous section.

The procedure FIXDATAFLOWEDGES shown in Algorithm 2 illustrates this idea. As an argument, it receives the set of function nodes of the code property graph. For each of these, it keeps a Boolean variable f_v that indicates whether the function has been fixed already. The procedure begins by initializing each f_v to `false`, and subsequently calls the recursive procedure FIXNODES for all function nodes. This procedure descends

Algorithm 2 Data flow recalculation

```
1:  procedure FixDataFlowEdges(V)
2:      for v ∈ V do
3:          f_v ← false                          ▷ Mark nodes as not fixed
4:      for v ∈ V do
5:          FixNode(v)
6:  procedure FixNode(v)
7:      if f_v = true then
8:          return false
9:      f_v ← true, u ← false
10:     for c ∈ Callees(v) do                    ▷ Fix all callees
11:         u ← u ∨ FixNode(c)
12:     if u = true then                         ▷ v needs to be updated
13:         UpdateDataFlow(v)
14:         return true
15:     return false
```

into the graph using a pre-order traversal, meaning that all callees are updated (line 11) before the current function is processed (line 13). Once completed, the resulting graph accounts for observable argument definitions as well as the resulting indirect data flows.



(A) Interprocedural code property graph.

```
1  void baz(void) {
2      int s = qux(3, 7);
3  }
```

```
4  int qux(int x , int y) {
5      return x * y;
6  }
```

(B) Code snippet of caller and callee

FIGURE 2.14: Interprocedural code property graph for the functions `baz` and `qux`. Syntax edges are shown as dotted lines and data-flow edges as solid lines [165].

As an example, Figure 2.14 shows a code snippet on the left and the resulting interprocedural version of the code property graph on the right. In particular, data flow edges between arguments and parameters, and from return statements back to variables of the caller are depicted in red. Control-flow edges are omitted to improve presentation.

2.4 Graph Databases

Code property graphs, as introduced in the previous section, offer a versatile and easily extensible representation of source code, however, it remains unclear how this data structure can be leveraged for vulnerability mining in practice. A key question in this context is how these large graphs can be efficiently stored and finally, exposed to the analyst via a flexible interface. The emerging storage technology of *graph databases* offers an elegant solution to this problem. Like document databases, object databases and key-value stores, these database systems depart from the *one-fits-all* paradigm of traditional relational database systems, and offer novel interfaces optimized for specific types of data. Graph databases, in particular, are designed with highly connected graph data in mind, and are a natural fit for graph-based program representations. However, stable and mature implementations of graph databases have just recently become available, and their use for program analysis has received little attention. In the following, we briefly introduce graph databases with a focus on program analysis and compare them to relational databases where necessary.

2.4.1 Graph Data-Models

Robinson et al. [123] define a *graph database* to be an *online database management system with Create, Read, Update, and Delete methods that expose a graph data-model*, that is, it is a system offering the basic operations expected from a database management system, however, to expose a graph data-model as opposed to e.g., a relational or object-oriented data model.

This definition is liberal in two ways. First, it does not dictate the format in which data must be stored, as long as it offers an interface that exposes a graph. In contrast, *native graph databases* additionally employ graphs as a primary storage format, implementing *index-free adjacency*, that is, nodes are directly connected to their neighbors making it possible to traverse to direct neighbors without requiring index lookups. Second, the definition allows any existing or future graph data-model to be used as the basis for graph databases. While in academic research, many different graph data-models have been developed in the 1980's and 1990's [see 5], the vast majority of modern graph databases are based on one of two data models unrelated to these efforts, *RDF triples* and *property graphs*.

- **RDF Graphs.** Resource Description Framework (RDF) Graphs are a representation of graph data designed for information interchange, and standardized by the World Wide Web Consortium (W3C) in a family of specifications [152]. Moreover, W3C provides a specification for an accompanying query language named SPARQL [153] for retrieval and manipulation of RDF graphs. RDF graphs are given by sets of triplets, where each triple consists of a subject, an object, and a predicate that links the subject to the object. To support interchange of these graphs, subject and object identifiers are typically not local to the graph, but are given by Internationalized Resource Identifiers (IRIs), which uniquely identify objects on the Web. Examples of graph databases based on RDF graphs are AllegroGraph, Bigdata, and Stardog.

- **Property Graphs.** In contrast to RDF graphs, Property graphs have not been standardized to date despite their appearance as a common data structure for a number of different graph database implementations. Property graphs are directed, attributed, edge-labeled multigraphs (see Section 2.3.1), a versatile data structure for storing graph data. This is the underlying graph data-model for many popular graph databases, including Neo4j, Titan, and OrientDB. While each of these introduce their own custom query language, the query language *Gremlin* can be employed for all of these databases via the Blueprints API [147], a common interface for databases implementing the property-graph model.

Hartig [55] points out that RDF graphs are more expressive than property graphs, as triples can be nested up to an arbitrary depth. For example, this allows properties to be assigned to entire graphs as opposed to single nodes and edges only. A clear advantage of Property graphs over RDF graphs is a much cleaner separation between node attributes and relationships as in RDF graphs, both attributes and relationships are expressed as subject-predicate-object triples. Moreover, formulating edge properties is clumsy in RDF, a problem, which has only recently been addressed by a proposed extension of RDF known as RDF* [56]. Finally, on a practical note, it is worth pointing out that converting between the two representations is often possible as outlined by Hartig [55]. We base our code analysis platform on the *Neo4j* graph database, a mature, open-source graph database developed by *Neo Technologies*. In the following, we therefore focus discussion on the property graph model.

2.4.2 Properties of Graph Databases

To date, relational database management systems in combination with the Structured Query Language (SQL) remain the predominant choice for most database applications. These systems allow efficient operations to be carried out on *relations*, that is tables containing data records. Moreover, they rest on solid theoretical foundations given by the relational data model [see 24]. Despite these merits, we found graph databases to be a better match for code analysis. The prime benefits are the following.

- **Match between conceptual view and storage format.** While it is possible to store graphs as tables in a relational database, it requires a suitable mapping from graphs to tables to be designed. This introduces a disconnect between the conceptual representation and the concrete structure in which it is stored, making it necessary to translate between the two representations efficiently or force the analyst into formulating queries with respect to the storage format. In contrast, when using graph databases the graph-based program representation can be stored as-is, keeping the conceptual view of the data aligned with the storage format.
- **Index-free adjacency.** Most implementation vulnerabilities are local properties of code, that manifest themselves in concrete programming errors located in a limited part of the program. Identifying these problems requires to locally explore the surroundings of candidate locations in the code, making efficient retrieval of neighboring nodes essential. In this setting, graph databases are at a conceptual advantage, as they have been specifically designed with this requirement in mind. While in relational databases, traversing to graph neighbors requires index-lookups proportional in runtime to the number of nodes of the graph, native graph

databases implement index-free adjacency, meaning that nodes are linked directly to their neighbors and can be reached in time independent of the total number of nodes. While performance evaluations of current implementations are rare and do not yet consistently highlight this advantage [see 12, 49, 150, 158], the conceptual advantage is clear.

- **Variety of query languages.** In theory, the relational model is not tied to any specific query language, however, in practical implementations available to researchers, it goes hand in hand with the Structured Query Language. Unfortunately, expressing even simple patterns in graphs using SQL can already lead to lengthy and complex queries. On the contrary, graph databases offer a variety of different query languages specifically designed for compactly expressing patterns in graphs (see Section 2.4.3).

The current state of flux query languages for graph databases are in plays into our cards, as it enables us to explore a variety of options for describing patterns in code. We now give a brief overview of the landscape of query languages and proceed to describe *Gremlin*, the query language of our choice.

2.4.3 Querying Graph Databases

Two competing approaches exist for querying modern graph databases. *Declarative* languages such as Neo4j's *Cypher* [144] and ArangoDB's *AQL* [31] are designed to closely resemble SQL in order to provide an intuitive way to interact with graph databases, particularly for users acquainted with typical relational database management systems. Like SQL, these languages are declarative in the sense that they only describe *what* to retrieve, but not *how*, leaving the latter to be decided by the database engine. In effect, the user is freed from the burden of designing a strategy to traverse the graph database, making these languages particularly suited for novice users. To additionally simplify querying, these languages focus on providing only a minimal set of language elements that allow simple patterns in property graphs to be described.

On the downside, whether a query can be executed efficiently or not depends highly on the concrete database engine, making it necessary to gain intimate knowledge of its implementation details to answer this question. Moreover, extending these declarative languages with user-provided language features is not trivial[7], as each new language element needs to be supported by the database engine, and it needs to be specified how it is executed when combined with existing language features. Finally, current declarative languages follow ad-hoc designs, based on no particular formalism or programming model. In consequence, no declarative query language supported across more than a single graph database system exists to date.

The largely imperative query language *Gremlin* is the most notable alternative to declarative languages for graph querying. In principle, Gremlin provides raw access to the property graph via the *Blueprints* API, a common interface implemented by all major open-source graph databases. This not only puts the user in full control over what

[7]While Cypher currently does not allow user-defined language extensions, AQL provides limited support for user-defined language elements by allowing so called *functions* written in Javascript to be provided. These are simply executed as-is when encountered by the database engine, thereby breaking out of the declarative paradigm.

to retrieve, but also *how* to retrieve it, making it possible to craft queries that easily outperform corresponding declarative queries by an order of magnitude [62]. Moreover, while arbitrary code written in the programming language Groovy can be used to retrieve data, Gremlin also provides a programming model to express traversals along with useful shorthands for common operations on property graphs. Finally, it allows users to create their own shorthands, making it possible to construct domain specific query languages.

On the downside, Gremlin is perceived by many users as more difficult to learn than declarative languages. Moreover, as Gremlin queries express both what to retrieve and how to retrieve it, they are often more lengthy than corresponding declarative queries.

Overall, we found the flexibility offered by Gremlin to be an important building block to enable mining for vulnerabilities using graph databases. Not only does it allow highly complex queries to be expressed elegantly, it also introduces an abstract programming model to do so. The main idea it follows is to express traversals in graphs as chains of function evaluations (see Section 2.3.2) that subsequently limit the parts of the graph that remains to be explored. In effect, we can express traversals for vulnerability discovery in terms of functions in the mathematical sense, and thus obtain a clean formal description inspired by, but independent of Gremlin.

2.5 Mining for Vulnerabilities

In this work, the code property graph is primarily used as a data source for machine-learning based methods for vulnerability discovery. However, unless it is possible to manually specify meaningful patterns for vulnerabilities by means of code property graphs, it will not enable us to learn these patterns automatically either. In this section, we therefore evaluate whether traversals can be manually crafted for real-world vulnerabilities to assess the expressive power of code property graphs.

We evaluate the capabilities of code property graphs in two consecutive experiments. First, we review all vulnerabilities reported for the Linux kernel in 2012 to determine which types of common vulnerabilities we can model using traversals. Second, we assess the code property graph's merits in uncovering previously unknown vulnerabilities by crafting different traversals and reviewing the retrieved code for vulnerabilities.

2.5.1 Coverage Analysis

We begin our analysis by importing the source code of the Linux kernel (version 3.10-rc1) into the Neo4J 1.9.5 graph database. The kernel spans approximately 1.3 million lines of code, resulting in a code property graph with approximately 52 million nodes and 87 million edges. On a laptop computer with a 2.5 Ghz Intel core i5 CPU and 8 GB of main memory, this process takes 110 minutes, producing a database of 14 GB for nodes and edges, and another 14 GB for efficient indexing.

We proceed to retrieve all CVE identifiers for vulnerabilities allocated for the Linux kernel in 2012 by the MITRE organization. In total, this amounts to 69 CVE identifiers, which address 88 unique vulnerabilities. We manually inspect the patches for each of

the 88 vulnerabilities to find their root cause and make use of this information to place each vulnerability into one of twelve types, which we describe in detail in Appendix B.

For each of the twelve groups, we determine (a) whether they can be modeled using the code property graph, and (b) which parts of the code property graph are needed to do so [8] In particular, we evaluate the ability to model the twelve types of vulnerabilities using the abstract syntax tree (AST) alone, the combination of syntax tree and dependence information (AST+PDG), the combination of syntax tree and control flow information (AST+CFG), and finally, the complete code property graph (AST+CFG+PDG). Table 2.1 shows the results of our analysis.

Vulnerability types	**Code representations**			
	AST	AST+PDG	AST+CFG	AST+CFG+PDG
Memory Disclosure				✓
Buffer Overflow		(✓)		✓
Resource Leaks			✓	✓
Design Errors				
Null Pointer Dereference				✓
Missing Permission Checks		✓		✓
Race Conditions				
Integer Overflows				✓
Division by Zero		✓		✓
Use After Free			(✓)	(✓)
Integer Type Issues				✓
Insecure Arguments	✓	✓	✓	✓

TABLE 2.1: Coverage of different code representation for modeling vulnerability types [162].

The abstract syntax tree alone offers little information for modeling of vulnerabilities, allowing only some forms of insecure arguments to be identified such as incorrect type casts. By additionally making available dependence information, we can model data flow, and can thus track attacker-controlled values. This makes some instances of buffer overflows, missing permission checks, and division by zero errors accessible. However, the statement execution order is not captured by this representation, making it insufficient to model vulnerabilities where exact locations of checks matter. Combining syntax trees with control flow graphs allows statement execution order to be modeled but fails to capture the majority of the twelve vulnerability types as missing data flow information prevents attacker control to be modeled. Still, some use-after-free vulnerabilities and resource leaks can be identified using this representation.

Using the code property graph, that is, by combining syntax, control flow and dependencies among statements, we are finally able to model ten of the twelve types of vulnerabilities. However, it also needs to be pointed out that it is hard to model race conditions and design errors using traversals. In the first case, graph traversals in their current form lack means to model concurrency. In the second, details of the desired design are necessary. In addition, the use-after-free vulnerabilities in the data set are often rather contrived cases that are difficult to describe without runtime information. A more thorough discussion of the limitations of modeling vulnerabilities using graph traversals is given in Section 7.2.

[8] The interested reader finds traversals for common types of vulnerabilities in our paper on modeling and discovering vulnerabilities with code property graphs [162].

2.5.2 Discovering Unknown Vulnerabilities

We proceed to craft traversals to uncover previously unknown vulnerabilities in the Linux kernel. To begin with, we create traversals for instances of the most common types of vulnerabilities encountered in the kernel in 2012, namely, *buffer overflows* due to missing bounds checks, and *memory disclosures* caused by incompletely initialized structures. In addition, we create traversals for *memory mapping vulnerabilities*, and *zero-byte allocations*, two types of vulnerabilities specific to the kernel, and thus hard to identify using conventional methods.

Type	Location	Identifier
Buffer Overflow	`arch/um/kernel/exitcode.c`	CVE-2013-4512
Buffer Overflow	`drivers/staging/ozwpan/ozcdev.c`	CVE-2013-4513
Buffer Overflow	`drivers/s390/net/qeth_core_main.c`	CVE-2013-6381
Buffer Overflow	`drivers/staging/wlags49_h2/wl_priv.c`	CVE-2013-4514
Buffer Overflow	`drivers/scsi/megaraid/megaraid_mm.c`	-
Buffer Overflow	`drivers/infiniband/hw/ipath/ipath_diag.c`	-
Buffer Overflow	`drivers/infiniband/hw/qib/qib_diag.c`	-
Memory Disclosure	`drivers/staging/bcm/Bcmchar.c`	CVE-2013-4515
Memory Disclosure	`drivers/staging/sb105x/sb_pci_mp.c`	CVE-2013-4516
Memory Mapping	`drivers/video/au1200fb.c`	CVE-2013-4511
Memory Mapping	`drivers/video/au1100fb.c`	CVE-2013-4511
Memory Mapping	`drivers/uio/uio.c`	CVE-2013-4511
Memory Mapping	`drivers/staging/.../drv_interface.c`	-
Memory Mapping	`drivers/gpu/drm/i810/i810_dma.c`	-
Zero-byte Allocation	`fs/xfs/xfs_ioctl.c`	CVE-2013-6382
Zero-byte Allocation	`fs/xfs/xfs_ioctl32.c`	CVE-2013-6382
Zero-byte Allocation	`drivers/net/wireless/libertas/debugfs.c`	CVE-2013-6378
Zero-byte Allocation	`drivers/scsi/aacraid/commctrl.c`	CVE-2013-6380

TABLE 2.2: Zero-day vulnerabilities discovered in the Linux kernel using our four graph traversals [162]

Running these four traversals, we identify 18 previously unknown vulnerabilities, all of which were acknowledged by the developers. Table 2.2 summarizes these findings along with the CVE identifiers assigned to these findings by the MITRE organization. From the perspective of the practitioner, it is also noteworthy that none of the four traversals took longer than 40 seconds to complete on a cold database. Moreover, once nodes and edges are loaded into memory, only 30 seconds were required at maximum, making it possible to iterative refine queries during the code auditing process.

2.6 Related Work

The platform for robust code analysis combines techniques and ideas from several fields of research, which we discuss in detail in the following. While our platform is specifically designed as the basis for vulnerability discovery via machine learning, we postpone discussion of related work on machine learning techniques for vulnerability discovery for now and focus on robust parsing and mining for vulnerabilities.

Robust Parsing of Source Code. The difficulty of creating suitable parsers for code scanning and reverse engineering methods has been recognized by several researchers. For example, refinement parsing as introduced in this chapter directly extends the work by Moonen [98] on island grammars, a formalism specifically designed in light of this

challenge. Similarly, Koppler [75] present a systematic approach to the design of fuzzy parsers, however, both of these approaches consider a setting where only limited parts of a language need to be recognized, while all other constructs can be ignored. This differs from the setting refinement parsing is designed for, where we strive to design a full-fledged but error-resilient grammar capable of dealing with incomplete code.

The difficulties in dealing with incomplete C++ code in particular, due to grammar ambiguities have been pointed out by Knapen et al. [73]. They additionally propose a set of heuristics and type inference to deal with incomplete C++ code gracefully. Closely related, Aycock and Horspool [10] argue for an approach in which the token remains ambiguous throughout the parsing process, a concept they refer to as *Schroedinger's Token*. Finally, Synytskyy et al. [141] point out that source files often mix several different languages, and present a multilingual robust parser based on Moonen [98]'s concept of island grammars.

Graph representations of code. The idea of performing program analysis by solving graph reachability problems was pioneered by Reps [114], who shows that many data-flow problems can be solved by traversing interprocedural graph-representations of data flow. Moreover, the creation of new graph-based program representations specifically with the discovery of defects in mind has previously been considered by Kinloch and Munro [71]. They present the Combined C Graph (CCG), which, in essence, is a system-wide program dependence graph. A similar representation is presented by Krinke and Snelting [79], who present a fine-grained system dependence graph that combines syntactical as well as control and data flow information similar to the code property graph. However, these approaches do not deal with efficient storage and retrieval from these data structures, nor are the merits of such representations for the discovery of defects and vulnerabilities explored.

Mining Source Code for Vulnerabilities. Mining large amounts of source code has been an ongoing field of research for many years. In particular, several approaches for lightweight scanning of source code have been proposed, including the well known scanners Flawfinder [159, 160], RATS [112], ITS4 [151], PScan [33], and Microsoft PREfast [82]. These tools are the result of efforts to discourage the use of certain problematic APIs and thus are particularly suited for educational purposes. Unfortunately, their success ultimately depends on their database of known offending APIs, and thus, they are incapable of identifying vulnerabilities related to program-specific APIs. This focus on common and well-understood error patterns makes these tools useful during the development process but means that code bases that have received scrutiny by security researchers in the past are typically out of scope, simply because these well known types of vulnerabilities have already been discovered. Finally, the parsers used by these approaches are kept simple, making it impossible for these approaches to take into account statement execution order. In effect, false positive rates are often prohibitively high.

To allow application-specific vulnerabilities to be identified, several researchers have considered to leverage expert knowledge to enhance automated approaches. For example, Hackett et al. [50] as well as Evans and Larochelle [40] leverage user-provided annotations to discover buffer overflow vulnerabilities. Similarly, Vanegue and Lahiri [149] introduce HAVOC-lite, a lightweight extensible checker shown to be capable of identifying vulnerabilities in Microsoft COM components. In addition, several approaches

based on security type systems have been proposed to identify information-flow vulnerabilities [see 59, 126, 134]. In particular, the *Jif* compiler [100, 101] allows annotated Java code to be checked for conformance to security policies. In addition, Jif performs type inference in order to reduce the required amount of user-specified annotations.

Several researchers have considered describing defects using query languages [e.g., 46, 51, 81, 94, 107]. In particular, Livshits and Lam [88] introduce the high level language PQL to describe typical instances of SQL injection and cross site scripting vulnerabilities in Java code. Moreover, Reps [114] shows that program properties can be determined by solving reachability problems in graphs, and additionally, Kinloch and Munro [71] as well as Krinke and Snelting [79] have proposed joint graph representations of code. However, to the best of our knowledge, the use of graph traversal languages to expose intermediate graph representations of code for scalable and extensible vulnerability discovery has not been considered to date.

3

Feature Spaces for Vulnerability Discovery

The previous chapter demonstrates that many types of vulnerabilities can be expressed as traversals in a code property graph, a joint representation of a program's syntax, control flow and data flow. However, these traversals are manually specified by an expert, and we have yet to explore possibilities to automatically infer them from code. In this chapter, we take a fundamental step into this direction by *embedding source code in feature spaces*, that is, we represent code by numerical vectors in order to make it accessible for machine learning algorithms. To this end, we define several different maps from source code to vectors, for example, to represent source files, functions, and even program slices [see 63, 157] by a wide range of different properties. We present these maps in a unified framework based on the well known *bag-of-words* embedding that finds applications in many fields, including information retrieval [see 92, 128], natural language processing [see 91], and computer vision [see 26]. On an implementation level, we find that the code property graph provides a versatile data source when implementing these embeddings, making it possible to recognize patterns in code on a large scale. In fact, we can formulate a general procedure for embedding of source code based on code property graphs, which we employ throughout this thesis to implement method's for pattern-based vulnerability discovery.

We begin this chapter by providing a brief overview of how objects can be made accessible to machine learning using *feature maps* (Section 3.1). In preparation for embedding of source code, we proceed to provide a slightly generalized formal description of the classical *bag-of-words* embedding that exposes the right set of parameters to allow all embeddings employed in this thesis to be formulated as instances of it (Section 3.2). We continue to describe feature hashing for the bag-of-words embedding to allow our methods to operate on large code bases such as the operating system kernel Linux (Section 3.3). Finally, we describe different embeddings for source code in particular (Section 3.4) and provide our method to generate these embeddings from code property graphs, the main data structure offered by our code analysis platform (Section 3.5). We conclude by discussing related work (Section 3.6).

3.1 Feature Maps

Machine learning algorithms are a powerful tool used in a wide range of research areas, including natural language processing, bioinformatics and chemistry. Their ability to recognize patterns in many different types of data rests on their abstract formulation: instead of accounting for the semantics of the data they process, they solve optimization problems involving numerical vectors, and thus, if the objects we seek to process can be represented in this way, machine learning becomes accessible immediately.

While the large number of problems that can be addressed using machine learning is intriguing, applying it is not foolproof. For example, a well known anecdote tells the story of United States Army researchers who built a classifier to distinguish camouflaged tanks from plain forests based on sample imagery [167]. While the predictor based on pixel values functioned well for the recorded images, it failed in practice. They found that the predictor had learned to distinguish cloudy from sunny days as the photos of tanks had been recorded on the former, while those of plain forest on the later.

This example illustrates that while machine learning may produce results for a data set, these results may not be meaningful for the task at hand. Their success in applications thus ultimately depends on the choice of meaningful *features* by which to characterize objects of interest. We define features as follows.

Definition 3.1. In machine learning, a feature is a measurable property of an object that is used to characterize the object. For each object, its value is uniquely defined.

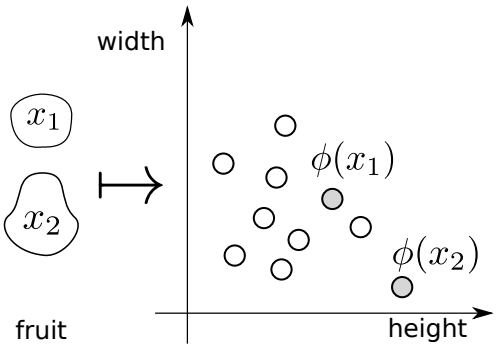


FIGURE 3.1: Example of a feature map

As an example, we consider the task of characterizing a basket of fruit. Possible features are the height of the fruit, and its width. However, one feature alone may not be sufficient to determine a fruit type, making it beneficial to record several features of the object in a joint *feature vector*. Figure 3.1 illustrates this feature extraction process, where we represent each fruit in a two dimensional space with one dimension to represent width, and another representing height. This measuring process can be captured by the notion of feature maps, which we define as follows.

Definition 3.2. A feature map is a function $\phi : \mathcal{X} \to \mathbb{R}^n$ that assigns a real-valued vector to each object of an input space $\mathcal{X}$. For any $x \in X$, $\phi(x) = (\phi_1(x), \ldots, \phi_n(x))$ is referred to as a *feature vector*, and ϕ's co-domain is referred to as a *feature space*.

Feature maps can be defined for many different types of objects, including sequences, hierarchies, and even graphs, making it possible to represent them as vectors in a feature space. Once in this space, distances between vectors can be measured geometrically as an indicator for object similarity.

3.2 Bag of Words

The bag-of-words embedding is a technique popularized in particular by the *vector space model* developed in information retrieval [128], where text documents are represented as feature vectors. In the original setting, we consider the set of text documents as an input space $\mathcal{X}$, and assume that each of these documents $x \in \mathcal{X}$ can be represented as a sequence of words of a so-called embedding language L [see 120]. To map a document to a corresponding vector, we then simply associate each word $w \in L$ with a dimension of the feature space and store *the number of times the word occurs in the document* as a coordinate. Formally, we achieve this by defining a feature map

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

from documents to a vector space spanned by the words of the language L. The values of coordinates are calculated independently for each word, that is we define a function ϕ_w for each $w \in L$. To count the number of occurrences, we set

$$\phi_w(x) := \#_w(x) \cdot v_w$$

where $\#_w(x)$ denotes the number of occurrences of the word w in x, and v_w is an optional weighting term to increase the importance of certain words over others. The vectorial representation of x is then simply given by

$$\phi(x) = (\phi_w(x))_{w \in L} \,.$$

There are a number of variations of the definition of $\phi_w(x)$. In particular, $\phi_w(x)$ can also be chosen to express whether the word w occurs in the document at all. To achieve this, we can define an indicator function $I_w : \mathcal{X} \to \{0, 1\}$ where

$$I_w(x) = \begin{cases} 1 & \text{if } x \text{ contains } w \\ 0 & \text{otherwise} \end{cases}$$

and choose $\phi_w(x)$ to be given by $\phi_w(x) := I_w(x) \cdot v_w$, that is, if the word w is contained in the document x, the weight v_w is stored in the corresponding coordinate, and otherwise, the coordinate is zero. Whether we choose ϕ_w to account the number of occurrences or indicate the existence of a word in the document is not relevant in the remainder of the chapter, and thus, we stick only to the first definition of ϕ_w to simplify discussion.

By slightly generalizing this idea, we obtain a template for an embedding that can be instantiated in different ways to obtain the embeddings for source code presented in this thesis. To this end, we assume that $\mathcal{X}$ is an arbitrary input space, and that each $x \in \mathcal{X}$ can be decomposed into a set of substructures. For each object x, we denote this set by T_x, and the set of all substructures by $\mathcal{T}$. Like addresses in memory or nodes of a graph, substructures can be thought of as unique locations, which exists independent of the contents they store. We attach contents, that is, words of the language L, to these

substructures using a function $s : \mathcal{T} \to L$, where several substructures may be mapped to the same word. The number of times the word $w \in L$ occurs in x is then given by $\#_w(x) = |\{t \in T_x, s(t) = w\}|$, that is, we determine the number of substructures that are mapped to the word w. The coordinate associated with the word w is thus given by

$$\phi_w(x) = \#_w(x) \cdot v_w = |\{t \in T_x, s(t) = w\}| \cdot v_w$$

where, again, the number of occurrences is multiplied by a weighting term v_w. In summary, our slightly generalized bag-of-word embedding is carried out in the following three steps.

1. **Object extraction.** We begin by choosing an input space $\mathcal{X}$ and thus decide which objects we wish to compare.
2. **Sub structure enumeration.** Objects are subsequently decomposed into the substructures by which we want to represent them. We denote the set of all substructures by $\mathcal{T}$.
3. **Mapping objects to vectors.** Finally, we map sub structures to words, and subsequently represent each object by a vector that encodes the number of occurrences of each word. This steps rests on the definition of an embedding language L and a function $s : \mathcal{T} \to L$ that maps substructures to words.

It may seem troubling at first that the bag-of-words feature map associates each word with a unique dimension, thereby creating possibly very high dimensional vectors. In fact, the language L is not necessarily finite, and thus, the resulting feature space may even be infinite dimensional. However, particularly in the setting of unsupervised learning, we can limit L to those words that actually occur in the data set, and thus obtain a vector space with a finite number of dimensions. Moreover, the resulting vectors are typically sparse as each document contains only a small subset of the words of the language. This sparsity can be exploited to efficiently store and process these vectors using data structures such as hash maps and sorted arrays [120].

3.3 Feature Hashing

A practical problem when implementing the bag of words embedding illustrated in the previous section is that libraries for machine learning and linear algebra typically require each dimension of the feature space to correspond to a numerical index, however, associating words with indices is not possible until all words of the data set are enumerated. Unfortunately, this means that objects cannot be mapped to vectors independently, and that all words contained in the data set need to be kept in memory at one point in time. In effect, applying the mapping to large data sets is prohibitive in practice.

A well known trick to deal with this problem is *feature hashing* [see 135]. Instead of striving to ensure that no two words are ever associated with the same dimension, we simply attempt to reduce the probability of this event. We achieve this by using a hash function to map words to dimensions, as hash functions aim to distribute input values uniformly over their co-domain. As the hash function is only calculated over the word, we can thus map objects to vectors independently without the need for exhaustive enumeration of words in the data set.

An interesting aspect of this approach is that the cardinality of the hash function's co-domain already defines the dimensionality of the resulting vector space. On the one hand, this is desirable as it gives control over the amount of required memory for operations on the full data matrix, at least if an upper bound on the size of the data set can be given. On the other, it is problematic as the chosen cardinality may be too low to avoid frequent hash collisions, and thus many false dependencies among data points are introduced.

Formally, feature hashing for the bag of words embedding can be implemented by defining a hash function $h : L \to \{1, \ldots, N\}$ from words to the natural numbers between and including 1 and N. The desired feature map $\bar{\phi} : \mathcal{X} \to \mathbb{R}^N$ maps objects onto N dimensional vectors, where N is the number of different hash values. As is the case for the classic bag of words embedding, each of the coordinates of the vector can be calculated independently, that is, $\bar{\phi}(x)$ can be expressed as

$$\bar{\phi}(x) = (\bar{\phi}_j(x))_{j \in \{1,\ldots N\}}$$

where for each dimension $j \in \{1, \ldots, N\}$, the function $\bar{\phi}_j : \mathcal{X} \to \mathbb{R}$ maps objects to real-valued coordinates. In each coordinate, we store the number of substructures that are eventually mapped to the hash value j, multiplied by a weighting term v_w. In correspondence with the formulation of feature hashing given by Shi et al. [135], we thus define the j'th coordinate by

$$\bar{\phi}_j(x) = \sum_{w \in L, h(w)=j} \#_w(x) \cdot v_w = \sum_{w \in L, h(w)=j} |\{t \in T_x, s(t) = w\}| \cdot v_w$$

where, in the final step, we make use of the definition of $\#_w$ given in the previous section. The implementation of feature hashing therefore merely requires a (non-cryptographic) hash function to be specified. In practice, we use hash functions from the MurmurHash [7] family, an efficient family of hash functions with implementations available as public domain software.

3.4 Feature Maps for Source Code

We now present four different types of embeddings for source code based on bag-of-words. These are presented in an order that corresponds to the amount of pre-processing required to be performed on the code. We begin with an embedding that can be generated based on lexical analysis of code alone, i.e., without requiring parsing of source code, and point out its inherent weaknesses. We continue to present embeddings based on specific symbols of interest that can be constructed using fuzzy parsers with a limited knowledge of the program syntax, and continue to discuss embeddings based on complete syntax trees. These embeddings play a vital role for our methods for vulnerability extrapolation and missing check detection presented in Chapters 4 and 5. Finally, we present an embedding based on hashing of graphs to represent arbitrary patterns in code property graphs, as well as multi-stage feature maps, which in combination enable learning of traversals for vulnerability discovery as presented in Chapter 6.

3.4.1 Token-based Feature Maps

Similar to the decomposition of text documents into streams of words in natural language processing, source files can be lexically analyzed to decompose them into streams of tokens. Continuing along this analogy, text documents logically group the sentences and words they contain, while source files group logically related code. Applying the bag-of-words idea to source code, we can therefore *represent source files by the tokens they contain*, as has been explored in the context of vulnerability discovery in a supervised setting by Scandariato et al. [130] to determine possibly vulnerable files, and by Perl et al. [108] to determine commits that introduce vulnerabilities.

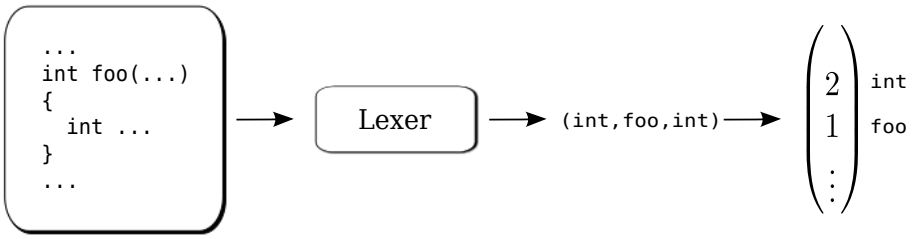


FIGURE 3.2: Token-based feature maps

Figure 3.2 illustrates this idea. The source file is simply tokenized by a lexer to yield a sequence of tokens, where each of these tokens is associated with a word of an embedding language. The bag-of-words embedding is subsequently carried out to represent source files in terms of the number of occurrences of its tokens. Following this idea, we can define token-based feature maps as an instance of a bag of words feature map (Section 3.2).

Definition 3.3. A *token-based feature map* is a bag-of-words feature map where the input space $\mathcal{X}$ corresponds to a set of source files, and, for each source file $x \in \mathcal{X}$, its set of substructures T_x corresponds to the tokens it contains according to a lexer.

Like the nodes of a graph, tokens have a unique identity, meaning that no two tokens are the same across the entire data set. However, tokens carry content in the form of words, and several tokens may store the same word. We assign content to tokens via the function $s : \mathcal{T} \to L$. In the simplest case, s simply maps a token to the string it stores, however, s can also introduce normalizations, e.g., all numbers or all arithmetic operators can be mapped to the same word. In correspondence with the generic definition of bag of words given in Section 3.2, the feature map ϕ is then given by

$$\phi : \mathcal{X} \to \mathbb{R}^{|L|}, \text{ with } \phi(x) = (\phi_w(x))_{w \in L}$$

where $\phi_w(x) = |\{t \in T_x, s(t) = w\}| \cdot v_w$, that is, we store the number of times a token contains the word w in the source file x in each coordinate, corrected by a weighting factor v_w that expresses the words importance over others.

This embedding is simple to implement and can be carried out even if a full-fledged parser is unavailable as it only relies on the definition of delimiters that make it possible to split source code into tokens. However, this simplicity does not come without drawbacks that make it of little use for vulnerability discovery in practice.

First, a predictor that suggests to the analyst that a vulnerability exists within thousands of lines of code may be right, however, it also leaves the analyst with thousands of lines

to review, making this information rather useless in practice. Second, tokenization does not expose higher level programming constructs such as statements, assignments, or calls. In effect, the learner can compare occurrences of textual tokens but attaches little meaning to these. For example, it cannot differentiate between the volatile names of local variables, and calls to API functions, and it does not see whether a variable is used on the left-hand or right-hand side of an assignment, or whether a statement is executed before or after another. In the following, we discuss symbol-based feature maps, which address some but not all of these concerns.

3.4.2 Symbol-based Feature Maps

Feature maps based on tokens leave two primary problems to address. On the one hand, we want to increase the granularity of the analysis to narrow in more effectively on vulnerable code. On the other, the quality of features used for object characterization requires improvement. In this section, we introduce symbol-based feature maps, an extension of token-based feature maps, to address these problems. These rely on limited parsing of source code as can be achieved using a fuzzy parser that extracts only certain program constructs of interest, e.g., function definitions, calls, or variable declarations. To this end, we can employ a two-level refinement parser (see Section 2.2.1) to obtain a corresponding abstract syntax tree.

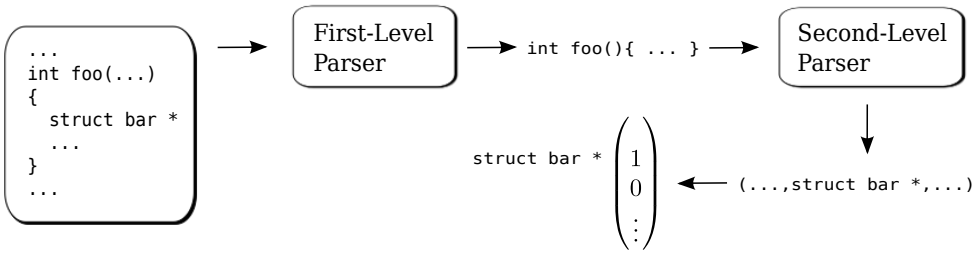


FIGURE 3.3: Symbol-based feature maps

Figure 3.3 illustrates how a symbol-based feature map is carried out. A first-level parser extracts the objects to compare. While in the token-based approach, these objects are source files where boundaries are known without parsing, any attempt to choose objects located inside these files relies on parsing to identify where the object begins and where it ends. For example, we may want to select all namespaces, or all functions as objects.

The second-level parser extracts features from objects to characterize them. For example, we may extract all calls or types used in parameter and local variable declarations. In contrast to tokens, these features are higher level programming constructs, possibly composed of several tokens. For example, the type name `struct myClass *` consists of the three tokens `struct`, `myClass`, and the star-operator. Programming constructs correspond to *symbols* of the grammar, that is, terminals as well as non-terminals (see Section 2.2.1), and hence we refer to these embeddings as symbol-based. Formally, we define symbol-based feature maps as follows.

Definition 3.4. A *symbol-based feature map* is a bag-of-words feature map where the input space $\mathcal{X}$ is given by a set of compounds obtained by a first-level parser, and the set of substructures T_x of $x \in \mathcal{X}$ is given by the set of symbols contained in x according to a second-level parser.

It is noteworthy that symbol-based embeddings generalize token-based embeddings. For example, by choosing objects to be source files and providing a second-level parser that simply identifies all tokens, we obtain a symbol-based embedding equivalent to the token-based embedding presented in the previous section.

Example. As an example of this type of embedding, *we represent functions by the API symbols they contain*, that is, its callees and types used in declarations of parameters and local variables. Figure 3.3 illustrates this embedding. The first-level parser begins by identifying function definitions inside source files and passes it on to the second-level parser. This parser, then identifies API symbols inside the function, and finally, the bag-of-words embedding is performed.

Formally, this feature map can again be specified as an instance of the bag of words embedding (Section 3.2), that is, it is of the form

$$\phi : \mathcal{X} \to \mathbb{R}^{|L|}, \text{ with } \phi(x) = (\phi_w(x))_{w \in L}$$

where again, $\phi_w(x)$ expresses the number of occurrences of the word w in x. *The input space $\mathcal{X}$ is now given by the set of functions. The set of substructures $\mathcal{T}$ is the set of API nodes in the syntax tree*, that is, all nodes of the syntax tree that represent types, and those that represent callees. The words of the language L are the textual representations of API symbols, and again, the function $s : \mathcal{T} \to L$ assigns words to each token.

Symbol-based embeddings are not limited to embedding of functions but can be created to characterize namespaces, blocks within loops, or even statements. They cannot however, be used to characterize program constructs that can only be extracted when data or control flow information is available, as for example, program slices.

In principle, symbols corresponds to nodes in abstract syntax trees, and thus, we represent functions by subtrees of its syntax tree. However, symbol-based embeddings do not make use of the decomposition offered by syntax trees, but rather treat them as flat objects by mapping trees to their string representations using the function s. In the following section, we show how embeddings based on syntax trees can be created that exploit this decomposition.

3.4.3 Tree-based Feature Maps

Symbol-based feature maps are already an improvement over token-based feature maps as they enable us to represent code by high level programming constructs. In principle, these symbols correspond to subtrees in the abstract syntax tree, however, the symbol-based embedding ignores this tree structure and represents the tree by a flat string. For symbols such as callees and types, that are likely to reoccur in exactly the same form in other parts of the code base, this is not problematic. However, this flattening is not suited for larger trees such as those corresponding to statements or loop bodies. While these may occur in other locations of the code in a similar form, finding the exact same string of code is unlikely.

This limitation can be addressed using *tree-based* feature maps. Again, we assume that we have access to a two-level refinement parser, where the first- and second-level parser extract objects of interest (e.g., functions) and program constructs for object characterization respectively. However, the second-level parser is now assumed to extract

more complex program constructs, e.g., entire statements or compound statements, as opposed to trees representing types, callees, or identifiers. In the extreme case, the second-level parser simply returns an entire abstract syntax tree for each function. The idea we now follow is to represent objects by the subtrees contained in its program constructs.

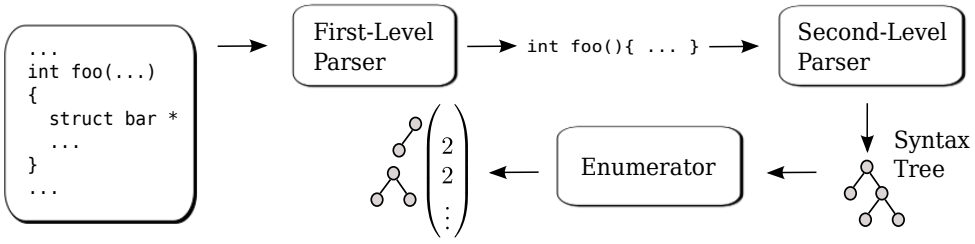


FIGURE 3.4: Tree-based feature maps

Example. As an example, we represent the functions of a code base by subtrees of their entire syntax tree (Figure 3.4). Again, a first-level parser is used to extract function boundaries, just like in the symbol-based embedding presented in the previous Section. However, the second level parser now creates a complete syntax tree. We then enumerate subtrees of the syntax tree, which are mapped to strings. Finally, objects are mapped to vectors following the remainder of the bag-of-words procedure.

Formally, this embedding is again an instance of the bag-of-words embedding. With a two-level refinement parser and an enumerator at hand, we can define tree-based feature maps as follows.

Definition 3.5. A *tree-based feature map* is a bag-of-words feature map where the input space $\mathcal{X}$ is a set of objects obtained by a first-level parser, each represented by a corresponding syntax tree obtained by a second-level parser, and for each $x \in X$, the set of substructures T_x of x is given by the set of subtrees of its syntax tree according to an enumerator.

In principle, this fully describes tree-based mappings, however, two practical problems remain: (a) how exactly do we choose subtrees of the syntax tree, and (b) how do we compare subtrees to determine equality?

Enumerating subtrees. First, syntax trees representing functions in real world code can easily span several thousands of nodes, and therefore, using all possible subtrees to characterize objects is computationally prohibitive in practice. We account for this by specifying an *enumerator*, that extracts only certain types of subtrees from the syntax tree. Let $\mathcal{A}$ denote the set of syntax trees for objects in $\mathcal{X}$. Then, an enumerator is a function

$$E : \mathcal{A} \to \mathcal{P}(\mathcal{T})$$

that maps syntax trees to sets of subtrees[1]. As an example, E may enumerate all subtrees of a given depth, or all subtrees up to that depth. Alternatively, it may enumerate only subtrees rooted in certain types of nodes as proposed by Rieck et al. [118] in the context of parse tree embedding. With an enumerator in place, the set T_x of subtrees contained in x is simply obtained by evaluation E on the syntax tree of x.

[1] $\mathcal{P}(\mathcal{T})$ denotes the power set of $\mathcal{T}$

Mapping subtrees to words. Once subtrees are enumerated, we need to decide when two of these trees should be considered the same, as these should be mapped to the same dimension of the feature space. While for graphs in general, this is a difficult problem, requiring us to determine whether the graphs are isomorphic, for ordered, attributed trees such as the abstract syntax tree, the problem is much easier to solve. As ordered, labeled trees can be serialized into a unique textual representation, we can simply compare the textual representations of trees to identify whether they are the same. To this end, we represent subtrees by *S-Expressions* [see 122], a common string representation of trees popularized by the Lisp programming language. Formally, we choose the language L to be the set of S-Expressions, while the function s that maps substructures to words is now defined to be a function that maps trees to their corresponding S-Expressions.

The capabilities of tree-based feature maps, and symbol-based feature maps in particular, are explored in Chapter 4 in detail, where we evaluate the applicability of different variations of these feature maps for the detection of similar functions.

3.4.4 Graph-based Feature Maps

The embedding presented on the previous section is well suited to compare graphs where the ordering of edges is relevant. However, there are features where robustness to variations in neighbor order is desirable. For example, the additive expression `a + b` has the same semantics as the expression `b + a` as addition is commutative. Unfortunately, the mapping from substructures to words presented in the previous section produces the S-Expressions $(+a\,b)$ and $(+b\,a)$, which are not the same, and thus, the two expressions are mapped to different dimensions.

Graph-based embeddings offer a more robust type of embedding for trees and graphs in general. The main idea is to calculate hash values for substructures and use these as dimensions of the feature space, that is, the embedding language is now a set of hash values. In particular, we present a method of this type that calculates so-called neighborhood hashes [61], a hash for nodes of a graph that takes into account the node itself as well as its neighbors.

Example. As an example, we consider the task of representing a function by the subgraphs of its control flow graph, a process illustrated in Figure 3.5. As in the previous two examples, the input space $\mathcal{X}$ is the set of functions, and again, we proceed to present each of these functions by its corresponding control flow graph via a second-level parser. Finally, we extract subgraphs of the control flow graph via an enumerator, and employ a hash function to map these subgraphs to words of an embedding language.

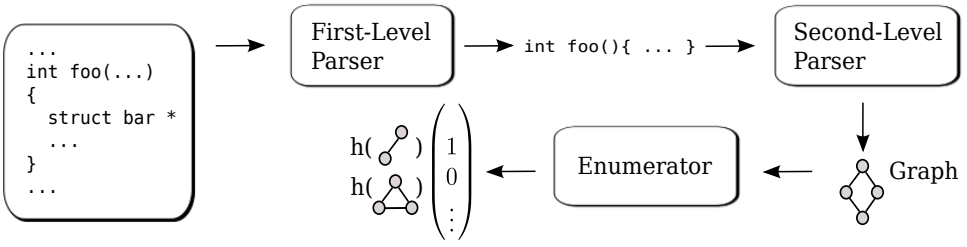


FIGURE 3.5: Graph-based feature maps

Graph-based feature maps are a generalization of tree-based feature maps and an instance of the generic bag of words feature map, which we define as follows.

Definition 3.6. A *graph-based feature map* is a bag-of-words feature map where the input space $\mathcal{X}$ is a set of objects obtained by a first-level parser, each represented by a graph obtained by a second-level parser, and for each $x \in X$, the set of substructures T_x of x is given by the set of subgraphs according to an enumerator. Finally, the mapping s from substructures to words is given by a hash function h.

Neighborhood hashing. With substructures at hand, the core question is how they can be mapped to hash values, the dimensions of our feature space. To this end, we perform *neighborhood hashing*, a procedure devised by Hido and Kashima [61]. The procedure begins by assigning an initial hash value to each node of the substructure using a function l. For inner nodes of the tree, we calculate this value from its node type while for leaf nodes, we calculate it from its value. With initial hash values for each node of the substructure, we now want to incorporate the hash values of child nodes into those of its parents to jointly represent both the parent node and its children by a single hash value. To this end, we employ the hash function $h : V \to L$ from nodes to hash values given by

$$h(v) = r(l(t)) \oplus \left(\bigoplus_{z \in C_v} l(z) \right)$$

where r denotes a bitwise rotation, and C_v are the child nodes of v [see 44, 61, 165]. We proceed to update the labels of the substructure by calculating $h(v)$ for each node and using this hash value as its new hash value. This process is carried out several times to incorporate information from child nodes, that are only indirectly connected to v. The function $s : \mathcal{T} \to L$ from substructures to hash values then simply assigns the hash value of t's root node to t.

Neighborhood hashing is a technique that can be employed for arbitrary graphs. In this thesis, we make extensive use of this technique in Chapter 6 to represent expressions that sanitize attacker-controlled data.

3.4.5 Multi-Stage Feature Maps

Graph-based feature maps already allow substructures to be mapped to the same dimension even if the order of child nodes differs. However, the embedding cannot account for any other similarities between substructures, and map them onto the same dimension. For example, if we represent a function by the API symbols it contains, the API symbol `malloc` and `calloc` are mapped to two orthogonal directions despite their textual similarity. This leads us to the idea of multi-staged feature maps. Instead of directly representing objects by their substructures, we instead embed substructures first and cluster them to obtain groups of similar substructures (see Section 1.2.2 and 6.2). We then represent objects by the groups their substructures are contained in.

Example. Let us reconsider representing functions by their API symbols as discussed already in Section 3.4.2. However, instead of performing a symbol-based embedding, we first cluster API symbols. In effect, we obtain a set of n clusters. As we want to represent functions by the clusters their API symbols are contained in, the embedding

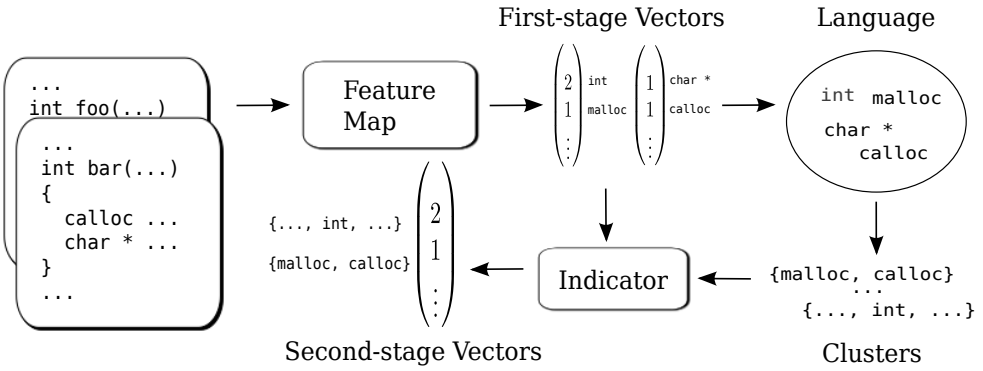


FIGURE 3.6: Multi-stage feature maps

language contains a dimension for each cluster (Figure 3.6). In detail, this embedding works as follows.

Clustering substructures. We again extract functions using the first-level parser and employ a second-level parser to uncover API-symbol nodes. API-symbol nodes are then mapped to API symbols using a function s as described in Section 3.4.2, and thus we obtain a set $\mathcal{L}$ of API symbols contained in the code base. We now form groups of similar words in $\mathcal{L}$, either by calculating string distances for all pairs, or by embedding the words of $\mathcal{L}$ in a vector space themselves, and employing a clustering algorithm on these data points. As a result, we obtain a set of clusters C, where each of these clusters is a set of words.

Representation by clusters. We continue to represent functions by the clusters their API symbols are contained in. The embedding language L is thus given by the natural numbers from 1 to $|C|$, that is, there is a dimension for each cluster, and the feature map is given by

$$\phi : \mathcal{X} \to \mathbb{R}^{|C|}, \phi(x) = (\phi_c(x))_{c \in C}$$

that is, x is mapped to a $|C|$ dimensional vector space and each coordinate is calculated independently as $\phi_c(x) = \#_c(x)$ where $\#_c(x)$ denotes the number of occurrences of the cluster c in x. As the words of the feature space are now cluster indices as opposed to words, however, the definition of $\#_c$ differs from that of the counting functions we discussed so far. The number of times the cluster occurs in x should be equal to the number of substructures that, when mapped to words, correspond to a cluster member of c, that is

$$\#_c(x) = |\{t \in T_x, s(t) \in c\}|$$

where, in contrast to definitions of counting functions in the bag of words embeddings discussed so far, we do not demand that $s(t)$ is equal to c, but instead, $s(t)$ is in c. Alternatively, we can also replace the counting function $\#_c$ by an indicator function I_c defined as

$$I_c(x) = \begin{cases} 1 & \text{if } |\{t \in T_x, s(t) \in c\}| \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

to merely indicate whether at least one of the substructures is mapped to a word contained in the cluster c.

Formally, multi-stage feature maps are yet again an instance of the bag of words feature map, and they can be defined as follows.

Definition 3.7. A *multi-stage feature map* is a bag-of-words feature map where the input space $\mathcal{X}$ is given by a set of objects, and the substructures are clusters of words of an embedding language L associated with a first-stage feature map $\hat{\phi}$. For each $x \in \mathcal{X}$, the set of substructures T_x are the clusters containing the words associated with its first-stage substructures, where the first-stage substructures of x are the substructures of x according to $\hat{\phi}$.

Multi-stage feature maps play an important role in Chapter 6 where they are employed to represent program slices by sets of strings that describe the initialization of variables. In particular, associating dimensions of the feature space with sets of strings allows us to represent initializers by corresponding string patterns expressed as regular expressions.

3.5 Feature Maps on Code Property Graphs

The previous section presents bag-of-words embeddings for source code based on different objects and substructures. However, it is not yet clear how these objects and substructures can be extracted in the first place. In this section, we address this problem by providing a general procedure for the creation of bag-of-words embeddings from code property graphs.

Recalling our description of the bag-of-words embedding given in Section 3.2, we need to define several parameters to obtain a concrete bag-of-words embedding. First, we need to choose an input space $\mathcal{X}$ containing the objects we wish to compare. Second, we need to decide what kind of substructures we want to extract from objects to characterize them. This amounts to the definition of a set of substructures $\mathcal{T}$ along with an enumerator E that maps each object to the set of substructures it contains. Finally, we need to define an embedding language L and a function $s : \mathcal{T} \to \mathcal{L}$ that maps each substructure to a word.

By limiting ourselves to objects that correspond to subgraphs of the code property graph, and representations of these in terms of their subgraphs, we arrive at the following procedure to implement the three steps of a bag-of-words embedding (see Section 3.2).

- **Object extraction.** We begin extracting objects from the code property graph and thereby implicitly defining the input space $\mathcal{X}$. We achieve this by selecting a seed node in the code property graph for each object, and subsequently expanding it to obtain the object's graph representation (Section 3.5.1).
- **Substructure Enumeration.** For each graph, we proceed to label its nodes and edges, and enumerate it subgraphs, thereby specifying the set of substructures $\mathcal{T}$ and the enumerator E. This enables us to represent objects in terms of the subgraphs they contain (Section 3.5.2).
- **Mapping objects to vectors.** With objects and substructures at hand, we finally map each object to a vector that represents the substructures it contains. To this end, we employ hashing as a simple strategy to enable processing of objects independently. Implementing this step implicitly defines the language L, along with the map s from substructures to words (Section 3.5.3).

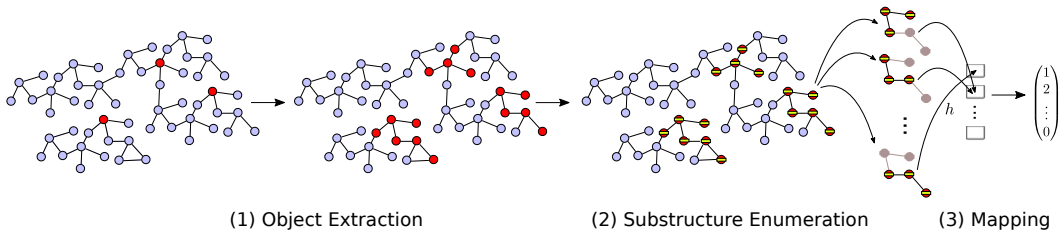


FIGURE 3.7: Embedding procedure based on code property graphs

Figure 3.7 illustrates this process. In the following, we discuss how each of these steps can be implemented in terms of operations on the code property graph, and provide the parameters of the embedding procedure, which concrete embeddings must implement.

3.5.1 Object Extraction

As discussed in Section 3.1, the first step for embedding source code in feature spaces is to select a suitable input space $\mathcal{X}$, that is, we need to choose objects of interest. Our embedding procedure assumes that these objects correspond to subgraphs of the code property graph. For example, if our task is to determine similar functions based on syntax, these objects may be syntax trees of functions. In another setting, we may want to compare calls to a specific API function. In this case, we can choose backward slices of invocations as objects.

Object extraction can be implemented as a two step procedure. First, we determine a set of *seed nodes* for each object, and second, we expand seed nodes recursively to uncover corresponding objects. For example, to extract syntax trees of functions, we select all function nodes as seed nodes, and recursively expand all syntax edges to uncover the respective syntax trees.

We can describe this process in terms of operations on the code property graph. Given a code property graph and its nodes V, we first execute a function S for seed-node selection that preserves seed nodes while discarding all other nodes. For each object, S returns a set of seed nodes. Each seed node is subsequently expanded according to the expansion $\mathcal{E}$ (see Section 2.3.2) and up to a maximum depth of $\mathcal{D}$. For example, to extract syntax trees of functions, we choose S to be a function that returns single-elemented sets of function nodes, and $\mathcal{E}$ to be given by $\textsc{OutE}_{\mathcal{A}}$, that is, the expansion that assigns outgoing syntax edges to each node. Finally, the maximum depth $\mathcal{D}$ is chosen to be infinite, enabling us to uncover complete syntax trees. For tree structures, the depth parameter is rather dispensable, however, for graphs in general, this parameter is important to ensure termination.

We denote the function that expands seed nodes into objects as $\textsc{Object}^{\mathcal{D}}_{\mathcal{E}}(X)$. This function can be defined as a restriction (see Appendix A) of the code property graph g and is given by

$$\textsc{Object}^{\mathcal{D}}_{\mathcal{E}}(X) = g|_{U^{\mathcal{D}}_{\mathcal{E}}(X), D^{\mathcal{D}}_{\mathcal{E}}(X)}$$

where $U^{\mathcal{D}}_{\mathcal{E}}$ and $D^{\mathcal{D}}_{\mathcal{E}}$ are traversals to uncover the node and edge set of the subgraph respectively. Recalling that $\hat{\mathcal{E}}(X)$ is the set of nodes reached by the expansion $\mathcal{E}$ (see

Section 2.3.2), the traversal for uncovering the node set is given by

$$U_{\mathcal{E}}^{d}(X) = \begin{cases} \emptyset & \text{if } d = 0 \\ X \cup U_{\mathcal{E}}^{d-1}(\hat{\mathcal{E}}(X)) & \text{otherwise} \end{cases}$$

that is, the node set of a set of input nodes X is given by these input nodes, along with the node set of all nodes reachable via nodes in X using the expansion. Analogously, the edge set is given by

$$D_{\mathcal{E}}^{d}(X) = \begin{cases} \emptyset & \text{if } d = 1 \\ \mathcal{E}(X) \cup D_{\mathcal{E}}^{d-1}(\hat{\mathcal{E}}(X)) & \text{otherwise} \end{cases}.$$

where the definitions of the two traversals differ only in the depth at which the recursion terminates, and in the fact that one returns nodes, while the other returns edges[2].

Putting these pieces together, for a seed node selector S, an expansion $\mathcal{E}$ and the nodes V of a property graph, the set of objects $\mathcal{O}$ is given by

$$\mathcal{O} := \{\textsc{Object}_{\mathcal{E}}^{\mathcal{D}}(X), X \in S(V)\}$$

meaning that we simply evaluate the function $\textsc{Object}_{\mathcal{E}}^{\mathcal{D}}$ on each set of seed nodes to obtain a set holding the resulting graphs (see Section 4.4.1).

3.5.2 Substructure Enumeration

After extracting objects in the form of graphs, we determine their subgraphs in two steps. First, we label their nodes, where the labeling determines which information our mapping from substructures to words of the embedding language takes into account. Second, we enumerate subgraphs such that for each object, a corresponding set of subgraphs is available.

Node labeling. While the nodes and edges of the code property graph are already attributed, these attributes often contain information irrelevant for the learning task, which introduces additional noise and hinders similarity assessment. Therefore, our first task is to leverage existing attributes to attach a single label to each node, which is the only node content the embedding will take into account. For example, we may seek an embedding where nodes are considered the same if they both represent a variable, regardless of the lines they appear on, or the name of the variable. We achieve this by labeling nodes by their types and discarding all other attributes.

The property graph's ability to carry attributes can be leveraged to label code property graphs. To this end, we simply replace the existing attribute function μ by a new attribute function μ_l that assigns values for the key *label* to all nodes of the code property graph. Formally, the labeled graph $\textsc{Labeled}_l(g)$ is given by $(V, E, \lambda, \mu_l, s, d)$ where μ_l is an attribute function using the single attribute key *label*, and defined as

$$\mu_l(x, k) = \begin{cases} l(x) & \text{if } k = \textit{label} \\ \epsilon & \text{otherwise} \end{cases}$$

[2]If expanding $\mathcal{D}$ times is not possible, this function gives back the full subgraph reachable from the seed node using the expansion function.

for all nodes x of the code property graph, l is a labeling function that maps nodes to property values, and ϵ is an empty word. For example, $l(x)$ can simply correspond to the *type* or *code* attribute of the node.

Subgraph enumeration. With a labeled graph for each object at hand, we can now enumerate the subgraphs it contains. As we have already made use of when extracting objects, for a graph g, the subgraph seeded in the node x obtained according to an expansion $\mathcal{E}'$ up to a depth d is obtained as the restriction

$$\textsc{Subgraph}^d_{\mathcal{E}'}(x) = g|_{U^d_{\mathcal{E}'}(\{x\}), D^d_{\mathcal{E}'}(\{x\})}$$

where again, $U^d_{\mathcal{E}'}$ and $D^d_{\mathcal{E}'}$ are traversals to uncover the node set and edge set respectively. In principle, the expansion $\mathcal{E}'$ does not need to be equal to the expansion $\mathcal{E}$ used in object extraction, however, for the embeddings implemented in our work, $\mathcal{E}$ is equal to $\mathcal{E}'$, meaning that the same expansion is used to uncover objects from seed nodes as is used to determine its substructures, albeit, with different start nodes and different expansion depths.

We can now enumerate all subgraphs by simply calculating this expression for all nodes of the graph, that is, object, and all values of d up to a chosen depth, that is, the set of subgraphs up to depth d contained in g is given by

$$\textsc{Subgraphs}^d_{\mathcal{E}'}(g) = \bigcup_{i=1}^{d} \bigcup_{v \in V} \textsc{Subgraph}^i_{\mathcal{E}'}(v)$$

where V is the node set of g and, as before, $\mathcal{E}'$ is an expansion. As a final result of this step, we obtain the set of subgraphs T_x given by $T_x := \textsc{Subgraphs}^d_{\mathcal{E}'}(x)$ for each object $x \in \mathcal{O}$, where the set of objects $\mathcal{O}$ is extracted as described in the previous Section.

3.5.3 Mapping objects to vectors

We now have access to the set of objects, as well as the set of substructures contained in each object, which is all that is required to finally map objects to vectors. In particular, no further information needs to be retrieved from the code property graph at this point as all subgraphs relevant for our embedding are available.

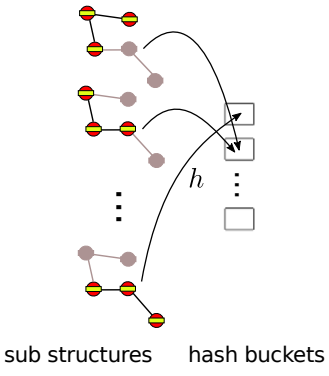


FIGURE 3.8: Feature hashing for sub structures

The task which remains is to choose an embedding language L and a mapping s from labeled substructures to words of the language. To this end, any of the definitions of L and s provided in the previous Section can be employed, provided that they are applicable to the enumerated substructures. For example, if the substructures are simply single nodes labeled by a string value, we can employ the definitions of L and s given for token- or symbol-based feature maps (see Sections 3.4.1 and 3.4.2) are applicable, while for trees, the corresponding definitions in Section 3.4.3 can be used. In cases where these mappings rely on labeled graphs, they can make use of the labels introduced in the process described in the previous Section.

While throughout the remainder of the thesis, we use different embedding languages and associated mappings from substructures to words, a truly generic embedding procedure can be obtained by employing a hash-based embedding as introduced in Section 3.4.4. This generality is achieved as hashes can be calculated for arbitrary labeled graphs, regardless of whether they consist of multiple or a single node or whether their edges are ordered or not. An additional advantage is that we can thus embed objects independently.

Recalling neighborhood hashing, the words of L are simply the numbers from 1 to N, where each number corresponds to a hash bucket of a hash function h such as the neighborhood hash introduced in Section 3.4.4. The function s that maps substructures to words then simply assigns the hash value of a substructure's designated root node to the structure, where the root node is identified by its label. Figure 3.8 illustrates this.

3.6 Related Work

Applying machine learning techniques to increase computer security has been considered by many authors in the past [see 67, 90], in particular for intrusion detection [e.g., 80, 116, 154], as well as malware detection [8, 70, 117]. However, most of these approaches deal with processing of network communication or binary code, as source code is typically not available in these settings. An exception is the work by Rieck et al. [119] for the detection of drive-by-download attacks where a token-based feature map (see Section 3.4.1) is employed to map Javascript code to a vector space.

In contrast, applications of machine learning for statically identifying vulnerabilities are rare, and are currently limited to supervised approaches. For example, Neuhaus et al. [102] train a support vector machine to rank *components* of the Web browser Firefox by their likelihood of being vulnerable, where a component consists of a C++ source file along with headers of the same name. Their representing employs is based on expert features such as imports and outgoing function calls, detected by scanning code for the pattern `identifier ( ...)` in lack of a fuzzy parser. Similarly, [130] train a classifier based on a bag-of-words representation obtained by tokenizing components. Moreover, Perl et al. [108] tokenize commits and train a support vector machine to identify commits that potentially introduce a vulnerability. [170] whether software metrics can be used for vulnerability prediction, where entire programs are classified. There are two main drawbacks to these approaches. First, they rely on the availability of a large number of labeled examples, and therefore, they can only be applied to code where a security history can be automatically queried. Second, and more importantly, analyzing entire components or commits is rather coarse-grained, possibly leaving the analyst with thousands of lines of code to review for each prediction.

4 Discovering Vulnerabilities using Dimensionality Reduction

With our platform for robust code analysis at hand, and a generic procedure for embedding code property graphs (see Chapters 2 and Chapter 3), we are now ready to develop concrete methods for pattern-based vulnerability discovery. In the following three chapters, we present three of these methods, each dedicated to showing the merits of one of the three major approaches to unsupervised learning in the context of vulnerability discovery, namely, dimensionality reduction, anomaly detection, and clustering. To this end, we formulate concrete problems encountered by analysts when reviewing code for vulnerabilities, and provide methods to address them.

In this chapter, we present a method for *finding vulnerabilities similar to a known vulnerability*, which is primarily based on dimensionality reduction. Our method extracts structural patterns from the code base and represents functions as compositions of these patterns. This allows us to decompose a known vulnerable function into its programming patterns and thereby identify functions employing similar programming patterns — potentially suffering from the same flaw. In particular, this method enables us to find several previously unknown vulnerabilities similar to a known vulnerability in popular open source projects.

We begin by discussing the setting addressed by our method and introduce the concept of *vulnerability extrapolation* in Section 4.1. We proceed to outline the capabilities of dimensionality reduction techniques with a focus on possible applications in code analysis in Section 4.2. Section 4.3 subsequently introduces the reader to a well known technique from natural language processing known as latent semantic analysis, which employs dimensionality reduction at its core. This method forms the algorithmic basis for our approach to vulnerability extrapolation, which we present in Section 4.4. Finally, the practical merits of our method are demonstrated empirically in Section 4.5, and we conclude by discussing related work in Section 4.6.

4.1 Task: Vulnerability Extrapolation

For our first method, we consider the initial phase of a code audit where the analyst knows only very little about the target code base. In this situation, it is often helpful and educating to review the program's security history by studying flaws identified by

others in the past, e.g., by browsing vulnerability databases or revision histories. The rational behind this is that vulnerabilities are often linked to specific programming errors, and that a given programming error is often made more than once throughout a code base. In the simplest case, vulnerable code is copied as-is throughout the development process, creating a vulnerable *code clone* [see 13, 14, 74], however, copied code can also be modified, raising the problem difficulty. In the most difficult case, code is not copied at all, and similar vulnerabilities are instances of an incorrect *programming pattern* based on false programmer assumptions. For example, a programmer may assume that an API function is able to deal with malformed data, while in fact, it is not, resulting in several instances of the same vulnerability.

This setting calls for a method that, given a known vulnerability, automatically identifies locations in the code employing the same programming patterns—possibly containing the same flaw. We refer to this strategy of vulnerability discovery as *vulnerability extrapolation* (see [163, 164]) as existing knowledge of problematic programming patterns concentrated in a vulnerability is extrapolated to find vulnerabilities in the rest of the code base. This approach is attractive because it is not specific to any particular type of vulnerability, and does not rely on advanced machinery, requiring only a robust parser and an initial vulnerability.

In many cases, vulnerability extrapolation can be performed manually by specifying rules that capture the vulnerable properties as explored in Chapter 2. However, this requires an expert to study the vulnerability in order to understand its core properties *before* a suitable rule can be formulated. The amount of manual work this requires calls for an automated approach.

Assuming that no information is available apart from the vulnerability itself, extrapolation the vulnerability poses two primary challenges. First, programming patterns existent in the code base need to be extracted automatically, and second, functions employing similar programming patterns to the known vulnerable function need to be determined. To address these challenges, our method for vulnerability extrapolation adapts latent semantic analysis, a well understood technique originally developed in natural language processing for the discovery of similar text documents.

4.2 Dimensionality Reduction

A primary challenge in applied machine learning is to choose features that are well suited to express those properties of the data relevant for the task at hand. This is true for numeric features, but even more so for the *bag-of-words*-like representations we employ for code analysis (see Chapter 3). In these spaces, the dimensionality of the feature space is equal to the number of different words in the data, making the resulting feature vectors high dimensional and sparse. This has the following three severe consequences.

First, and less problematic for sparse feature spaces, dealing with large feature spaces can easily become prohibitive due to time and memory constraints. Second, many features are possibly irrelevant, and thus, they only introduce extra noise. And third, the amount of data required to generate a prediction function that expresses statistically significant properties of the data typically grows with the dimensionality of the feature space, and therefore, a feature space with fewer dimensions is desirable.

These problems can be alleviated by *dimensionality reduction*. Given data in a high dimensional feature space, dimensionality reduction techniques seek another space with a lower dimensionality where the data can be represented with comparative expressiveness, however, with fewer features.

There are two main strategies for dimensionality reduction, feature selection and feature extraction. While in the former case, we restrict ourselves to selecting a sub set of the existing features, in the latter, we construct new features from the existing. As the solutions obtainable via feature extraction include sub sets of features, feature extraction can be considered the more general of the two strategies. However, in situations where newly constructed features do not lend themselves to simple interpretation and the interpretation of features is vital, feature selection can be the better choice [27]. As newly constructed features created from feature spaces considered in this work are designed to be interpreted easily, we focus our discussion on feature extraction.

There are a large variety of feature extraction methods, both limited to exploiting linear dependencies (e.g., Principal Component Analysis, Non-negative Matrix Factorization, Linear Discriminant Analysis), and algorithms capable of dealing with non-linear dependencies (e.g., Kernel PCA [131], Isomap [145], and Locally Linear Embedding [125]). With respect to our description of machine learning (Section 1.2), these algorithms each implement the same action. For a given input space $\mathcal{X}$, the methods extract features, which are the sole parameters for the creation of a model θ. Using this model, we can instantiate a prediction function f_θ that projects a data point $x \in \mathcal{X}$ into the newly created feature space of reduced dimensionality $\mathcal{Y}$.

As we discuss in more detail in the following Sections, Principal Component Analysis is a technique for dimensionality reduction particularly interesting in the setting of code analysis. To construct a feature space of reduced dimensionality, Principal component analysis extracts directions in the feature space where the data varies most, that is, directions that represent strong correlations between features of the original space. Applied to code analysis and with the embeddings described in Chapter 3 in mind, this enables us to determine common compositions of program constructs. With these features at hand, we can easily identify code sharing the same common compositions of program constructs. Moreover, we can represents objects of the program in light of the overall code base, making it possible to take into account which of an object's features constitute a pattern, and which are merely noise. These merits of dimensionality reduction have been successfully exploited in natural language processing to identify patterns in textual data. In particular, a method named *latent semantic analysis* has evolved to become a standard technique. This method forms the basis for our approach, and thus, we describe it in the following.

4.3 Latent Semantic Analysis

Latent semantic analysis, also commonly referred to as latent semantic indexing [32], is a technique based on dimensionality reduction, developed to identify *topics of discussion* in sets of text document, as well as documents similar to a given document in terms of topics. To this end, text documents are represented in a vector space and dimensionality reduction is performed to find a lower dimensional representation of text documents in terms of the most common topics discussed in the corpus. This is achieved in the following three steps.

- **Embedding.** Documents are first embedded in a vector space using a bag-of-words embedding (see Section 3.2), where the weight v_w for each word w corresponds to the TF-IDF weighting. This classic weighting scheme from natural language processing removes the bias towards longer documents, and ensures that words frequent in the corpus have very little effect on document representation. A detailed description of the TF-IDF weighting scheme is given by Salton and McGill [127].

- **Dimensionality Reduction.** Second, common topics of discussion are determined from the corpus via dimensionality reduction. To this end, we identify dominant directions in feature space, that is, combinations of words that commonly occur together, by calculating a *truncated singular value decomposition* of the data matrix. Projecting document vectors onto these directions provides us with a representation of documents in terms of the topics they contain.

- **Similarity Assessment.** Finally, we can determine the similarity between two documents geometrically by calculating the distance between the corresponding vectors using a suitable distance function. Latent semantic analysis employs the cosine distance
$$\cos(x, y) = 1 - \frac{x \cdot y}{\|x\| \cdot \|y\|}$$
for this purpose, where x and y are the d-dimensional representations of the two documents and $\|\cdot\|$ denotes the L_2 norm. While technically, the cosine distance is not a metric as the triangle inequality does not hold, it is a popular choice for document comparison nonetheless due to the fact that it normalizes documents by their length, thereby removing the bias towards longer documents.

At heart of this procedure, dimensionality reduction is employed to *denoise* the data set, that is, to obtain a representation less susceptible to artifacts.

4.4 Extrapolation using Syntax Trees

Latent semantic analysis provides us with a method to extract prominent topics from a corpus of text documents, and additionally allows us to identify documents that are similar with respect to these topics. The problem we need to address when extrapolating vulnerabilities is similar: we want to extract prominent programming patterns presents in a code base, and identify functions similar in terms of these programming patterns. In analogy to the representation of *topics* by combinations of words in text documents, we therefore propose to describe *programming patterns* as combinations of AST sub trees.

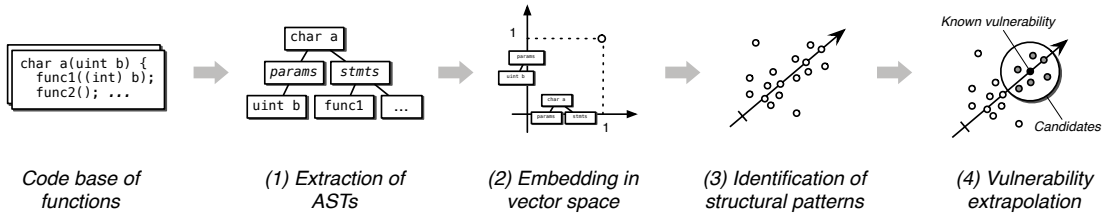


FIGURE 4.1: Overview of our method for vulnerability extrapolation [164]

Based on this idea, we can formulate our method for vulnerability extrapolation as a four step procedure in close resemblance to latent semantic analysis. This procedure is illustrated in Figure 4.1 and outlined in the following.

1. **Extraction of syntax trees.** We begin by extracting abstract syntax trees for all functions of the code base using a suitable graph-database lookup (Section 4.4.1).
2. **Embedding of syntax trees.** Abstract syntax trees are subsequently embedded in a vector space to allow for applications of machine learning techniques. To this end, we employ concrete implementations of our embedding procedure for property graphs (Section 4.4.2).
3. **Identification of structural patterns.** We proceed to identify the most dominant structural patterns in the code base by performing the same dimensionality reduction technique to the vectorial representations of functions as employed by latent semantic analysis to text documents (Section 4.4.3).
4. **Vulnerability extrapolation.** Finally, functions of the code base are represented as linear-combinations of the most dominant structural patterns they contain, allowing us to identify functions similar to a vulnerable function in terms of structural patterns (Section 4.4.4).

In the following, we describe each of these steps in greater detail and discuss how they are implemented with respect to our robust code analysis platform (see Chapter 2).

4.4.1 Extraction of Syntax Trees

We implement our method for vulnerability extrapolation on top of the architecture for robust code analysis presented in Chapter 2. In particular, we thus have access to code property graphs for each function and can employ the general embedding procedure for code property graphs introduced in Chapter 3.

The first step of the embedding procedure is to define and extract the objects to process, that is, we need to define an input space $\mathcal{X}$ (see Section 3.5). For example, this can be the space of all source files, all functions, or all statements. This choice is crucial for the success of our method as it decides upon the granularity of the analysis. On the one hand, we want to narrow in on vulnerable code as much as possible, making small objects preferable. On the other, we want an object to capture all aspects relevant for the vulnerability as well as the context it appears in, an argument for larger objects.

Functions offer a good compromise between these two objectives. Like source files, they are explicitly named grouping constructs that enclose logically related code. In addition, they are the most fine-grained, named grouping construct available in the C programming language, and thus better support our desire for small grouping constructs than entire source files.

We characterize functions by their abstract syntax trees as these trees are the most fine-grained code representation made available to us by the code property graph. These trees faithfully encode program syntax by providing a hierarchical decomposition of functions into their language elements. Moreover, with our embedding procedure on

code property graphs at hand, extracting these trees can be achieved easily by (a) defining a seed node selector S that retrieves all function nodes, and (b) defining an expansion $\mathcal{E}$ that recursively expands these nodes into syntax trees. Seed node selection thus boils down to the evaluation of the function

$$S(X) = \{\{x\}, x \in \textsc{Lookup}(\textit{type}, \textit{function}, X)\}.$$

On an implementation level, this corresponds to a simple index lookup that retrieves all nodes where the *type* is *function*. As the Neo4j graph database keeps an index of nodes by property, this operation executes in constant time.

The nodes returned by the seed node selector now serve as seed nodes to uncover their corresponding abstract syntax trees. We achieve this by instantiating the function $\textsc{Object}_{\mathcal{E}}^{\mathcal{D}}$ (see Section 3.5.1) with the expansion $\mathcal{E} := \textsc{OutE}_{\mathcal{A}}$ and $\mathcal{D} := \infty$, that is, we walk along syntax edges until expansion is no longer possible.

4.4.2 Embedding Syntax Trees

Upon extracting syntax trees for all functions of the code base, we proceed to map each syntax tree to a corresponding vector encoding the sub trees contained in the function.

The design of this mapping is crucial for the success of vulnerability extrapolation, and since it is not clear beforehand which mapping produces good results in practice, we consider three competing mappings, which differ in the types of AST sub trees they employ to model functions. In particular, we consider the following mappings in this work, where $\mathcal{T}$ denotes the set of sub trees.

- **API nodes.** We represent each function by the API symbols it contains, i.e., by the names of types and callees. This corresponds to the embedding presented in the original paper on vulnerability extrapolation [163]. The set of sub trees $\mathcal{T}$ corresponds to the set of API nodes present in the code base.
- **API subtrees.** Each function is represented by its sub trees up to a depth $\mathcal{D}$, however, only the contents of API symbol nodes is preserved. The set of sub trees $\mathcal{T}$ is thus given by all AST sub trees of the code base up to a depth $\mathcal{D}$ that contain at least one API symbol, where ASTs are preprocessed to replace all non-API nodes with empty nodes.
- **API/S subtrees.** We represent each function by its sub trees up to a depth $\mathcal{D}$ and preserve the contents of all nodes, that is, API symbols, as well as syntactical elements such as arithmetic and assignment operators. The set of sub trees $\mathcal{T}$ thus corresponds to that of all AST sub trees up to a depth $\mathcal{D}$ containing at least one API or syntax node, where again, ASTs are preprocessed to purge the contents of all other types of nodes.

Defining the set of sub trees $\mathcal{T}$ is sufficient to directly employ a *tree-based feature map* (see Section 3.4.3) in order to represent all functions of the code base by the sub trees of their syntax tree. To this end, sub trees are simply mapped to string representations using a function $s : \mathcal{T} \to L$, that is, the embedding language L is given by the set of string representations of ASTs. The desired feature map is then given by a function

$\phi : \mathcal{X} \to \mathbb{R}^{\|L\|}$ from functions, represented by their syntax trees, to $|L|$ dimensional vectors, where $|L|$ corresponds to the number of words in the embedding language. As is true for all bag-of-words embeddings, this function is defined as

$$\phi(x) = \left(\#_w(x) \cdot v_w\right)_{w \in L}$$

where $\#_w(x, s)$ corresponds to the number of sub trees of x that are mapped to the word w, and v_w is the TF-IDF weighting term (see Section 4.3).

We can implement this embedding on code property graphs based on the generic embedding procedure described in Section 3.5. This is achieved by first labeling the nodes of the AST using a labeling function, extracting its sub trees using a suitable sub graph enumerator, and finally, hashing each sub graph to associate it with a dimension of the feature space.

For the first two embeddings, the labeling function must preserve the *code* attribute of API nodes while purging the information in all other nodes. This can be achieved using the labeling function

$$l_{\text{API}}(x) = \begin{cases} \mu(x, \mathit{code}) & \text{if } \mu(x, \mathit{type}) \in A \\ \epsilon & \text{otherwise} \end{cases}$$

where A is the set of all node types that correspond to API nodes. Labeling for the third embedding is simpler as each node is simply labeled by its *code* attribute, that is we apply the labeling function $l_{\text{API/S}}(x) = \mu(x, \mathit{code})$.

Once syntax trees are labeled, we define a sub graph enumerator to extract the sub graphs they contain. To this end, we instantiate the function $\textsc{SubGraphs}^d_{\mathcal{E}}$ (see Section 3.5.2) using an expansion that follows syntax tree edges only, that is, we set $\mathcal{E} := \textsc{OutE}_{\mathcal{A}}$ and $d := 1$ for the first embedding, and $d := 3$ for the second and third embedding. Finally, objects are mapped to vectors by mapping sub structures to words using the function s, where we ignore any sub trees that are mapped to the empty string by s.

Each of the presented embeddings provides a different view on the functions of the code base. While API nodes alone only provide information about the interfacing of functions, API sub trees additionally describe the structural context these nodes occur in, and finally, API/S subtrees augment this representation with the syntactical elements employed by functions.

```
1  int foo(int y)
2  {
3      int n = bar(y);
4
5      if (n == 0)
6          return 1;
7
8      return (n + y);
9  }
```

FIGURE 4.2: Sample code of a function `foo` [164]

To illustrate the embedding procedure, let us consider an API node embedding and the sample code provided in Figure 4.2 along with a simplified illustration of its abstract

syntax tree in Figure 4.3. The syntax tree contains three API nodes, namely, the parameter of type `int`, the declaration of type `int` and the call to the function `bar`. The resulting sparse vector therefore contains only three non-zero dimensions while all other dimensions are set to zero. While the vector space has a high dimensionality, its sparsity allows these vectors to be efficiently represented and compared in linear time using data structures such as sorted arrays and hash maps [see 120].

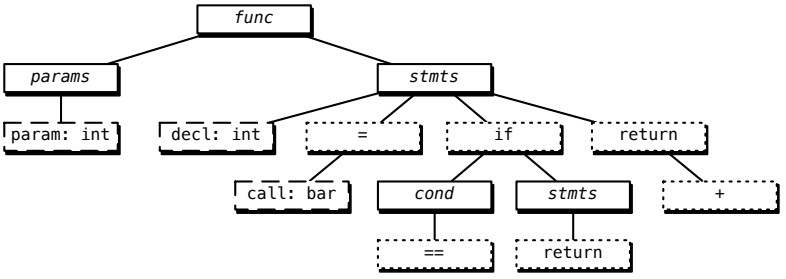


FIGURE 4.3: Abstract syntax tree for the function `foo` [164]

As a result of this step, we obtain a sparse M by N data matrix X, where M corresponds to the size of the hash function's co-domain and N is equal to the number of functions of the code base. Following latent semantic analysis, we finally apply the TF-IDF weighting scheme to this matrix (see Section 4.3) to remove the bias towards longer functions and reduce the effect of very common sub trees.

4.4.3 Identification of Structural Patterns

The representation of functions introduced in the previous section describes functions in terms of sub trees of their abstract syntax trees, a representation created in direct analogy to the representation of text documents by the words they contain. We proceed further along the path set by latent semantic analysis and calculate the truncated singular value decomposition of the data matrix X as $X \approx U\Sigma V^T$. Just like in the text mining setting, we thus obtain directions in the feature space where the data varies most, as well as a representation of functions in terms of these directions. However, the interpretation of these directions differ from those in text mining setting. While in text mining, these directions correspond to common combinations of words, that is, topics of discussion, in our setting, these are common combinations of sub trees, that is, *structural patterns*.

Formally, we seek the d orthogonal dimensions in feature space that capture most of the data's variance. For a term-by-document matrix X, this can be achieved by calculating its *truncated singular value decomposition*, given by:

$$X \approx U\Sigma V^T = \begin{pmatrix} \leftarrow u_1 \rightarrow \\ \leftarrow u_2 \rightarrow \\ \vdots \\ \leftarrow u_{|\mathcal{S}|} \rightarrow \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_d \end{pmatrix} \begin{pmatrix} \leftarrow v_1 \rightarrow \\ \leftarrow v_2 \rightarrow \\ \vdots \\ \leftarrow v_{|\mathcal{X}|} \rightarrow \end{pmatrix}^T .$$

These three matrices in combination allow the similarity of functions to be assessed and the most prominent structural patterns to be determined. In particular, the matrices contain the following information.

- The unitary matrix U contains the d most dominant directions of the feature space in its columns, that is, the vectorial representation of the structural patterns.
- The diagonal of Σ gives access to the corresponding singular values, which are a measure for the importance of each of the d structural patterns stored in U.
- Finally, V contains the d-dimensional representations of functions in its rows, that is, each function is represented as a mixture of the d most dominant directions in feature space.

In the context of latent semantic analysis, this decomposition is typically calculated using the Lancsoz algorithm, an iterative procedure implemented by both classic libraries for linear algebra such as SVDPACK [15] as well as modern machine learning frameworks such as Apache Mahout [42].

4.4.4 Extrapolation of Vulnerabilities

Upon calculating the truncated singular value decomposition as outlined in the previous section, the analyst can make use of the resulting matrices U, Σ and V for vulnerability discovery and vulnerability extrapolation in particular.

The matrix V plays the key role in vulnerability extrapolation. In its rows, it holds a vector for each of the program's functions, describing the function as a linear combination of the most prominent structural patterns present in the code base. The analyst can make use of these vectors to compare functions in terms of the structural patterns they employ. In particular, to extrapolate a vulnerability, the analyst simply chooses the vector corresponding to the vulnerable function and compares it with each of the other row vectors in V using a suitable distance metric. Choosing those vectors with the smallest distance, the analyst obtains those functions of the code base most similar to the vulnerable function in terms of structural patterns, and thus prime candidates for functions suffering from the same flaw. To achieve this, we use the cosine distance as in the original algorithm for latent semantic analysis (see Section 4.3).

While extrapolation can be performed using the matrix V alone, the matrix U can also be of use to the analyst. In its columns, it stores the most prominent structural patterns, allowing insights into the structure of the code base to be gained. For example, this information can be used to uncover major clusters of similar functions that employ similar programming patterns, making it possible for the analyst to focus on interested parts of the code base early in the analysis.

Finally, the vector space of reduced dimensionality spanned by structural patterns can serve as an input space for subsequent anomaly detection techniques. This application is discussed in detail in Chapter 5 where we see that certain deviations from common programming patterns constitute vulnerabilities and that they can often be determined automatically.

4.5 Evaluation

In the following, we evaluate our method for vulnerability extrapolation on the source code of four well known open source projects from different areas of application. In

particular, we assess our method's ability to identify similar functions and find vulnerabilities in practice. To this end, we carry out an evaluation in two steps. First, a controlled experiment is conducted based on ground truth, and second, the merits of our method for real world vulnerability discovery are demonstrated in two case studies, where we identify several previously unknown vulnerabilities.

For our evaluation, we consider the code bases of the image processing library LibTIFF, the video and audio decoding library FFmpeg, the instant messenger Pidgin, and the Voice-over-IP framework Asterisk. In each case, a known vulnerability is chosen and we manually label *candidate functions* which should be reviewed for the same type of vulnerability. In the following, we describe our data set in detail.

1. LibTIFF (`http://www.libtiff.org`) is an image processing library for the Tagged Image File Format, consisting of 1,292 functions and 52,650 lines of code. In Version 3.8.1., the library contains an exploitable stack-based buffer overflow in the parsing of TLV elements (CVE-2006-3459). As candidate functions, we select all parsers of TLV elements.

2. FFmpeg (`http://www.ffmpeg.org`) is a library for audio and video decoding that spans 6,941 functions and a total of 298,723 lines of code. Version 0.6 contains, an incorrect calculation of indices during video frame decoding allows attackers to execute arbitrary code (CVE-2010-3429). As candidate functions, we select all video decoding routines that write decoded video frames to a pixel buffer.

3. Pidgin (`http://www.pidgin.im`) is an instant messenger consisting of 11,505 functions and 272,866 lines of code. In version 2.10.0, the program contains a remote denial-of-service vulnerability in the implementation of the AOL instant messaging protocol (CVE-2011-4601). Candidate functions are all AIM protocol handlers that convert incoming binary messages to strings.

4. Asterisk (`http://www.asterisk.org`) is a platform for Voice-over-IP telephony, covering a total of 8,155 functions and 283,883 lines of code. Version 1.6.1.0 contains a memory corruption vulnerability allowing attackers to remotely crash the Asterisk server and possibly execute arbitrary code (CVE-2011-2529). We select all functions that read incoming packets from UDP/TCP sockets as candidates.

4.5.1 Controlled Experiment

We begin with a controlled experiment based on ground truth data. To obtain this data, we thoroughly inspect code bases for candidate functions manually, that is, for a given vulnerability, we identify functions that should be checked for the same vulnerability by manually inspecting the code. It should be noted that this manual process does not offer a realistic alternative for vulnerability extrapolation, as it required several weeks of manual work.

Upon collecting ground truth data, we execute our method to rank all functions of the code base with respect to their similarity to the known vulnerable function. Assuming that the analyst examines functions by strictly following the generated ranking, this allows us to assess the number of functions the analyst must audit before finding all candidates. We carry out this experiment for all three embeddings described in Section 4.4.2, making it possible to compare their effectiveness.

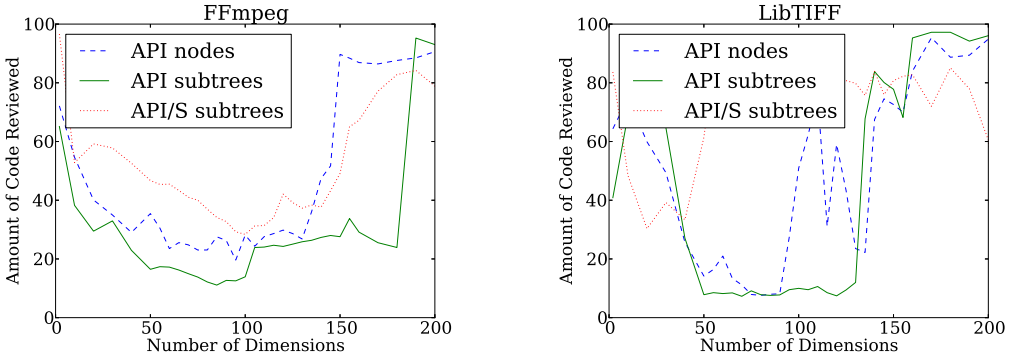


FIGURE 4.4: Performance of vulnerability extrapolation in a controlled experiment [164].

Figure 4.4 shows the amount of code requiring review before all candidates are found, plotted against the number of structural patterns chosen to represent the code base. The corresponding plots for the remaining two code bases are similar and have thus been omitted. As the Figure shows, the embedding based on API subtrees clearly outperforms the other two representations as it allows all candidate functions to be discovered by auditing only 8.7% of the code on average, as compared to 11.1% for flat API symbols and 25% for API/S subtrees. Moreover, the results show that the optimal choice of the number of structural patterns depends on the code base, however, choosing it within a band between 50 to 100 dimensions yields good results for all the the code bases studied despite their different sizes. Therefore, the number of dimensions is not a critical parameter and is fixed to 70 for the real world experiments described in the following sections.

Table 4.1 offers a more fine-grained analysis of our method, showing the percentage of code to review to uncover 75%, 90% and 100% of the candidates for each code base and each embedding. Again, the API subtrees show the best performance, allowing 75% of the candidates to be discovered by reading only 3% of the code on average. For Pidgin and Asterisk, reading less than 1% suffices to achieve these results. However, there is also room for improvement, particularly when all candidates need to be identified, where the amount of code to read reaches 16% in the worst case.

	API nodes			**API subtrees**			**API/S subtrees**		
	75%	90%	100%	75%	90%	100%	75%	90%	100%
Pidgin	0.1	0.36	2.00	0.35	0.22	0.98	0.22	0.67	25.98
LibTIFF	6.35	6.97	7.58	5.65	6.66	7.27	6.49	9.36	17.32
FFmpeg	6.17	8.10	19.61	5.00	8.66	11.09	7.71	15.21	28.35
Asterisk	0.06	10.64	15.29	0.24	10.23	15.54	1.19	16.50	28.45
Average	3.17	6.52	11.12	2.81	6.44	8.72	3.90	10.44	25.03

TABLE 4.1: Performance of vulnerability extrapolation in a controlled experiment. The performance is given as amount of code (%) to be audited to find 75%, 90% and 100% of the potentially vulnerable functions [164].

Overall, these results indicate that the task of finding functions possibly containing a flaw similar to that in a known vulnerable function can be accelerated significantly by our method for vulnerability extrapolation when compared to manual auditing. However, a controlled experiment cannot provide insights on the practical merits of the approach.

In the following, we therefore perform several case studies showing that our method plays the key role in uncovering several previously unknown vulnerabilities.

4.5.2 Case Studies

We proceed to evaluate the practical merits of our method by applying it to the source code of FFmpeg and Pidgin, allowing us to uncover eight previously unknown vulnerabilities by extrapolation two known vulnerabilities.

4.5.2.1 Case Study: FFmpeg

Memory corruption vulnerabilities are a common problem among libraries that parse and decode media formats in memory-unsafe languages such as C or C++. In particular, unchecked array indices used in write operations can allow attackers to write arbitrary data to arbitrary locations in memory, a powerful exploit primitive that can often be leveraged for code execution. This case study considers a bug of this kind in the video decoder for FLIC media files implemented as part of FFmpeg (CVE-2010-3429). As this exact vulnerability is also considered as a case study in our original paper on vulnerability extrapolation [163], it allows us to directly compare our original method to the improved method based on structural patterns.

```
static int flic_decode_frame_8BPP(AVCodecContext *avctx,
                                  void *data, int *data_size,
                                  const uint8_t *buf,
                                  int buf_size)
{ [...]
  signed short line_packets; int y_ptr;
  [...]
  pixels = s->frame.data[0];
  pixel_limit = s->avctx->height * s->frame.linesize[0];
  frame_size = AV_RL32(&buf[stream_ptr]); [...]
  frame_size -= 16;
  /* iterate through the chunks */
  while ((frame_size > 0) && (num_chunks > 0)) { [...]
    chunk_type = AV_RL16(&buf[stream_ptr]);
    stream_ptr += 2;
    switch (chunk_type) { [...]
    case FLI_DELTA:
        y_ptr = 0;
        compressed_lines = AV_RL16(&buf[stream_ptr]);
        stream_ptr += 2;
        while (compressed_lines > 0) {
            line_packets = AV_RL16(&buf[stream_ptr]);
            stream_ptr += 2;
            if ((line_packets & 0xC000) == 0xC000) {
                // line skip opcode
                line_packets = -line_packets;
                y_ptr += line_packets * s->frame.linesize[0];
            } else if ((line_packets & 0xC000) == 0x4000) {
                [...]
            } else if ((line_packets & 0xC000) == 0x8000) {
                // "last byte" opcode
                pixels[y_ptr + s->frame.linesize[0] - 1] =
                   line_packets & 0xff;
            } else { [...]
                y_ptr += s->frame.linesize[0];
            }
        }
        break; [...]
    }
  [...]
  } [...]
  return buf_size;
}
```

```
static void vmd_decode(VmdVideoContext *s)
{
  [...] int frame_x, frame_y;
  int frame_width, frame_height;
  int dp_size;

  frame_x = AV_RL16(&s->buf[6]);
  frame_y = AV_RL16(&s->buf[8]);
  frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
  frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;

  if ((frame_width == s->avctx->width &&
       frame_height == s->avctx->height) &&
      (frame_x || frame_y)) {
    s->x_off = frame_x;
    s->y_off = frame_y;
  }
  frame_x -= s->x_off;
  frame_y -= s->y_off; [...]
  if (frame_x || frame_y || (frame_width != s->avctx->width)
      || (frame_height != s->avctx->height)) {
       memcpy(s->frame.data[0], s->prev_frame.data[0],
          s->avctx->height * s->frame.linesize[0]);
  } [...]
  if (s->size >= 0) {
      pb = p;
      meth = *pb++; [...]
      dp = &s->frame.data[0][frame_y * s->frame.linesize[0]
          + frame_x];
      dp_size = s->frame.linesize[0] * s->avctx->height;
      pp = &s->prev_frame.data[0][frame_y *
             s->prev_frame.linesize[0] + frame_x];
      switch (meth) { [...]
      case 2:
        for (i = 0; i < frame_height; i++) {
            memcpy(dp, pb, frame_width);
            pb += frame_width;
            dp += s->frame.linesize[0];
            pp += s->prev_frame.linesize[0];
        } break; [...]
        }
    }
}
```

FIGURE 4.5: Original vulnerability in `flic_decode_frame_8BPP` (left) and zero-day vulnerability found in `vmd_decode` (right) [164].

Original vulnerability. Figure 4.5 shows the original vulnerability located in the function `flic_decode_frame_8BPP`, a decoding function for FLIC videos. The function

operates on *video frames* containing encoded image data and meta information such as the image dimensions and offsets. Ultimately, the decoder's task is to create a buffer containing the decoded pixel data. To this end, it proceeds to allocate the buffer in accordance with the frame dimensions, index the array using the frame offsets, and finally, write decoded pixels to the buffer. Before performing the write operation, it is crucial to verify that the location referred to by the frame offsets is within the boundaries of the buffer. Unfortunately, the vulnerable functions does not verify frame offsets at all, enabling attackers to corrupt memory.

The function implements a taint-style vulnerability (see Section 1.1.2), where data propagates from an attacker controlled source to a sensitive operation without undergoing prior verification. In this case, the critical operation is the write operation on line 32 where the least significant byte of the attacker-controlled integer `line_packets` is written to a location relative to the pixel buffer `pixels`. The location to which the data is written depends on the integer `y_ptr`. This integer stems from the attacker-controlled source `AV_RL16` that reads a 16 bit integer from the media file, and, as is defining for a taint-style vulnerability, it is not checked before it reaches the sensitive sink.

Sim.	Function name
0.98	flic_decode_frame_15_16BPP
0.92	decode_frame
0.92	decode_frame
0.91	flac_decode_frame
0.90	decode_format80
0.89	decode_frame
0.89	tgv_decode_frame
0.89	vmd_decode
0.89	wavpack_decode_frame
0.88	adpcm_decode_frame
0.88	decode_frame
0.88	aasc_decode_frame
0.88	vqa_decode_chunk
0.87	cmv_process_header
0.87	msrle_decode_8_16_24_32

Sim.	Function name
0.87	wmavoice_decode_init
0.85	decode_frame
0.84	smc_decode_stream
0.84	rl2_decode_init
0.84	xvid_encode_init
0.84	vmdvideo_decode_init
0.83	mjpega_dump_header
0.82	ff_flac_is_._valid
0.82	decode_init
0.82	ws_snd_decode_frame
0.81	bmp_decode_frame
0.81	sbr_make_f_master
0.80	ff_h264_decode_ref_pic_.
0.80	decode_frame
0.79	vqa_decode_init

TABLE 4.2: Top 30 most similar functions to a known vulnerability in FFmpeg [164].

Extrapolation. Executing our method with the vulnerable function as input, we obtain the ranking shown in Table 4.2. This ranking shows the 30 functions most similar to the vulnerable function selected from a total of 6,941 functions, where light shading and dark shading indicate candidates and vulnerabilities respectively. Reviewing these results, we find the following.

- Of the 30 most similar functions, 20 are candidate functions, meaning that they are, too, decoders writing data to a pixel buffer. In addition, all of the first 13 functions are candidates.
- The function `flic_decode_frame_15_16BPP` is found to be most similar to the vulnerable function.Indeed, this function also processes FLIC video frames and was fixed along with the original flaw.

- The function `vmd_decode` shown in Figure 4.5 likewise contains a taint-style vulnerability very similar to the previous two. Like in the original vulnerability, a frame offset is retrieved from the attacker-controlled source `AV_RL16` on line 8 and used as an index into a pixel buffer on line 28. Finally, a write operation is performed at the designated offset without prior validation of the frame offset. Our method reports a similarity of 89%, leading us almost directly to this vulnerability. As demonstrated in our prior work [163], this vulnerability can be found based on API information alone.

- Finally, the function `vqa_decode_chunk` contains another vulnerability, which also was not identified by the method proposed in our prior work [163]. This vulnerability is difficult to identify based on API information alone as reading from the attacker-controlled source using the characteristic API is not performed inside the vulnerable function but by the initialization routine `vqa_decode_init` on 21 and 22. However, we can compensate the missing API information in the vulnerable function by additionally taking into account its structure, and in particular, the nested loop iterating over the pixel buffer on line 6 and 8, a typical pattern for frame decoding routines.

```
1  static void vqa_decode_chunk(VqaContext *s)
2  {
3    [...]
4    int lobytes = 0;
5    int hibytes = s->decode_buffer_size / 2; [...]
6    for (y = 0; y < s->frame.linesize[0] * s->height;
7      y += s->frame.linesize[0] * s->vector_height){
8      for (x = y; x < y + s->width; x += 4,lobytes++,hibytes++)
9        {
10       pixel_ptr = x;
11       /* get the vector index, the method for
12          which varies according to
13        * VQA file version */
14       switch (s->vqa_version) {
15       case 1: [...]
16       case 2:
17       lobyte = s->decode_buffer[lobytes];
18       hibyte = s->decode_buffer[hibytes];
19       vector_index = (hibyte << 8) | lobyte;
20       vector_index <<= index_shift;
21       lines = s->vector_height;
22       break;
23       case 3: [...]
24        }
25       while (lines--) {
26         s->frame.data[0][pixel_ptr + 0] =
27           s->codebook[vector_index++];
28         s->frame.data[0][pixel_ptr + 1] =
29           s->codebook[vector_index++];
         s->frame.data[0][pixel_ptr + 2] =
           s->codebook[vector_index++];
         s->frame.data[0][pixel_ptr + 3] =
           s->codebook[vector_index++];
         pixel_ptr += s->frame.linesize[0];
         }
     }
   }
}

static av_cold int vqa_decode_init(AVCodecContext *avctx)
{
    VqaContext *s = avctx->priv_data;
    unsigned char *vqa_header;
    int i, j, codebook_index;
    s->avctx = avctx;
    avctx->pix_fmt = PIX_FMT_PAL8; [...]
      /* load up the VQA parameters from the header */
    vqa_header = (unsigned char *)s->avctx->extradata;
    s->vqa_version = vqa_header[0];
     s->width = AV_RL16(&vqa_header[6]);
    s->height = AV_RL16(&vqa_header[8]); [...]
    /* allocate decode buffer */
    s->decode_buffer_size = (s->width / s->vector_width) *
        (s->height / s->vector_height) * 2;
    s->decode_buffer = av_malloc(s->decode_buffer_size);
    s->frame.data[0] = NULL;
    return 0;
}
```

FIGURE 4.6: The second zero-day vulnerability found by extrapolation of CVE-2010-3429 in `vqa_decode_chunk` [164].

This case study illustrates that our method is capable of assisting an analyst in identifying vulnerabilities similar to known vulnerabilities in real world code. Moreover, it demonstrates that augmented structural information can be valuable in cases where API information is lacking.

4.5.2.2 Case Study: Pidgin

To ensure that our approach does not rely on any properties specific to FFmpeg or the vulnerability, we perform a second case study on a different vulnerability located in the packet handling code of the instant messenger Pidgin. Moreover, the fraction of candidate functions with respect to the total number of functions is considerable

```
static int
receiveauthgrant(OscarData *od,
                 FlapConnection *conn,
                 aim_module_t *mod,
                 FlapFrame *frame,
                 aim_modsnac_t *snac,
                 ByteStream *bs)
{
  int ret = 0;
  aim_rxcallback_t userfunc;
  guint16 tmp;
  char *bn, *msg;
  /* Read buddy name */
  if ((tmp = byte_stream_get8(bs)))
    bn = byte_stream_getstr(bs, tmp);
  else
    bn = NULL;
  /* Read message (null terminated) */
  if ((tmp = byte_stream_get16(bs)))
    msg = byte_stream_getstr(bs, tmp);
  else
    msg = NULL;
  /* Unknown */
  tmp = byte_stream_get16(bs);
  if ((userfunc =
       aim_callhandler(od, snac->family, snac->subtype)))
    ret = userfunc(od, conn, frame, bn, msg);
  g_free(bn);
  g_free(msg);
  return ret;
}
```

```
static int
parseicon(OscarData *od,
          FlapConnection *conn,
          aim_module_t *mod,
          FlapFrame *frame,
          aim_modsnac_t *snac,
          ByteStream *bs)
{

  int ret = 0;
  aim_rxcallback_t userfunc;
  char *bn;
  guint16 flags, iconlen;
  guint8 iconcsumtype, iconcsumlen, *iconcsum, *icon;

  bn = byte_stream_getstr(bs, byte_stream_get8(bs));
  flags = byte_stream_get16(bs);
  iconcsumtype = byte_stream_get8(bs);
  iconcsumlen = byte_stream_get8(bs);
  iconcsum = byte_stream_getraw(bs, iconcsumlen);
  iconlen = byte_stream_get16(bs);
  icon = byte_stream_getraw(bs, iconlen);
  if ((userfunc =
       aim_callhandler(od, snac->family, snac->subtype)))
    ret = userfunc(od, conn, frame, bn, iconcsumtype,
                   iconcsum, iconcsumlen, icon, iconlen);
  g_free(bn);
  g_free(iconcsum);
  g_free(icon);
  return ret;
}
```

FIGURE 4.7: Original vulnerability (CVE-2011-4601) in `receiveauthgrant` (left), zero-day vulnerability in `parseicon` (right) [164].

lower in this case, where only 67 of 11,505 functions are candidates. Regardless of this increased difficulty, extrapolation of the vulnerable function allows us to uncover nine similar vulnerabilities within the top 30 most similar functions, and in particular, six previously unknown vulnerabilities.

Original Vulnerability. The vulnerability resides in the function `receiveauthgrant` shown in Figure 4.7, where attacker-controlled strings are read from a network packet using the function `byte_stream_getstr` on line 15. The value thus read is then passed to a suitable packet handler on line 27 without verifying that it is a valid UTF-8 string, allowing an attacker to cause a denial of service at least, and possibly even execute arbitrary code.

Extrapolation. Applying our method to obtain the 30 most similar functions, we obtain the ranking shown in Table 4.3. Of the 30 functions, 28 are candidate functions, selected from a total of 11,505 functions. Moreover, nine of the functions contain are instances of the same vulnerable programming practice as found in the original vulnerability. As an example, Figure 4.7 shows the function `parseicon`, which reads a string from a network packet on line 16 and passes it to a packet handler on line 25 without verifying whether it is a valid UTF-8 string. It has been verified that this flaw can indeed be triggered to cause a denial of service.

The second case study shows that extrapolation is possible even when the fraction of candidate functions is small compared to the total number of functions. Moreover, it demonstrates that the method is not limited to finding vulnerabilities in media decoding libraries. However, it is worth pointing out that the vulnerabilities considered in the case studies are both taint-style vulnerabilities.

Sim.	Function name
1.00	receiveauthgrant
1.00	receiveauthreply
1.00	parsepopup
1.00	parseicon
1.00	generror
0.99	incoming_..._buddylist
0.99	motd
0.99	receiveadded
0.99	mtn_receive
0.99	msgack
0.99	keyparse
0.99	hostversions
0.98	userlistchange
0.98	migrate
0.98	error

Sim.	Function name
0.98	incomingim_ch4
0.98	parse_flap_ch4
0.98	infoupdate
0.98	parserights
0.98	incomingim
0.98	parseadd
0.97	userinfo
0.97	parsemod
0.97	parsedata
0.97	rights
0.97	rights
0.97	uploadack
0.96	incomingim_ch2_sendfile
0.96	rights
0.96	parseinfo_create

TABLE 4.3: Top 30 most similar functions to a known vulnerability in Pidgin [164].

4.6 Related Work

The method presented in this chapter provides analysts with a tool to accelerate manual vulnerability discovery. To this end, we extend work on the extrapolation of vulnerabilities, a problem, which is itself closely related to code clone detection. In the following, we thus discuss related work in these three areas in detail.

Extrapolation of vulnerabilities. The notion of vulnerability extrapolation was first introduced in prior work primarily conducted by the author of this thesis [163]. The work describes the simplest of the embeddings considered in this work, the API node embedding, and provides first empirical evidence for the success of vulnerability extrapolation in real world code auditing. The work presented in this chapter significantly extends prior work by showing how structural and syntactical information provided by the abstract syntax tree can be incorporated into the analysis. As a result, we are able to model programming patterns beyond API symbols, represented by arbitrary patterns in the abstract syntax tree. In particular, this enables us to compare different embeddings in a controlled experiment, showing that embeddings focusing on API symbols are indeed superior to those which do not. Moreover, we demonstrate the merits of vulnerability extrapolation on a significantly larger body of source code. Finally, Pewny et al. [110] extend our work on vulnerability extrapolation for the analysis of binaries, and even provide an approach that allows semantically similar code to be identified across architectures [109].

Code clone detection. A related strain of research deals with *code clone detection* [see 14], that is, the discovery of code that has been copied and possibly slightly modified. While code clones share the same programming patterns, not all functions sharing the same programming patterns are also code clones. We can thus consider code clone detection to deal with a special case of the problem considered by our work.

Early approaches to code clone detection employ simple numerical features. For example, Kontogiannis et al. [74] use features such as the number of called functions and the McCabe cyclomatic complexity [95] to identify code clones. Baxter et al. [13] provide a more fine-grained approach based on comparing abstract syntax trees, and thus closely related to our work from a methodological point of view. Kamiya et al. [69] strike a balance between these two approaches by presenting a token-based method for code clone detection. Moreover, both Li et al. [84] and Jang et al. [65] have presented methods to scan entire operating system distributions for code clones, uncovering many bugs in the process.

From an algorithmic point of view, the work by Marcus and Maletic [93] on the discovery of high-level concept clones most closely resembles ours as it also employs latent semantic analysis. However, in stark contrast to our work, their representation of code is based on comments and identifiers while API symbols are discarded. This highlights a key difference in the requirements for code clone detection when compared to vulnerability extrapolation: while for copied code, it can be assumed that comments are preserved, this is not true for code that merely shares programming patterns with the original code.

Manual code review. The literature on strategies for manual discovery of vulnerabilities is rather scarce. Dowd et al. [35] provides a comprehensive overview of strategies for manual vulnerability discovery. Similarly, Koziol et al. [76] discuss vulnerability discovery with a focus on bug classes specific for system code, while Stuttard and Pinto [138] focus on the discovery of vulnerabilities in Web applications. In the same series of books, Miller et al. [97] discuss vulnerability discovery on the Apple iOS platform, while Drake et al. [36] deal with vulnerabilities on Android systems. Finally, Klein [72] provides an introduction to the topic based on a series of case studies with real world applications. Our work is related in that the method we propose implicitly suggests an auditing strategy, namely, the reviewing of known vulnerabilities and subsequent identification of similar code fragments likely to be vulnerable as well.

5

Discovering Vulnerabilities using Anomaly Detection

The previous chapter shows how dimensionality reduction can be leveraged to identify similar code. A situation is addressed where an analyst has access to a known flaw and is interesting in retrieving more functions sharing its programming patterns. The success of this approach hinges on the assumption that reoccurring programming patterns exist in the code base, and enables the analyst to spot these repetitions.

The method presented in the previous chapter is successful if the programming pattern itself constitutes a vulnerability. If this is the case, then by finding additional instances of the pattern, we find additional vulnerabilities. However, there are also vulnerabilities where the programming pattern is used correctly in many places and incorrectly only in a few. This is particularly often the case for mature and well-audited code where vulnerabilities are an exception rather than the rule. In these cases, the method presented in the previous chapter allows to focus on instances of the patterns, but offers no assistance in determining instances more suspicious than others. Particularly for large code bases where the sheer mass of code prohibits examining all functions employing the same programming pattern, we are interesting in further narrowing in on vulnerabilities. In theory, exact methods such as constraint solving, model checking, or symbolic execution can be used for this purpose, however, the difficulties in adapting these techniques to the specifics of a code base and vulnerability type, as well as the required computational effort, encourage to explore alternatives.

In this chapter, we examine how the method presented in the previous chapter can be extended and adapted to narrow in on vulnerable code by pointing out particularly pronounced deviations from programming patterns. Based on the assumption that, for mature software projects, programming patterns are used correctly in the majority of cases, we extend our analysis by an *anomaly detection* step. The intuition we follow is that if a check occurs in 9 out of 10 functions in a program, the single function missing the check is an anomaly that deserves the analyst's attention. With this idea in mind, we select functions that are similar with respect to the programming patterns they employ but different in the constraints they impose on user input. We focus on missing or unusual constraints ("checks"), as failure to correctly restrict user input is a common source of vulnerabilities (see Section 5.1).

As a result, we obtain a method for *finding missing checks or anomalous checks based on an attacker-controlled source or sensitive sink*. To this end, our method first identifies

functions employing similar programming patterns by employing the method presented in the previous chapter, and subsequently analyzes the data flow in these functions. Finally, anomaly detection is employed to highlight anomalous or missing checks.

In the following, we begin by discussing the setting in which an immediate demand for missing check detection arises and the information the analyst needs to supply to operate our method in Section 5.1. We proceed to provide a brief discussion of anomaly detection in Section 5.2, the unsupervised machine learning technique our approach is based on. The resulting approach for missing check detection is subsequently presented in Section 5.3 along with an empirical evaluation in Section 5.4. We conclude by discussing related work in Section 5.5.

5.1 Task: Missing Check Detection

Many vulnerabilities can be directly linked to the omission of security critical checks as suggested by the results discussed thus far (Section 4.5) and strikingly illustrated by severe vulnerabilities discovered in the past years. For example, a missing check in the access control logic of the Java 7 runtime discovered in January 2013 enabled attackers to install malware on millions of hosts via drive-by-download attacks (see CVE-2013-0422).

```
1  int foo(char *user, char *str, size_t n)
2  {
3    char buf[BUF_SIZE], *ar;
4    size_t len = strlen(str);
5
6    if(!is_privileged(user))
7      return ERROR;
8
9    if(len >= BUF_SIZE) return ERROR;
10   memcpy(buf, str, len);
11
12   ar = malloc(n);
13   if(!ar) return ERROR;
14
15   return process(ar, buf, len);
16 }
```

FIGURE 5.1: Exemplary security checks in a C function: a check implementing security logic (line 6) and two checks ensuring secure API usage (line 9 and 13) [166].

As examples for security critical checks, consider the three checks in the listing shown in Figure 5.1, each of which fall into one of the following two categories.

- **Checks ensuring secure API usage.** Securely operating functions provided by application programming interfaces often requires pre- or post-conditions to be met. For example, when calling the function `malloc`, the return value should always be checked to be non-null to catch cases where allocation fails. The listing shown in Figure 5.1 contains two of these types of checks, namely the check for the return value of `malloc` (line 13) and the check on the amount of data to copy (line 9) via `memcpy`. In both cases, performing the check amounts to good programming practices but becomes crucial when the processed value is attacker-controlled.
- **Checks implementing security logic.** Other checks ensure that access control is implemented correctly, allowing security critical operations to be executed only

when certain conditions are met. These types of checks are commonly found in Web applications but also in system software such as operating system kernels. In our sample listing shown in Figure 5.1, the first of the three checks (line 6) is of this type. It ensures that the operation offered by the function `foo` can only be carried out if the user is privileged to do so.

This leads us to the following definition for missing check vulnerabilities that captures both checks related to secure API usage, as well as checks that implement security logic.

Definition 5.1. A *missing-check vulnerability* is a vulnerability caused by a failure to restrict data that (a) originates from an attacker-controlled source, or (b) reaches a security sensitive operation.

In this chapter, we consider the task of exposing these types of missing checks to the analyst by only making use of the available code. Hence, we consider an unsupervised setting where no examples of vulnerable and non-vulnerable code are available, nor are annotations accessible as for example used by the security checker *Splint* [40]. Instead, we make use of the fact that sources and sinks are typically executed several times throughout a code base. Assuming that they are operated correctly in most cases, we can employ anomaly detection to determine incorrect usage automatically. In this context, we consider the following two settings encountered by analysts in practice.

- **Initial code browsing.** In the first setting, the analyst sees the code for the first time and is not acquainted with the application programming interfaces it makes use of. As she browses a function, we want to highlight invocations of APIs that are missing their respective typical checks. This allows the analyst to spot potential vulnerabilities despite lack of existing knowledge about correct API usage.
- **Scanning based on known source or sink.** The second setting considers the situation where the analyst knows of a source she controls or an operation that requires protection, and she would like to scan the code for instances where these sources or sinks are used without their respective typical checks. This setting occurs throughout the auditing process as the analyst understands the attack surface.

The primitive to implement in both settings is that of identifying for a given function and a *symbol of interest* (e.g., a source or sink) whether it is invoked with the typical check pattern in place or not. However, before we discuss how this primitive can be implemented in detail using a suitable method, we now briefly discuss *anomaly detection*, our method's core ingredient.

5.2 Anomaly Detection

The main building block for our approach to missing check detection is anomaly detection, a family of unsupervised machine learning algorithms that deal with the identification of anomalous data points. To this end, a model of normality is inferred from the data, making it possible to compare data points to this model to uncover anomalies.

This comparison is implemented by a prediction function $f_\theta : \mathcal{X} \to \mathbb{R}$ that maps each $x \in \mathcal{X}$ to a numerical value referred to as an anomaly score $y \in \mathbb{R}$ (see Section 1.2).

In the machine learning literature, anomaly detection methods are commonly classified into global and local methods, where global methods calculate models of normality over the entire data set, while local methods consider only data points nearby in the feature space [see 54]. Chandola et al. [21, Sec. 2.2] additionally introduce the concept of *contextual anomalies*, a notion they see implemented particularly as researchers apply anomaly detection to time series. They define contextual anomalies to be data points that can be considered anomalous within a context, but not otherwise. What is interesting about this idea is that, in contrast to local anomalies, data points in the same context need not be nearby in feature space, and can instead be established via properties unencoded in the vectorial representation. For example, in code analysis, we may consider only functions written by the same author to be in the same context, or only functions written within a specific time frame.

One of the simplest approaches to global anomaly detection is to choose the data's *center of mass* as a model [see 116]. For a data set $X \subset \mathbb{R}^n$ of n-dimensional vectors, the model is then simply given by the vector

$$\mu = \frac{1}{|X|} \sum_{x \in X} x.$$

where $|X|$ denotes the number of elements in X. The anomaly score for an $x_0 \in X$ is then calculated as

$$f(x_0) = ||\mu - x_0||$$

where $\| \cdot \|$ is a suitable metric, that is, we simply calculate a distance from x_0 to the center of mass. This simple approach can be easily extended to determine contextual anomalies, as we make use of in our approach for missing check detection. To achieve this, we simply restrict the set X to those data points in the same context as x_0 when calculating its anomaly score.

5.3 Discovering Missing Checks

To identify missing checks in source code, we combine the method for finding similar functions presented in the previous chapter with anomaly detection based on center of mass as outlined in the previous section. Conceptually, our method proceeds in two consecutive steps. First, we discover *neighborhoods* of similar functions based on API symbols. Second, we determine anomalies in the way user input is checked for each of these neighborhoods separately. We implement this procedure in the following five steps that can be executed for any given function of interest (see Figure 5.2).

- **Source and sink identification.** For the function of interest, we begin by determining all sources and sinks. For each of these, we also determine all other functions employing the same source or sink (Section 5.3.1).
- **Neighborhood Discovery.** Whether a check is required may depend on the context it is used in. We therefore employ the machinery introduced in Chapter 4 to first identify functions operating in a similar context (Section 5.3.2).

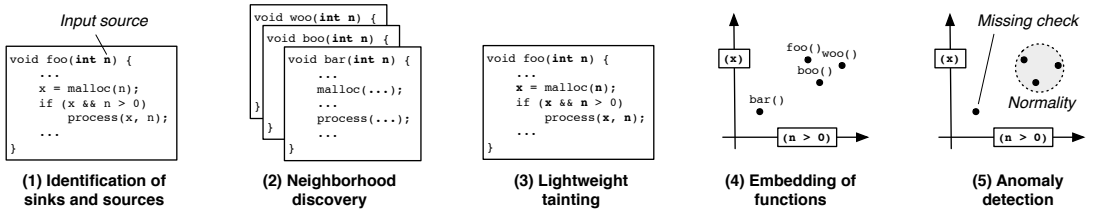


FIGURE 5.2: Overview of our method for missing check detection: (1) sources and sinks are identified, (2) functions with similar context are grouped, (3) variables depending on the sources/sinks are tainted, (4) functions are embedded in a vector space using tainted conditions, and (5) functions with anomalous or missing conditions are detected [165].

- **Lightweight Tainting.** We proceed to identify the sub set of checks that are relevant for the given source or sink by following data flow edges in the program dependence graph (Section 5.3.3).
- **Embedding of functions.** Functions of the neighborhood are subsequently represented as vectors encoding the checks they perform, allowing for subsequent analyze using machine learning methods (Section 5.3.4).
- **Anomaly Detection.** Finally, we compute a model of normality over the functions of the neighborhood and perform anomaly detection to determine those functions where a check is likely to be missing (Section 5.3.5).

We now describe each of these steps in greater detail and highlight connections to the concepts and methods presented in the previous chapters.

5.3.1 Source and sink identification

The method presented in this section is entirely based on syntax trees, and hence, can be implemented on top of our platform for robust code analysis. In particular, we extract the following information from the database for each of function of the code base.

- **Sources and sinks.** We extract all function parameters, function calls, as well as global and local variables and consider them potential *sources* and *sinks* of information. Each of these symbols may be tied to their own unique set of conditions that must hold in order for it to be operated securely. We further increase the granularity of the analysis by considering fields of structures as separate symbols.
- **API symbols.** As a prerequisite for neighborhood discovery, we additionally extract all nodes for types used in parameter and local variable declarations as well as all callees. We refer to these as API symbols in correspondence with the API nodes used in Section 4.4.2.
- **Assignments.** To develop a lightweight form of taint analysis that can be performed on the syntax tree alone (see Section 5.3.3), we additionally extract assignment nodes from functions, as they describe the flow of information between variables.

- **Conditions.** Finally, condition nodes are extracted as we ultimately seek to compare functions in terms of the conditions they impose on input originating in or propagating two a source or sink respectively.

This exhaustively enumerates the information required to carry out our method for missing check detection, which we proceed to describe in the following sections.

5.3.2 Neighborhood Discovery

Whether a missing check results in a vulnerability is dependent on the context code operates in. For example, failure to check the sizes of strings when processing configuration files may be acceptable, while omitting the same check when parsing attacker-controlled network packets can have devastating effects. To identify relevant anomalies in the way a function checks its input, the function should only be compared to other functions operating in a similar context, as opposed to all functions of the code base. To this end, our method begins by identifying a *neighborhood* of functions for the function of interest.

We identify a function's neighborhood by employing the method for similar function identification presented in the previous chapter, and in particular, API nodes embedding (see Section 4.4.2). The rational behind this choice is that the combination of interfaces used by a function is characteristic for the subsystem it operates in and the tasks it fulfills.

In conformance with the notation in Section 4.4.2, we denote the set of functions by $\mathcal{X}$ and the set of all API symbols in the code base by L. Then, following our method for similar function detection, we can map functions to a vector space using the function

$$\phi : \mathcal{X} \longmapsto \mathbb{R}^{|L|}, \quad \phi(x) = \big(\#_w(x) \cdot v_w\big)_{w \in L}$$

where, again, $\#_w(x)$ denotes the number of API nodes in the function x that are mapped to the API symbol s, and v_w corresponds to a TF-IDF weighting term[1].

We can now identify the k most similar functions employing the symbol of interest geometrically $\mathcal{N} \subset \mathcal{X}$, where, sticking to the method presented earlier, the cosine distance is used as a measure of dissimilarity. As we show in the empirical evaluation (Section 5.4.1, our method is not very sensitive to the choice of k, and values between 10 and 30 provide good performance.

5.3.3 Lightweight Tainting

Functions can contain many checks, only few of which are related to a particular source or sink of interest. Focusing our analysis on a specific source or sink therefore requires a procedure to discard unrelated checks.

To address this problem, we perform *lightweight tainting* for a given function and symbol of interest in the following two stages.

[1] Optionally, a truncated singular value decomposition can be performed to reduce the dimensionality of the feature space, however, a study succeeding our work shows that this has little effect on the results of this particular method [89].

- **Dependency modeling.** We model the dependencies between variables in a dependency graph, a directed graph where the nodes represent the function's identifiers, that is, possible sources and sinks, and the edges correspond to assignments between identifiers. We additionally add edges from arguments to the functions which receive them.
- **Taint propagation.** To discover identifiers related to a source or sink, we start at the symbol of interest and traverse the graph in a top-down as well as bottom-up direction. Propagation terminates at function boundaries, that is, we do not expand edges from arguments to functions.

As an example, Figure 5.3 shows the dependency graph for the function `foo` from Figure 5.1 and the symbol `memcpy`. Taint propagation marks three identifiers tainted as depicted by gray shading, two of which are directly connected nodes (`len` and `buf`), and one of which is only indirectly connected (`strlen`).

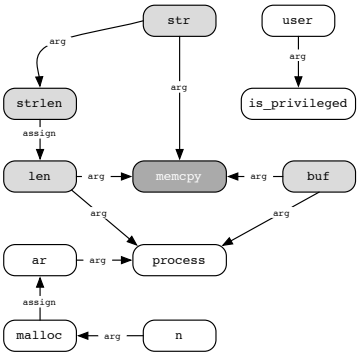


FIGURE 5.3: Dependency graph for function `foo`. Nodes reachable from `memcpy` are shaded, where the taint propagation stops at function boundaries. Isolated nodes are omitted [166].

Upon determining tainted identifiers in the dependency graph, we finally select relevant conditions by choosing only those conditions that reference at least one of the tainted identifiers. For a function represented by its function node x, we refer to the corresponding set of condition nodes as C_x.

Another possibility to identify only those checks related to the source or sink is to perform program slicing based on program dependence graphs as explored in the context of our method by Maier [89]. This is a useful practical addition to our original technique, in particular because it allows a symbol to be analyzed as a source or sink only, as opposed to a possible source and sink. However, this is shown by Maier [89] to perform very similar to the method based on lightweight tainting in the controlled setting (see Section 5.4.1).

5.3.4 Embedding of Functions

Upon execution of the previous two steps, a neighborhood for the function of interest as well as conditions relevant in these functions for the symbol of interest are available. We can now leverage this information to identify deviations from typical checks via anomaly detection techniques. To this end, we map functions of the neighborhood to a vector

space once more, however, we now represent them by the checks they contain as opposed to their usage of API symbols. We implement this embedding via the following two step procedure.

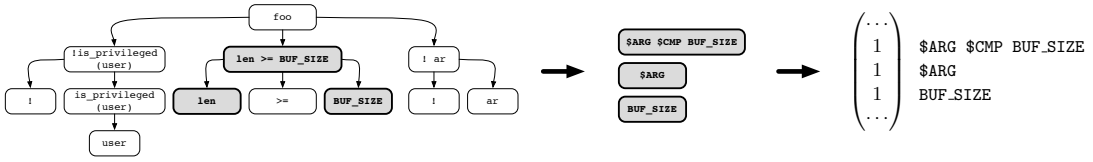


FIGURE 5.4: Schematic depiction of embedding. Conditions related to the specified source/sink are extracted from the abstract syntax tree and mapped to a vector space using their expressions [166].

Normalization. As our method does not evaluate expressions to identify whether they need to be true or false in order to be relevant for execution of a sink or termination after reading from a source, we perform a number of normalizations to account for variations in check formulation we do not recognize as such. We begin by removing negations and replacing relational and equality operators by the symbol `$CMP`. We additionally replace numbers by `$NUM` as, for example, a function checking whether a symbol x is smaller than 9 while all functions in the neighborhood check whether x is smaller than 10 should not be punished for being more restrictive. Similarly, to allow arguments and return values of the source or sink of interest to be compared irrespective of their specific names, we replace return values with the symbol `$RET` and arguments with the symbol `$ARG`.

Bag-of-subtrees. We proceed to extract all expressions from normalized conditions, that is, all sub trees they contain. Denoting the set of all of these expressions in all functions of the neighborhood as $\mathcal{T}$, we can again implement a tree-based feature map (see Section 3.4.3) to represent functions by their expressions. This map is given by

$$\varphi : \mathcal{X} \to \mathbb{R}^{|\mathcal{L}|}, \quad \varphi(x) = \big(I_w(x)\big)_{w \in \mathcal{L}}$$

where $\mathcal{L}$ is the set of expressions in textual representation, and I_w is simply an indicator function given by

$$I_w(x) = \begin{cases} 1 & \text{if } x \text{ contains an expression } e \text{ with } s(e) = w \\ 0 & \text{otherwise.} \end{cases}$$

where $w \in L$ is a string, and $s : \mathcal{T} \to \mathcal{L}$ maps sub trees to their textual representations.

This procedure is illustrated in Figure 5.4 for the example function `foo` from Figure 5.1, where the single condition relevant for the sink `memcpy` is first determined and its sub-expressions are subsequently normalized and extracted. Finally, we obtain the bag-of-subtrees representation of `foo` by applying the mapping φ.

As is true for all feature maps presented thus far, we can implement φ using the general embedding procedure presented in Section 3.5 in the following two steps.

Object extraction. As an input space, we consider the set of all functions. However, in contrast to the feature maps presented thus far, these are not represented by their

complete syntax trees, but only by sub trees rooted in condition nodes and relevant for the symbol of interest. We implement object extraction using the seed node selector

$$S(X) = \{C_x : x \in \textsc{Lookup}(\mathit{type}, \mathit{function}, X)\},$$

that is, for each function node, we determine the set of relevant condition nodes. We proceed to extract full sub trees starting in these condition nodes using the function $\textsc{Object}_{\mathcal{E}}^{\mathcal{D}}$ with $E := \textsc{OutE}_{\mathcal{A}}$ and $\mathcal{D} := \infty$, the same function already employed in Section 4.4.1 for the extraction of the complete syntax tree from the root node.

Sub structure enumeration. To enumerate sub structures, we use the attribute *code* of each node as its label. Following the generic embedding procedure, we simply define the labeling function to be $l(x) = \mu(x, \mathit{code})$ to achieve this. The set of substructures T_x for the object x corresponds to the nodes of the syntax trees of its relevant conditions, that is, the set C_x. The set of all substructures $\mathcal{T}$ is then given by the union of all sets T_x over all objects. Following the embedding procedure, we can simply instantiate the function $\textsc{Subgraph}_{\mathcal{E}'}^{d}$ with $\mathcal{E}' = \mathcal{E}$, and $d = 1$, meaning that for each node of the object, we return the node itself.

Mapping objects to vectors. Finally, we need to define an embedding language $\mathcal{L}$ and a function $s : \mathcal{T} \to \mathcal{L}$ from substructures to words of the embedding language to fully define the parameters of the embedding procedure. We define the embedding language $\mathcal{L}$ to be given by the set of labels assigned to nodes, and the function s as a function that maps each node to its label.

5.3.5 Anomaly Detection

With vectorial representations of all neighboring functions at hand, we can now identify missing and anomalous checks in the function of interest automatically. We achieve this by calculating a model of normality, which encodes those expressions common among functions of the neighborhood, and determining the deviation of the target function from this model. In particular, this allows us to identify checks present in the majority of functions of the neighborhood that are missing in the target function. In addition, we obtain an anomaly score that allows functions to be ranked in terms of the available evidence for a missing check. Conceptually, we thus expose contextual anomalies, as the checks absent from a function are only flagged as anomalies in comparison to functions operating in a similar context.

For a function x and a symbol of interest, we implement this step based on its neighborhood $\mathcal{N}$ and the mapping φ that represents functions in terms of the expressions in its conditions, however, limited to those expressions relevant for the symbol of interest. As a model of normality, we choose the center of mass of all functions in the neighborhood, given by

$$\mu = \frac{1}{|\mathcal{N}|} \sum_{n \in \mathcal{N}} \varphi(n).$$

As each of the dimensions of the feature space are associated with an expression, each coordinate of μ encodes the fraction of neighborhood functions that contain a particular expression, where in the case of a value of one, all functions contain the expression, while in the case of zero, none do.

To calculate missing checks, we can now simply determine the distance vector $d \in \mathbb{R}^{|E|}$ given by $d = \mu - \varphi(x)$, where each coordinate is a value between -1 and $+1$. As we subtract the function of interest from the model of normality, a positive number denotes a *missing check*, that is, an expression that is checked by a fraction of the neighborhood but not in the function of interest. In contrast, a negative value indicates an expression checked in the function but not checked by any of its neighbors.

While the coefficients of d already allow us to point out the exact missing expression to the analyst, we can additionally calculate an anomaly score to rank functions according to their deviation from normality. To this end, we define the anomaly score as

$$f(x) = ||\mu - \varphi(x)||_\infty = \max_{e \in E} \big(\mu_e - I(x, e)\big).$$

The anomaly score is thus given by the largest coordinate of d, that is, the maximum deviation observed from any of the expressions used in the neighborhood. We choose the maximum norm as functions deviating in a single expressions while containing all other checks typical for the neighborhood are usually at least as interesting as those differing in their conditions entirely.

5.4 Evaluation

Following the same methodology as for the evaluation of our method for vulnerability extrapolation, our evaluation is two-fold. First, we perform a quantitative evaluation in a controlled setting to measure the method's detection performance (Section 5.4.1), and second, we put our method to work in a real world audit to explore its practical value in a series of case studies (Section 5.4.2).

5.4.1 Controlled Experiment

We perform a controlled experiment on the source code of five popular open-source projects, namely Firefox, Linux, LibPNG, LibTIFF, and Pidgin. To this end, we review the security history of each of these projects to uncover cases where a security-relevant check is present in several functions but missing in a few, thus causing vulnerabilities. In all but one case, the vulnerabilities chosen are critical, allowing for full system compromise. Moreover, we make sure to choose different types of missing security checks, that is, checks involving security logic, function return values, and function arguments. Table 5.1 summarizes our data set. The following description of this data set is taken verbatim from [166].

Project	Component	Vulnerability	LOC	# functions	# with check
Firefox 4.0	JavaScript engine	CVE-2010-3183	372450	5649	10
Linux 2.6.34.13	Filesystem code	CVE-2010-2071	955943	19178	8
LibPNG 1.2.44	Entire library	CVE-2011-2692	40255	473	19
LibTIFF 3.9.4	Entire library	CVE-2010-2067	33335	609	9
Pidgin 2.7.3	Messaging	CVE-2010-3711	332762	7390	18

TABLE 5.1: Overview of our dataset. For each project the missing-check vulnerability, the lines of code (LOC), the number of functions and the number of functions involving the check is listed [166].

- **Firefox.** The JavaScript engine of the popular Web browser Firefox (version 4.0) contains 5,649 functions and 372,450 lines of code. A failure to check the number of arguments passed to native code implementations of JavaScript functions (i.e., the parameter `argc`) leads to a use-after-free vulnerability (CVE-2010-3183). Ten utility functions implementing array operations perform the same security check to avoid this.

- **Linux.** The filesystem code of the Linux operating system kernel (version 2.6.34.13) contains 19,178 functions and 955,943 lines of code. A missing check before setting an ACL allows to bypass file system permissions (CVE-2010-2071). The check involves the parameter `dentry` and its structure field `dentry->d_inode`. Eight functions of different filesystems implement a corresponding security check correctly.

- **LibPNG.** The image processing library LibPNG (version 1.2.44) contains 437 functions and 40,255 lines of code. A missing check of the PNG chunk's size (i.e., the parameter `length`) results in a memory corruption (CVE-2011-2692). Nineteen functions processing PNG chunks perform the same critical check to avoid this.

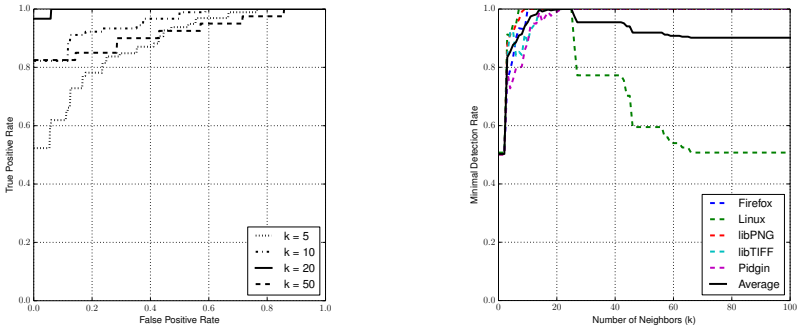
- **LibTIFF.** The image processing library LibTIFF (version 3.9.4) contains 609 functions and 33,335 lines of code. Missing checks of the length field of TIFF directory entries (i.e., the parameter `dir` and its field `dir->tdir_count`) lead to two independent stack-based buffer-overflows (CVE-2006-3459 and CVE-2010-2067). Nine functions processing TIFF directory entries perform a security check to avoid this problem.

- **Pidgin.** The instant messaging library of the popular instant messenger Pidgin (version 2.7.3) contains 7,390 functions and 332,762 lines of code. A missing check of the return value of the internal base64-decoding routine `purple_base64_decode` leads to a denial-of-service vulnerability (CVE-2010-3711). Eighteen functions parsing network data in Pidgin perform a corresponding security check correctly to avoid this.

This data set comprises known vulnerabilities caused by missing checks, but also non-vulnerable functions, which contain a corresponding check. This allows us to assess whether our method is capable of correctly distinguish the vulnerable from the non-vulnerable functions. However, we can also go a step further: by patching the known vulnerabilities and removing them in a previously non-vulnerable function, we can evaluate our method's ability to find the missing check, had it been contained in one of the other functions. In sum, this allows us to perform 64 different experiments (see the last column of Table 5.1) where one function omits the necessary check while the remaining examples do not.

For several known non-vulnerable and one vulnerable function, we execute our method to rank these functions according to their usage of the source or sink requiring validation. While we rank only those functions known to be vulnerable on non-vulnerable, it is noteworthy that the method considers all functions of the code base for neighborhood selection. In effect, the experiment resembles the situation encountered in practice where a source or sink of interest is given and the analyst scans the source code for anomalies (see Section 5.1).

Figure 5.5a shows the *Receiver Operation Characteristics* (ROC) for our method averaged over all projects and for different values of k , that is, the detection rate as a



(A) Averaged ROC curve (B) AUC (Minimal Detection Rate)

FIGURE 5.5: Detection performance for the five projects with neighborhoods of different size [166].

function of the false positive rate. In particular, we see that for $k = 5$, we can already identify 50% of the vulnerabilities with few false positives. As k is increases, a growing number of missing checks is found at zero false positives, until finally, at $k = 20$, 96% of the missing checks are identified.

We explore the effect of the parameter k in greater detail by plotting the area under the ROC curve (AUC) as a function of k for the different code bases (Figure 5.5b). For $k = 25$, we obtain perfect results across all code bases, showing that for the given data set, choosing around 25 neighbors to calculate the model of normality is a good choice for all code bases. While this does not allow us to conclude that choosing exactly 25 neighbors is optimal for arbitrary code bases, it shows that a good value can be found across code bases, and thus, we fix k to 25 in the following practical experiments.

Finally, for we can see that for the cases where an API is used insecurely (Firefox, LibPNG, LibTIFF, and Pidgin), the maximum performance is reached once k passes a threshold. In this case, neighborhood discovery becomes dispensable as the check seems to be performed by functions using the symbol regardless of context.

In contrast, for missing checks implementing security logic in the Linux file system code, the performance reaches a peak at around 20 neighbors and subsequently declines. We find that the symbol `dentry` is used in many different context and a check is required only in a few, making neighborhood discovery essential for the success of our method.

5.4.2 Case Studies

In the following, we proceed to evaluate the practical merits of our method in a series of case studies on the source code of the image processing library LibTIFF and the instant messenger Pidgin, where our method plays a vital role in uncovering 7 previously unknown vulnerabilities. While we found another 5 vulnerabilities during our evaluation, we omit their discussion for brevity.

5.4.2.1 Case Study: LibTIFF

Libraries processing multimedia content are a common target for attack as the corresponding file formats can be hard to parse securely (see Section 4.5.2.1). Moreover,

they are an attractive target as vulnerabilities in libraries affect callers of the vulnerable code, thereby making them relevant for a wider range of applications. In this first case study, we focus on LibTIFF, a library and suite of tools for the Tagged Image File Format (TIFF). This library is well known among vulnerability researchers for an infamous buffer overflow on the stack that enabled attackers to run third party code on early versions of Apple's iPhone [see 28].

Failure account for the limited range of integers in arithmetic operations is a particularly common source common source for vulnerabilities in the context of image processing. In particular, integer overflows when multiplying image dimensions and bit depths are common, as prominent examples in libpng (CAN-2004-0599), the BMP decoder of Firefox (CVE-2004-0904), the corresponding decoder in Microsoft GDI+ (CVE-2008-3015), and the vector graphic library Cairo (CVE-2007-5503) show. With these flaws in mind, we use our method to rank all functions of the code base with respect to anomalous use of any parameters or local variables named `with`, `height`, `w`, or `h`.

Of the 74 functions dealing with these variables, only a single function with an anomaly score of 100% is reported, namely, the function `tiffcvt`. Examining this function (Figure 5.6a), we find that the values for the variables `width` and `height` are (a) directly obtained from the image file on lines 11 and 12, and are thus attacker controlled, and (b) not checked before subsequently calling the function `cvt_by_tile`. In contrast, all neighbors of the function perform a check on the variable `height`, while 79% additionally check the variable `width`.

```
1
2
3  static int
4  tiffcvt(TIFF* in, TIFF* out)
5  {
6    uint32 width, height; /* image width & height */
7    uint16 shortv;
8    float floatv;
9    char *stringv;
10   uint32 longv;
11   uint16 v[1];
12
13   TIFFGetField(in, TIFFTAG_IMAGEWIDTH, &width);
14   TIFFGetField(in, TIFFTAG_IMAGELENGTH, &height);
15
16   CopyField(TIFFTAG_SUBFILETYPE, longv);
17   [...]
18   if( process_by_block && TIFFIsTiled( in ) )
19     return( cvt_by_tile( in, out ) );
20   else if( process_by_block )
21     return( cvt_by_strip( in, out ) );
22   else
23     return( cvt_whole_image( in, out ) );
24 }
25
26
27
```

(A)

```
1  static int
2  cvt_by_strip( TIFF *in, TIFF *out )
3
4  {
5    uint32* raster; /* retrieve RGBA image */
6    uint32  width, height; /* image width & height */
7    [...]
8    TIFFGetField(in, TIFFTAG_IMAGEWIDTH, &width);
9    TIFFGetField(in, TIFFTAG_IMAGELENGTH, &height);
10   /* Allocate strip buffer */
11   raster = (uint32*)
12    _TIFFmalloc(width*rowsperstrip*sizeof (uint32));
13   if (raster == 0) {
14     TIFFError(TIFFFileName(in),
15               "No space for raster buffer");
16     return (0);
17   } [...]
18   for(row=0;ok&&row<height;row+=rowsperstrip )
19     { [...]
20       /* Read the strip into an RGBA array */
21       if (!TIFFReadRGBAStrip(in,row,raster)) {
22         [...]
23       } [...]
24     }
25   _TIFFfree( raster ); [...]
26   return ok;
27 }
```

(B)

FIGURE 5.6: Missing checks of the variables width and height in the function `tiffcvt` (left). The resulting integer overflow in the function `cvt_by_strip` triggers a buffer overflow when calling `TIFFReadRGBAStrip` (right) [166].

The missing check indeed results in a vulnerability when calling the function `cvt_by_strip` on line 19, which itself is suggested to check the field `width` by 50% of its neighbors. The vulnerable function is shown in Figure 5.6b where, upon triggering the integer overflow, a buffer smaller than expected can be allocated on line 11, which is subsequently used as a target of a copy operation (`TIFFReadRGBAStrip`) on line 21, resulting in a heap-based buffer overflow.

As a second example, we employ our method to uncover NULL pointer dereferenciations. To this end, we rank all functions of the code base according to anomalous use of the symbol `_TIFFMalloc`, a simple wrapper for the function `malloc` from the C standard library `libc`.

Score	Source File	Function Name
0.92	tools/thumbnail.c	initScale
0.88,	tools/rgb2ycbcr.c	cvtRaster
0.88	tools/rgb2ycbcr.c	setupLuma
0.88	tools/ycbcr.c	setupLuma
0.84	tools/pal2rgb.c	main
0.84	tools/tiff2bw.c	main
0.80	libtiff/tif_print.c	TIFFPrintDirectory
0.80	tools/raw2tiff.c	guessSize
0.76	tools/sgisv.c	svRGBContig
0.76	tools/sgisv.c	svRGBSeparate

TABLE 5.2: Top ten functions returned for the sink `_TIFFmalloc`. All 10 functions fail to check the return value of the sink. Vulnerabilities are indicated by dark shading [166].

Overall, a total of 237 functions call `_TIFFMalloc`, where Table 5.2 shows the top ten functions of the resulting ranking. In each of the ten cases, our method reports a missing check for the return value, that is, the expression `$RET`, as confirmed in all ten cases. In four cases, the missing check allows attackers to cause a denial of service condition via specifically crafted input, while in the other six cases, only a software defect results.

```
1  cvtRaster(TIFF* tif, uint32* raster,
2            uint32 width, uint32 height)
3  {
4    uint32 y;
5    tstrip_t strip = 0;
6    tsize_t cc, acc;
7    unsigned char* buf;
8    uint32 rwidth = roundup(width, horizSubSampling);
9    uint32 rheight = roundup(height, vertSubSampling);
10   uint32 nrows = (rowsperstrip > rheight ?
11                   rheight : rowsperstrip);
12   uint32 rnrows = roundup(nrows,vertSubSampling);
13
14   cc = rnrows*rwidth + 2*((rnrows*rwidth) /
15        (horizSubSampling*vertSubSampling));
16   buf = (unsigned char*)_TIFFmalloc(cc);
17   // FIXME unchecked malloc
18   for (y = height; (int32) y > 0; y -= nrows){
19     uint32 nr = (y > nrows ? nrows : y);
20     cvtStrip(buf, raster + (y-1)*width, nr, width);
21     nr = roundup(nr, vertSubSampling);
22     acc = nr*rwidth + 2*((nr*rwidth)/
23      (horizSubSampling*vertSubSampling));
24     if(!TIFFWriteEncodedStrip(tif,strip++,
25                               buf,acc)){
26       _TIFFfree(buf); return (0);
27     }
28   }
29   _TIFFfree(buf); return (1);
30 }
```

FIGURE 5.7: A missing check detected in the function `cvtRaster` of the library LibTIFF [166].

The function `cvtRaster` (Figure 5.7) provides an illustrative example for a missing check of `_TIFFMalloc`'s return value identified by our method. In fact, a comment placed by the programmer on line `17` confirms that the return value of `_TIFFMalloc` requires validation. In this particular case, the check is suggested by 85% of the function's neighbors. Moreover, 40% suggest to compare against the constant `NULL`. Furthermore, it is noteworthy that no other Boolean expression is found to consistently occur across all neighbors in more than 30% of the cases, and that, compared to deviations from all other symbols, that in the coordinate of `_TIFFMalloc` is most pronounced. In fact, the function is among the top 15% anomalous functions in the global ranking, and thus, our

method directs the analyst towards this function even if an interest in `_TIFFMalloc` is not expressed.

5.4.2.2 Case Study: Pidgin

In our second case study, we use our method to analyze the source code of the popular instant messenger Pidgin, allowing us to uncover two denial-of-service vulnerabilities in its implementation of the Microsoft Instant Messaging Protocol. In particular, we identify a vulnerability that allows users to remotely crash Pidgin instances of other uses without requiring cooperation from the victim.

To bootstrap the analysis, we review the C standard library `libc` for functions that crash when a `NULL` pointer is passed to them as an argument. As an example, we choose the sink `atoi` and `strchr` and employ our method to rank all functions with respect to missing or anomalous checks associated with these sinks.

Score	Source File	Function Name
0.84	msn.c	msn_normalize
0.76	oim.c	msn_oim_report_to_user
0.72	oim.c	msn_parse_oim_xml
0.72	msnutils.c	msn_import_html
0.64	switchboard.c	msn_switchboard_add_user
0.64	slpcall.c	msn_slp_sip_recv
0.60	msnutils.c	msn_parse_socket
0.60	contact.c	msn_parse_addr..._contacts
0.60	contact.c	msn_parse_each_member
0.60	command.c	msn_command_from_string
0.56	msg.c	msn_message_parse_payload

TABLE 5.3: Top ten functions returned for the sinks `atoi` and `strchr` in Pidgin's implementation of the Microsoft Instant Messenger Protocol. Vulnerabilities are indicated by dark shading [166].

Table 5.3 shows the resulting ranking, where only functions with an anomaly score greater than 50% are preserved, that is, cases where more than half our the function's neighbors suggest a check to be introduced. In all of these cases, our methods suggest that the argument (`$ARG`) should be checked, allowing us to discover two cases among the top ten missing checks that allow an attacker to remotely crash the application.

First Example. For the function `msn_parse_oim_xml` shown in Figure 5.8a, our method reports that it does not validate an argument passed to `atoi` while 72% of its neighbors perform a check of this kind. Line 19 confirms this claim, showing a call to `atoi`, where its firth argument `unread` is unchecked. In addition, our method reports that 75% of the function's neighbors check the return value of calls to `xmlnode_get_data`. In combination, the two missing checks allow an attacker to crash Pidgin by sending an XML-message with an empty `E/UI` node, causing the call to `xml_node_get_data` to return `NULL` on line 13, eventually causing a crash on line 19.

Second Example. Our method reports a missing check for the argument passed to the sink `strchr` with an anomaly score of 56% for the function `msn_message_parse_payload` shown in Figure 5.8b. Line 15 shows the vulnerable call that can be triggered by sending a message containing the string `Content-Type` immediately followed by two successive

```
1  //
2  static void
3  msn_parse_oim_xml(MsnOim *oim, xmlnode *node)
4  {
5    xmlnode *mNode;
6    xmlnode *iu_node;
7    MsnSession *session = oim->session;
8    [...]
9    iu_node = xmlnode_get_child(node, "E/IU");
10
11   if(iu_node != NULL &&
12    purple_account_get_check_mail(session->account))
13     {
14       char *unread = xmlnode_get_data(iu_node);
15       const char *passports[2] =
16         { msn_user_get_passport(session->user) };
17       const char *urls[2] =
18         { session->passport_info.mail_url };
19
20       int count = atoi(unread);
21
22       /* XXX/khc: pretty sure this is wrong */
23       if (count > 0)
24         purple_notify_emails(session->account->gc,
25                              count, FALSE, NULL,
26                              NULL, passports,
27                              urls, NULL, NULL);
28       g_free(unread);
29     }
30   [...]
31 }
32 //
```

(A)

```
1  void
2  msn_message_parse_payload(MsnMessage *msg, [...])
3  {
4    [...]
5    for (cur = elems; *cur != NULL; cur++)
6      {
7        const char *key, *value; [...]
8        tokens = g_strsplit(*cur, ": ", 2);
9        key = tokens[0];
10       value = tokens[1];
11       [...]
12       if (!strcmp(key, "Content-Type"))
13         {
14           char *charset, *c;
15           if ((c = strchr(value, ';')) != NULL)
16             {
17               [...]
18             }
19           msn_message_set_content_type(msg, value);
20         }
21       else
22         {
23           msn_message_set_attr(msg, key, value);
24         }
25       g_strfreev(tokens);
26     }
27   g_strfreev(elems);
28   /* Proceed to the end of the "\r\n\r\n" */
29   tmp = end + strlen(body_dem);
30   [...]
31   g_free(tmp_base);
32 }
```

(B)

FIGURE 5.8: Missing checks in the functions `msn_parse_oim_xml` (left) and `msn_message_parse_payload` (right) of the instant messenger Pidgin [166].

carriage return line feed sequences. As a result, the variable `value` is set to `NULL` on line 10, a value, which is subsequently propagated to `strchr` on line called on line 15. It has been confirmed that this vulnerability allows users to crash Pidgin instances of over users.

5.5 Related Work

In the following, we provide an overview of prior work related to our method for missing check detection. In particular, related approaches to uncover missing checks via data mining techniques are outlined. Moreover, we discuss fuzz testing, a dynamic approach that can be used to uncover certain types of missing checks, as well as taint tracking, the technique that serves as the main inspiration for our lightweight tainting procedure.

Detecting missing checks using data mining techniques. While the merits of machine learning techniques for the discovery of defects and vulnerabilities have received little attention, several authors have explored the use of classical data mining techniques in this setting [e.g., 22, 48, 85, 87, 142, 169]. Among these, the work most closely related to the method presented in this Chapter is the method presented by Chang et al. [22] for determining missing checks using frequent item set mining. To this end, they construct program dependence graphs and identify their common graph minors as a model of normality. In particular, these graph minors contain conditions, allowing instances where these conditions are missing to be determined. The main weakness of their approach in comparison to ours is that similarities of conditions are not taken into account: while slight normalization is performed similar to that found in our method, conditions are subsequently treated as flat strings in the data mining framework. In

contrast, the tree-based embedding employed by our approach allows the similarity of conditions to be recognized via matching sub trees.

Tan et al. [142] present AutoISES, a method to automatically infer security specifications from code. Given a manually specified list of functions known to perform security critical checks, the method automatically infers the security critical operations they protect. Designed with kernel code in mind, these may be read and write operations on variables, including the fields of structures. Similarly, [156] provide a method to detect common sequences of method calls and their violations. The main limitation of these approaches is that checks are required to be calls to functions. In contrast, the checks derived by our method may be arbitrary conditions associated with the source or sinks of the code base. This is a prerequisite for the identifying of missing bounds checks, which are commonly associated with buffer overflows.

Finally, Thummalapenta and Xie [146] observe that many API functions are associated with multiple different patterns, resulting in high false positive rates for these functions as particular instances may conform only to some of the patterns, but not to all of them. As a remedy, they combine the different patterns of an API function into an *alternative pattern* that matches, whenever at least one of the original patterns matches. In addition, their approach differs from the data-mining based approaches discussed thus far as it can deal with checks unrelated to function calls. This is achieved by analyzing checks to partially recover their semantics. In contrast to the method presented in this chapter, this procedure is technically involved and language-dependent, and yet, it cannot account for check similarities.

Fuzz testing. Fuzz testing or *fuzzing* is a popular dynamic approach for vulnerability discovery, where the main idea is to provide possibly unexpected input to the target application and monitor it for crashes or other behavior beneficial for attackers. In particular, fuzzing is well suited to uncover missing checks leading to memory corruption vulnerabilities such as buffer overflows, as these are often made apparent by program crashes. A difference is typically made between black-box fuzzing on the one hand [e.g., 113, 139], where the target program is tested without access to its code, and whitebox fuzzing [e.g., 45, 52], where access to code is leveraged to guide fuzzing. In particular, Godefroid et al. [45] employ *symbolic execution* for whitebox fuzz testing. As implied by its name, the main idea of symbolic execution is to execute the target program with symbolic rather than concrete values, in order to explore a large number of possible executions at once. This technique can be employed to guide fuzzers by inverting constraints attached to paths of observed executions and employing a constraint solver [see 148] to determine corresponding inputs, allowing previously unseen code blocks to be reached. This procedure helps to account for one of the two main drawbacks of fuzzing, namely, that only paths observed at runtime can be analyzed for vulnerabilities, a deficit that is inherent to dynamic analysis. The second main drawback of these methods is that vulnerabilities can only be detected in this way if they either result in crashes or other program behavior that can be easily monitored.

Taint Tracking. A closely related strain of research considers *dynamic taint tracking* or *dynamic taint analysis*, where data controlled by attackers is tracked as it propagates through the system, and a vulnerability is reported if a sensitive sink is encountered without prior validation [e.g., 104, 155]. In contrast to fuzzing, this allows missing

checks to be detected even if they do not result in crashes. Finally, taint tracking can also be performed statically, a technique leveraged by researchers in the past to uncover specific types of vulnerabilities such as format string bugs [133], as well as SQL injection and cross site scripting vulnerabilities [68, 88]. Taint tracking offers a useful approach for monitoring data flows, however, it does not itself suggest which data flows to track or highlight in the first place. Our approach attempts to address this problem by suggesting particularly interesting data flows to the analyst.

6

Discovering Vulnerabilities using Clustering

This chapter presents the last and most evolved technique for pattern-based vulnerability discovery developed in this thesis. This final method combines ideas from all approaches presented thus far, and additionally explores the merits of clustering for vulnerability discovery, the last of the three themes of unsupervised learning to explore.

On the one hand, we saw in Chapter 2 that large amounts of code can be mined for vulnerabilities using manually crafted search patterns, and that we can even create templates for commonly reoccurring types of bugs. On the other, Chapters 4 and 5 show that we can automatically infer patterns from code using machine learning algorithms. In this chapter, we combine these two ideas by presenting a method to *automatically extract search patterns from source code*, that is, we derive a model of normality from the code similar to the method presented in the previous chapter, but additionally generate a query that can be used to mine for its deviations. The primary advantage this offers over the learning-based methods presented thus far is that an explicit, editable representation of the inferred patterns is created. This makes it possible to (a) expose and enumerate the patterns available in the code base in a form understandable by the analyst, and (b) create robust signatures for known vulnerabilities that allow for their extrapolation. Finally, the analyst thus gains the ability to tune the generated queries to introduce additional knowledge that cannot be derived from the statistics of the code base alone.

Clustering plays a key role in our approach as it allows us to create groups of statements similar enough to describe them using a common regular expression for all of the cluster's members. This idea has previously been successfully employed for signature generation in the context of malware detection [see 103], and thus our approach naturally extends this idea to vulnerability discovery.

The remainder of this chapter is structured as follows. We begin by introducing the task of learning search patterns for taint-style vulnerabilities in Section 6.1. In Section 6.2, we review *clustering*, the primary machine learning technique employed by our approach. We proceed to introduce our method for automatic extraction of search patterns in Section 6.3, and provide an empirical evaluation of our method in Section 6.4. We conclude by discussing related work in Section 6.5.

6.1 Task: Search Pattern Inference

Source code contains a wealth of patterns, few of which are linked to vulnerabilities. In effect, a common theme among the methods presented so far is that they rely on analysts to bootstrap the analysis by providing information on what they deem to be security relevant. Mining for vulnerabilities as presented in Chapter 2 is not possible without specifying a search pattern that describes a vulnerable programming practice, and extrapolating vulnerabilities is not possible without providing a known vulnerability first. Finally, missing checks can be discovered in abundance, but they only become security critical when they protect a security sensitive sink or restrict attacker-controlled data.

```
/* ssl/d1_both.c */
// [...]
int dtls1_process_heartbeat(SSL *s)
{
  unsigned char *p = &s->s3->rrec.data[0], *pl;
  unsigned short hbtype;
  unsigned int payload;
  unsigned int padding = 16; /* Use minimum padding */
  /* Read type and payload length first */
  hbtype = *p++;
  n2s(p, payload);
  if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec.4*/
  pl = p;
  // [...]
  if (hbtype == TLS1_HB_REQUEST){
   unsigned char *buffer, *bp;
   int r;
   // [...]
   buffer = OPENSSL_malloc(1 + 2 + payload + padding);
   bp = buffer;
   /* Enter response type, length and copy payload */
   *bp++ = TLS1_HB_RESPONSE;
   s2n(payload, bp);
   memcpy(bp, pl, payload);
   bp += payload;
   /* Random padding */
   RAND_pseudo_bytes(bp, padding);
   r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT,buffer,
                         3 + payload + padding);
   // [...]
   if (r < 0) return r;
  }
  // [...]
  return 0;
}
```

FIGURE 6.1: The "Heartbleed" vulnerability in OpenSSL [165].

What we are missing for now, is a method to assist an analyst in discovering *which patterns to look for in the first place*, while reducing the amount of information required to bootstrap the method to a bare minimum. In particular, we seek to enumerate only a small number of patterns in the code that are likely to be interesting for vulnerability discovery. For this enumeration to work, the method cannot be expected to know specifics of a code base. However, it can implement an abstract template for the type of vulnerable programming pattern it attempts to uncover and expose its instances. The research presented in this chapter shows that this is indeed possibly by implementing a method to uncover search patterns for *taint-style vulnerabilities*, an abstraction that captures different types of buffer overflows, buffer overreads, and injection vulnerabilities.

The *Heartbleed Bug* [29] in the cryptographic library OpenSSL is one of the most prominent recent examples of a taint-style vulnerability. Figure 6.1 shows the vulnerable code: on line 11, the macro `n2s` is employed to read a sixteen bit integer from a network stream and store it in the local variable `payload`. This integer is subsequently passed to the function `memcpy` as a third argument on line 25 without undergoing prior validation. In particular, it is not checked whether `payload` is smaller or equal to the size of the source buffer `pl`, allowing heap memory beyond this buffer's right boundary to be copied to the memory location pointed to by `bp`. The data thus stored is finally send out to the network on line 29 via a call to the function `dtls1_write_bytes`. With this example in mind, we define taint-style vulnerabilities as follows.

Definition 6.1. A *taint-style vulnerability* is a vulnerability caused by failure to restrict attacker-controlled data that reaches a security sensitive operation.

While this definition seems very similar to that of missing check vulnerabilities, and in fact, some taint-style vulnerabilities coincide with missing check vulnerabilities, their detection is considerably more evolved: while for missing check vulnerabilities, checks are tied to sources or sinks alone, and typically occur in their close proximity, checks in taint-style vulnerabilities are tied to the *combination* of a source and sink. This considerably complicates the detection of taint-style vulnerabilities as the corresponding source for a sink is often located several in a different function, making interprocedural analysis indispensable. However, by definition, there are also far fewer common combinations of sources and sinks in a code base than single sources and sinks, enabling us to considerably reduce the number of extracted patterns over the method described in the previous chapter. Once this method is in place, we can address the following two tasks in practice.

- **Enumeration of taint-style systems.** At the beginning of an audit, the analyst not only lacks knowledge about the APIs employed in a code base, but also does not know how API functions interact. We want to provide a method for the analyst to enumerate taint-style systems, i.e., *common pairs of sources and sinks along with their typical checks*. This allows an overview of the sources of data for sensitive operations to be gained, which quickly guides the analyst towards security-relevant taint-style systems. Moreover, if the system is inherently insecure, the corresponding search patterns can be directly employed to mine for vulnerabilities as we see in Section 6.4.2.2.
- **Signature generation for known bugs.** The generated search patterns can also be used to automatically characterize a taint-style vulnerability once it is known, in order to create a database of typical flaws to mine for in the future. This offers a form of vulnerability extrapolation (see Section 4.1) limited to taint-style vulnerabilities but also more powerful in this special case as it creates explicit representations of patterns.

These two tasks are addressed using the same procedure and merely differ in the input provided by the analyst. For enumeration, the analyst can optionally provide a sink of interest such as `memcpy`, while for signature generation, the location of a vulnerable call-site is specified. Before describing this procedure, we briefly review clustering, the key ingredient of our approach.

6.2 Cluster Analysis

The previous two chapters highlight the merits of unsupervised learning for vulnerability discovery, showing that both dimensionality reduction and anomaly detection can be employed to assist in the discovery of vulnerabilities, even in production-grade software. In this chapter, we show that *clustering* the third of the primary types of unsupervised algorithms (see Section 1.2) is equally useful in this setting. Clustering is employed to group objects according to their similarity. Given a finite set of objects $X \subseteq \mathcal{X}$ from an input space $\mathcal{X}$, along with a method to compare objects of the input space, clustering algorithms learn a model θ. This model can then be used to instantiate a prediction function $f_\theta : X \to I$ that associates each object with a cluster, denoted by an index from a set $I = \{1, \ldots, k\}$ of natural numbers.

There are many different clustering algorithms, which differ mainly in the types of clusters they produce, runtime and memory requirements, and the parameters they require. *Partitioning* and *hierarchical* algorithms are two of the most well-understood classical clustering techniques [see 64]. Partition-based algorithms such as *K-Means* and *K-Medoid* divide the input space into k partitions, where the number of partitions k must be specified in advance. This is achieved by iteratively refining the partitioning using schemes such as expectation maximization [see 16, Chp. 9] until the process converges. These algorithms are well suited for data where the number of clusters can be approximated beforehand, and difficult to employ if this is not the case.

Hierarchical algorithms offer an alternative to partitioning algorithms in situations where choosing the number of clusters in advance is difficult. These algorithms operate either by (a) first placing all objects into the same cluster and recursively splitting clusters until each object is located in a separate cluster, or (b) first placing each object into a separate cluster and joining clusters until all objects are in the same cluster. These strategies are referred to as *divisive* and *agglomerative clustering* respectively. Both approaches lead to the construction of a hierarchy of clusters, without requiring the number of clusters to be known in advance. However, the hierarchy by itself does not provide a clustering, that is, it does not unambiguously assign a cluster to each object. To obtain flat clusters from a hierarchy, we can for example choose a minimum number of clusters for a given maximum distance of objects inside clusters. Hierarchical clustering therefore does not free us from the burden of specifying the granularity of the clustering, but merely enables us to express it via different parameters.

The method for automated generation of search patterns presented in this chapter relies heavily on *linkage clustering*, a well-understood implementation of agglomerative clustering. The reason for this choice is that in the settings we consider, a maximum inter-cluster distance can be chosen intuitively, while this is not the case for the number of clusters. Linkage clustering as employed in our work requires two parameters to be specified: the maximum inter-cluster distance c, and a so-called linkage function $D : \mathcal{C} \times \mathcal{C} \to \mathbb{R}$ that enables measuring of distances between clusters.

With these parameters at hand, the method proceeds by placing each object into a unique cluster. Iterating over clusters, each cluster is subsequently linked with the most other cluster, according to the linkage function D. Linked clusters are then interpreted as new clusters themselves. In our work, we choose the linkage function to be given by

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

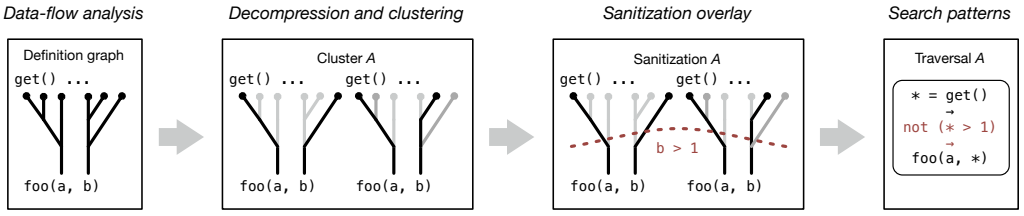


FIGURE 6.2: Our method for inferring search patterns for taint-style vulnerabilities [165]

where $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a function that measures the distance between two objects. In other words, the value of the linkage function for two clusters X and Y is given by the maximum distance among all distances of pairs (x, y) with $x \in X$ and $y \in Y$. Linkage clustering with this particular choice of D is known as *complete-linkage clustering*, a method known to create compact groups of objects while being easy to calibrate [see 4].

6.3 Inferring Search Patterns

We are now ready to discuss our method for inferring search patterns for taint-style vulnerabilities. Starting from a security sensitive sink such as `memcpy`, the method generates search patterns in the form of graph traversals that encode common sources of arguments along with their respective checks. In order to uncover vulnerabilities, these traversals return calls to the sink where source descriptions match, but no match can be found for at least one of the checks, thereby leading the analyst to sink invocations where typical checks are not in place. To be useful, these traversals need to encode patterns in the code as opposed to specific invocations. In addition, they need to capture data flow precisely across functions to allow definitions of individual arguments to be correctly tracked. Finally, traversals should be easy to understand and amendable by the analyst to allow additional domain knowledge to be incorporated.

In order to generate search patterns with these qualities, we combine code property graphs extended for interprocedural analysis (see Section 2.3.4), our embedding procedure (see Section 3.5), and clustering in a four-step procedure illustrated by Figure 6.2 and described in the following.

- **Generation of definition graphs.** We begin by determining all calls to the selected sink and generate corresponding *definition graphs*, a representation akin to interprocedural program slices [see 63, 78, 157]. These graphs compactly encode how the sink's arguments are sanitized and initialized in a two-level structure that allows to easily enumerate feasible invocations (Section 6.3.1).
- **Decompression and Clustering.** Definition graphs are subsequently decompressed to enumerate individual invocations. Invocations are then clustered to determine sets of invocations with similar argument initializers (Section 6.3.2).
- **Creation of sanitization overlays.** For each cluster of invocations, we subsequently determine potential sanitizers for arguments, that is, conditions executed in between initialization and sink invocation (Section 6.3.3).

- **Generation of graph traversals.** Finally, we generate search patterns from clusters in the form of graph database traversals for our robust code analysis platform (Section 6.3.4).

In the following, we describe each of these steps in detail, and make use of the running example shown in Figure 6.3 for illustration.

```
int bar(int x, int y) {                    1
     int z;                                2
     boo(&z);                              3
     if (y < 10)                           4
       foo(x,y,&z);                        5
 }                                         6
                                           7
 int boo(int *z) {                         8
     *z = get();                           9
 }                                         10
                                           11
int moo() {                                12
     int a = get();                        13
     int b = 1;                            14
     bar(a, b);                            15
 }                                         16
                                           17
 int woo() {                               18
     int a = 1;                            19
     int b = get();                        20
     bar(a, b);                            21
 }                                         22
```

FIGURE 6.3: Running example for inference of search patterns [165]

The respective code shows a call to the sink `foo` with three arguments (line 5), where the arguments can be initialized in two different ways: on the one hand, the function `bar` enclosing the call to `foo` may be called by the function `moo`, on the other, it may be called by `woo`, resulting in different initializations of the variables `x` and `y`. Finally, the variable `z` is initialized by the function `boo` that is called by `bar` before calling `foo`.

6.3.1 Generation of Definition Graphs

Both methods for learning-based vulnerability discovery presented in the previous two chapters rest on the comparison of program functions. In contrast, the method presented in this chapter compares sink invocations, that is, *function calls*. A key challenge to address is therefore to choose a suitable representation for function calls that encodes the initialization of all arguments passed to the sink, as well as their respective validation. This is not trivial as the statements providing this information are often spread across several functions and buried in unrelated code.

We choose *definition graphs* to represent function calls, a graph-based representation that contains all necessary information for the generating of search patterns for taint-style vulnerabilities. These graphs contain a carefully chosen subset of the nodes of the corresponding interprocedural program slice [see 63, 157], nodes that can be easily extracted from the code property graph, so long as it has been extended for interprocedural analysis as described in Section 2.3.4.

Definition graphs are constructed by first modeling functions locally, and then combining these graphs in a second step to model function interaction. We proceed to describe these steps in detail.

6.3.1.1 Local Function Modeling

Determining for a statement which other statements affect it is a classical problem in program analysis that is addressed by program slicing techniques [see 63, 78, 157]. With a program dependence graph at hand (see Section 2.2.4.3), program slicing amounts to following incoming data-flow and control-dependence edges recursively. In the spirit of these techniques, we create a hierarchical representation that captures all definition statements involving variables used in the call as well as all conditions that control the execution of the call site. As an example for the construction of this representation, the call to the function `foo` (line 5) in Figure 2.13 is considered as a sink of interest. Starting at its corresponding node in the code property graph, we create this hierarchical representation by passing over the code property graph according to the following rules.

- For the selected sink, we begin by following outgoing syntax edges to its arguments. In the example, the sink `foo` is expanded to reach the arguments `x`, `y`, and `z`.
- For arguments and the statements they are defined by, we proceed to follow data-flow and control-dependence edges to determine further definition statements and conditions that control whether the call site is executed or not. In our example, the definition `int z` and the condition `y < 10` are uncovered in this way.
- Finally, interprocedural edges are followed from calls that define any of our variables to their function bodies, allowing us to identify further defining statements affecting the arguments of the selected sink. From there, we immediately traverse to definition statements that reach the exit node of the function via data flow edges. This leads us to definition statements inside callees that affect arguments and the return value of the call. In our example, we thus uncover the call to `boo`, which leads us to the definition statement `*z = get()`.

If by application of these three rules, we reach a parameter, we consider all respective call sites as sinks as well and locally model functions of these call sites by applying the same three rules. For each function, we thus obtain a tree that can be easily constructed from the code property graph via a depth first traversal with an expansion function that implements the three rules.

6.3.1.2 Definition Graphs

The tree representations of functions already allow argument definitions and their sanitization to be analyzed locally, however, function callers remain unexplored. With respect to our example, we can therefore uncover the local variable definition of `z`, however, the we do not trace the definitions of the parameters `x` and `y` past the function boundary. Unfortunately, simply expanding interprocedural data flow edges connecting these parameters to their arguments leads to infeasible combinations of definitions as discussed in detail by Reps [114]. Our example makes this clear: following this simple strategy, we

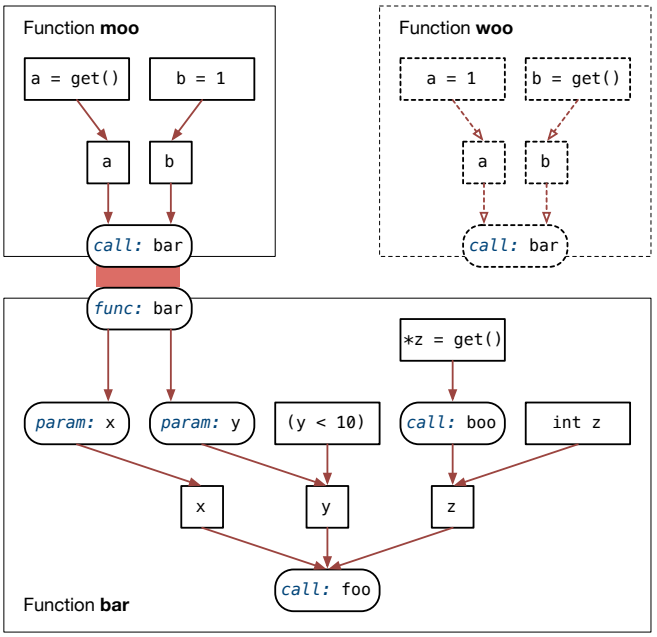


FIGURE 6.4: Definition graph for the call to `foo` [165]

uncover the combination {`int a = get(), int b = get()`} in particular. Unfortunately, this combination is invalid as the first definition only occurs for the caller `moo`, and the second occurs for the caller `woo`, however, these definitions never occur in combination.

This classic problem of interprocedural program analysis can, for example, be solved bu formulating a corresponding context-free-language reachability problem [114]. A simpler solution is to ensure that the parameter nodes of a function are always expanded together as the graph is traversed.

The *definition graph* implements this idea by simply tying parameter together, modeling the interplay of entire functions as opposed to parameters and their arguments. To this end, the nodes of the definition graph correspond to the trees that locally model functions, and we connect these trees by edges expressing their calling relations. Definition graphs are therefore a two-level structure that connect entire trees via edges to form a graph of trees. As an example, Figure 6.4 shows the definition graph for the sink `foo` of the sample code from Figure 6.3. We define definition graphs formally as follows.

Definition 6.2. A *definition graph* $G = (V, E)$ for a call site c is a graph where V consists of the trees that model functions locally for c and those trees of all of its direct and indirect callers. For each $a, b \in V$, an edge from a to b exists in E if the function represented by a calls that represented by b [165].

6.3.2 Decompression and Clustering

At this point, definition graphs are available for all call sites of the selected sink, and we proceed to decompress these graphs to enumerate individual invocations. Upon decompression, our method determines groups invocations containing invocations similar in the way they initialize their arguments. As the reader may have guessed by now, we determine these groups by embedding invocations in a vector space and subsequently

clustering them, and as it true for the methods proposed in the previous chapters, the key to the success of this approach lies in the choice of a suitable feature map. In particular, the vectors obtained using the feature map should allow invocations to be compared in the way each of their arguments are initialized, being robust against variations in the formulation of initializing statements.

To meet these demands, we implement a multi-stage feature map (see Section 3.4.5) that *represents invocations by the clusters of API symbols their initializers occur in.* We achieve this by first clustering API symbols according to a string metric, and then determining for each initializer of each invocation which clusters their API symbols occur in. Each argument of an invocation can subsequently be represented by a *bag of cluster identifiers.* Concatenating the resulting vectors obtained for individual arguments, we thus obtain a representation of invocations that encodes the initialization of all of its arguments, finally allowing us to cluster invocations. We now describe decompression, clustering of API symbols, and finally, clustering of invocations in detail.

6.3.2.1 Decompression of Definition Graphs

Each definition graph describes a single call site, however, it possibly encodes several combinations of argument definitions. For example, the definition graph shown in Figure 6.4 for the invocation of `foo` contains both the combination {`int z, a = get(), b = 1`} and the combination {`int z, a = 1, b = get()`} in a compressed form. Fortunately, individual combinations can be easily enumerated from the definition graph using the simple recursive procedure shown by Algorithm 3. In this algorithm, $[v_0]$ denotes a list containing only the node v_0 and the operator $+$ denotes list concatenation.

Algorithm 3 Decompression of definition graphs [165]

```
1:  procedure DECOMPRESS(G)
2:      return RDECOMPRESS(G, r(V))
3:  procedure RDECOMPRESS(G := (V, E), v_0)
4:      R = ∅
5:      D ← PARENTTREES(v_0)
6:      if D = ∅ then
7:          return {[v_0]}
8:      for d ∈ D do
9:          for L ∈ RDECOMPRESS(G, d) do
10:             R ← R ∪ ([v_0] + L)
11:     return R
```

We recall that the nodes of the definition graph are trees that model functions locally. Starting from the root node $r(V)$, the algorithm recursively combines the current tree with all possible call chains, that is, all lists of trees in the code base that lead to this tree. As a result, we obtain the set $\mathcal{X}$ of all observed invocations, represented by their combination of argument initializers.

6.3.2.2 Clustering of API Symbols

API symbols with a similar name often implement similar functionality. For example, the functions `malloc` and `realloc` are both concerned with memory allocation, while the

functions `strpcy` and `strcat` deal with string processing. As we want to be able to detect similar initializers even if the API symbols they employ do not match exactly, we first create clusters of similar API symbols, where members of a cluster are all eventually mapped to the same dimension of our final feature space.

To cluster API symbols, we perform complete-linkage clustering. As discussed in Section 6.2, this requires a distance function to be specified that allows the similarity of API symbols to be assessed. While many string metrics exist that can be employed in this setting, we choose the Jaro distance [66] for this task, a string metric specifically designed with short strings in mind. For any two strings, the Jaro distance quantifies their similarity by a value between 0 and 1, where a value of 1 indicates that the strings are exactly the same, while a value of 0 indicates that they do not match at all.

We can now perform linkage clustering on API symbols simply by choosing a minimum similarity of strings inside clusters. This parameter can be employed by the analyst to control the granularity of the clustering, however, we found values of the Jaro distance between 0.7 and 0.9 to produce a relatively stable clustering, and thus we fixed the value to 0.8 for all experiments.

The clustering is applied to the API symbols of each argument separately, thereby keeping the number of strings to compare small. As a final result of this step, we obtain a set C of API-symbol clusters for each argument. With respect to our description of multi-stage feature maps, this step corresponds to the clustering of sub structures, and we can now proceed to represent objects by their clusters.

6.3.2.3 Clustering Invocations

We proceed to represent invocations by the clusters the API symbols of their initializers occur in, and determine groups of similar invocations by clustering. To simplify discussion, let us assume a sink with a single argument. We recall that $\mathcal{X}$ is the set of all invocations represented by their combinations of argument initializers, and C is the set of API-symbol clusters. Then, for each invocation $x \in \mathcal{X}$, we can determine the set of clusters $C_x \subseteq C$ its definitions are contained in. This is sufficient to implement a multi-stage feature maps as discussed in Section 3.4.5. A multi-stage feature maps is given by a function $\phi : \mathcal{X} \to \mathbb{R}^{|C|}, \phi(x) = (\phi_c(x))_{c \in C}$ maps objects to $|C|$-dimensional vectors, that is, it is a bag-of-words embedding where the embedding language is given by the natural numbers from 1 to $|C|$. To encode in each coordinate of $\phi(x)$ whether x is associated with a cluster or not, we define ϕ_c as

$$\phi_c(x) = \begin{cases} 1 & \text{if } c \in C_x \\ 0 & \text{otherwise} \end{cases}$$

that is, $\phi_c(x)$ indicates whether the cluster c occurs in any of the API-symbol clusters C_x associated with the invocation x.

For sinks with multiple arguments, this operation is performed for each sink independently and the resulting vectors are simply concatenated. As an example, we consider an invocation x where the first argument is initialized by a call to `malloc`, and the second is defined as a local variable of type `size_t`. Then, the corresponding vector may have the following form.

$$\phi(x) \mapsto \begin{pmatrix} \dots \\ 0 \\ 1 \\ \dots \\ 1 \\ 0 \\ \dots \end{pmatrix} \begin{array}{l} \dots \\ \{\texttt{char[52]}, \texttt{uchar[32]}, \dots\} \\ \{\texttt{malloc}, \texttt{xmalloc}, \dots\} \\ \dots \\ \{\texttt{size\_t}, \texttt{ssize\_t}, \dots\} \\ \{\texttt{int}, \texttt{uint32\_t}, \dots\} \\ \dots \end{array} \begin{array}{l} \\ \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Arg. 1} \\ \\ \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Arg. 2} \\ \dots \end{array}$$

Upon mapping all invocations to a feature space, we employ linkage clustering to determine groups of similar invocations. In this case, we choose the city-block distance as a measure of similarity as it provides an intuitive way to measure the number of matching API-symbol clusters of two vectors. Throughout our experiments, we fix the maximum distance of members in a cluster to 3, meaning that up to 3 clusters may be different for members of the same cluster,

The final result of this step is a set of clusters, each consisting of sets of similar invocations, constituting patterns in the code base. Moreover, the size of these clusters allows to rank clusters according to their strength, that is, the number of individual invocations that support the pattern.

6.3.3 Creation of Sanitization Overlays

The clusters uncovered in the previous step correspond to argument definition models that provide the necessary information to generate search patterns indicating common combinations of arguments for the sink of interest. However, they do not contain information about argument sanitization. We address this limitation by creating overlays for argument definition models that express typical sanitizations for each argument. To this end, we analyze each argument separately, combing the information contained in associated conditions spread over all of the definition graphs associated with the argument definition model.

Our goal is to determine conditions commonly associated with the argument. However, as is true for the detection of common argument sources, we aim to be robust against slight changes in the way conditions are formulated, as finding the exact same condition in source code multiple times is rather unlikely. We achieve this by following the same idea already followed for finding common sources: we cluster conditions and count cluster occurrences as opposed to occurrences of individual conditions. To this end, we embed and cluster conditions as follows.

Embedding conditions. For our set of objects $\mathcal{X}$, we choose the set of conditions associated with the argument of interest, represented by their syntax trees. As conditions commonly contain commutative logical operators, e.g., the logical `and` operator, we choose a graph-based feature map (see Section 3.4.4) as these maps are robust against changes in child-node order. Recalling graph-based embeddings, we need to choose a labeling function l that assigns initial labels to all nodes. We choose l such that it assigns a hash value calculated from the `type` attribute and the `code` attribute for inner nodes and leaf nodes respectively. With initial labels assigned to nodes, the feature map

calculates neighborhood hashes for each node, and finally, represents each condition by the number of occurrences of these hash values.

Clustering conditions. Upon mapping conditions to a feature space, we employ linkage clustering yet again using the city-block distance with a fixed parameter of 2. This yields clusters of conditions. We store the number of times each cluster occurs along with the argument definition model, allowing us account for this information when generating search patterns.

6.3.4 Generation of Graph Traversals

Clusters of source-to-sink systems enhanced with sanitization overlays fully express search patterns that we can now finally express as traversals for our code mining system (see Section 2.1. We achieve this by filling out a generic template for the description of taint-style vulnerabilities that can be instantiated using the information contained in our clusters.

6.3.4.1 Traversal Template

Figure 6.5 shows our template for the discovery of taint-style vulnerabilities formulated in the traversal language Gremlin (Section 2.4.3). Instantiating this template requires the name of the sink to be specified along with descriptions for all arguments and their sanitization. These are referred to as `arg`*i*`Source` and `arg`*i*`Sanitizer` in the template where *i* denotes the argument number. The traversal uncovers all call sites of the specified sink and proceeds to independently process each sink using the traversals `taintedArgs`, followed by the traversal `unchecked`.

```
getCallsTo(sink)

.taintedArgs(
        [arg1Source,..., argnSource]
)

.unchecked(
        [arg1Sanitizer, ... argnSanitizer]
)
```

FIGURE 6.5: Template for taint-style vulnerability as a graph traversal in the query language Gremlin [165].

The traversal `taintedArgs` allows to check whether the sources of arguments passed to the sink match the provided source descriptions (`arg`*i*`Source`). It achieves this by generating the corresponding definition graph for the sink's call site (see Section 6.3.1). For each source description, it proceeds to determine whether the call site can possibly fulfill it. This can be achieved without decompressing the definition graph by simply checking whether for each source description, at least one matching statement exists in the definition graph.

While this step drastically decreases the number of call sites to examine in detail, we do not know with certainty that the definition graphs thus uncovered matches the source descriptions. For example, the particular combination of source descriptions we seek

may be present in the definition graph but invalid (see Section 6.3.1.2). To analyze only valid combinations of argument definitions, the traversal proceeds to decompress the definition graph using Algorithm 3. With definition combinations at hand, it is now trivial to check whether argument descriptions are matched. As a result, the traversal now returns all definition combinations matching the source descriptions to pass them on to the traversal `unchecked`.

Finally, the traversal `unchecked` determines all call sites with at least one missing sanitizer according to the sanitizer descriptions. To this end, the traversal proceeds by checking each of the conditions in the definition graphs against the respective sanitizer descriptions.

6.3.4.2 Template Instantiation

To instantiate the template presented in the previous section, source-to-sink clusters simply need to be translated into source and sanitizer descriptions. We recall that for each argument, a source-to-sink cluster contains (a) a set of API-symbol clusters for sources, and (b) a set of clusters containing normalized expressions for conditions. These clusters merely need to be summarized in a format suitable to be understood and adapted by analysts. We achieve this by generating regular expressions from these clusters by determining longest common subsequences as often performed in signature generation [see 103]. Finally, we can employ the resulting graph traversals to mine code for taint-style vulnerabilities.

6.4 Evaluation

Remaining consistent with the evaluation methodology of the previous two chapters, our evaluation is two-fold. First, we explore our method's ability to generate search patterns for known vulnerabilities in a controlled setting, where we measure its ability to reduce the amount of code to review (Section 6.4.1). Second, we assess the practical merits of our approach by using it as the primary method for uncovering previously unknown vulnerabilities in a real code audit (Section 6.4.2).

6.4.1 Controlled Experiment

To evaluate our method's ability to generate traversals for real vulnerabilities, we begin by reviewing the security history of five popular open-source projects for taint-style vulnerabilities: the cryptographic library OpenSSL, the media player VLC, the rendering library Poppler as used in the document viewers Evince and Xpdf, and again, the instant messenger Pidgin, and the Linux kernel. We determine a recent taint-style vulnerability and the corresponding sink for each of these code bases. Table 6.1 summarizes our data set. For each vulnerability, it contains the project name and its version, the component the vulnerability appears in, its CVE identifier, the corresponding sensitive sink, and the number of callers of the sink present in the code base. The following description of this data set is taken verbatim from [165].

- **CVE-2013-4513 (Linux).** An attacker-controlled variable named `count` of type `size_t` is passed as a third argument to the sink `copy_from_user` without being sanitized, thereby triggering a buffer overflow.
- **CVE-2014-0160 (OpenSSL "Heartbleed").** The variable `payload` of type `unsigned int` as defined by the source `n2s` is passed as a third argument to `memcpy` without being checked, causing a buffer overread.
- **CVE-2013-6482 (Pidgin).** The string `unread` is read from the attacker-controlled source `xmlnode_get_data` and passed to the sink `atoi` without undergoing sanitization, thereby possibly causing a NULL pointer to be dereferenced.
- **CVE-2012-3377 (VLC).** The length of the data buffer `p_stream->p_headers` is dependent on an attacker-controlled allocation via the function `realloc` and reaches a call to `memcpy` without verifying the available buffer size, leading to a buffer overflow.
- **CVE-2013-4473 (Poppler).** The attacker-controlled string `destFileName` is copied into the local stack buffer `pathName` of type `char [1024]` using the function `sprintf` without checking its length, leading to a stack-based buffer overflow.

For all of these sinks, we generate traversals as summarized by Table 6.2. The table shows the number of traversals generated for each vulnerability, and whether our method is able to generate a traversal that expresses the correct source and sanitizer. Moreover, it shows the time required for traversals generation and execution in seconds. Finally, the last column of the table shows the *reduction percentage*, that is, the percentage of call sites that do not have to be inspected as they are not returned by the traversal.

In all cases, our method generates correct descriptions for the respective argument sources. Correct descriptions for sanitizers are generated for all vulnerabilities except CVE-2013-4473. In this case, no sanitizer description is generated at all, as only 22 call sites are available, making the inference of a sanitizer description difficult using statistical methods. Nevertheless, the reduction percentage is high for all vulnerabilities, making it possible to locate all vulnerabilities while skipping inspection of 94.9% of the call sites on average.

Project	Version	Component	LOC	CVE	Sensitive sink	# Sites
Linux	3.11.4	Drivers	6,723,955	2013-4513	`copy_from_user`	1715
OpenSSL	1.0.1f	All	378,691	2014-0160	`memcpy`	738
Pidgin	2.10.7	All	363,746	2013-6482	`atoi`	255
VLC	2.0.1	All	555,773	2012-3377	`memcpy`	879
Poppler	0.24.1	All	227,837	2013-4473	`sprintf`	22

TABLE 6.1: Data set of five open-source projects with known taint-style vulnerabilities. The table additionally lists the sensitive sinks of each vulnerability and the number of traversals inferred by our method [165].

Finally, Table 6.3 shows the sink, along with the inferred regular expressions for sources and sanitizers, where for arguments with multiple sanitizers, only one is shown. The regular expressions describe the correct sources, and the sanitizer descriptions match those necessary to ensure secure operation of the source-sink system. Moreover, additional sanitizers are inferred in some cases. For example, it is determined that the first argument to `memcpy` from the source `n2s` is often compared to `NULL`, ensuring that it is not a `NULL` pointer.

	Correct Source/Sanitizer	# Traversals	Generation Time	Execution Time	Reduction[%]
CVE-2013-4513	✓✓	37	142.10 s	10.25 s	96.50
CVE-2014-0160	✓✓	38	110.42 s	8.24 s	99.19
CVE-2013-6482	✓✓	3	20.76 s	3.80 s	92.16
CVE-2012-3377	✓✓	60	229.66 s	20.42 s	91.13
CVE-2013-4473	✓	1	12.32 s	2.55 s	95.46
Average					94.90

TABLE 6.2: Reduction of code to audit for discovering the five taint-style vulnerabilities. For the last vulnerability no correct sanitizer is inferred due to the low number of call sites [165].

	CVE-2013-4513	CVE-2014-0160	CVE-2013-6482	CVE-2012-3377	CVE-2013-4473
Sink	`copy_from_user`	`memcpy`	`atoi`	`memcpy`	`sprintf`
Argument 1	`.*`	`.*`	`.*xmlnode_get_.*`	`.*alloc.*`	`.*char \[ .* \].*`
Argument 2	`.*const .*cha.*r *.*`	`.*`	`.*`	`.*`	`.*`
Argument 3	`.*size_t.*`	`.*n2s.*`	`.*`	`.*`	`.*`
Sanitizer 1	-	`.*sym ( ==|!=) NULL.*`	`.*sym.*`	`.*sym.*`	-
Sanitizer 2	-	-	-	-	-
Sanitizer 3	`.*sym .*(\d+).*`	`.*sym.*\+(\d+).*`	-	-	-

TABLE 6.3: Regular expressions contained in the search patterns for the five taint-style vulnerabilities, where *sym* is replaced by the tracked symbol at runtime. For the last vulnerability, no sanitizers are inferred [165].

6.4.2 Case Studies

In the following, we present two case studies, one for the generation of signatures from known bugs, and another where we enumerate taint-style systems. First, we use our method to generate a signature for the *Heartbleed* vulnerability, showing that it captures both of its instances robustly. Second, we employ the method to enumerate all taint-style systems associated with the sink `memcpy` in the popular open-source media player, showing that we immediately uncover previously unknown vulnerabilities by choosing particularly interesting patterns alone.

6.4.2.1 Case Study: The Heartbleed Vulnerability

In this case study, we show that our method is capable of successfully generating a search pattern for the "Heartbleed" vulnerability in OpenSSL, our motivating example introduced in Section 6.1. To this end, we import the code of OpenSSL version 1.0.1f, the last version of the library that contains the vulnerability. We proceed to employ our method to generate patterns for the security-sensitive sink `memcpy`, a sink from the C standard library commonly associated with buffer overflow vulnerabilities, similar to the functions `strcpy`, `sprintf`, or `strcat` [see 6, 35].

First, the heuristic for the discovery of library functions that taint their arguments is employed (see Section 2.3.4.1). The results of this analysis are shown in Table 6.4. We manually verify these results and find that, with the exception of one, all are inferred correctly. The falsely identified tainted third argument of the function `memset` results from the fact that this argument is often of the form `sizeof(buffer)`, where buffer is a variable that reaches `memset` without prior definition. This problem can be easily fixed by slightly adapting our heuristic to suppress arguments of `sizeof`, leaving us with no false positives in this particular case.

Function	Defining
fgets	1. argument
sprintf	1. argument
memset	1. argument
write	3. argument
memcpy	1. argument
memset*	3. argument
n2s	2. argument
n2l	2. argument
c2l	2. argument

TABLE 6.4: Inferred argument definitions [165]

Regular expression
.*n2s.*
.*memset.*
.*strlen.*
.*int .*len.*
.*int arg.*
.*size_t.*
.*unsigned.*
.*int.*
.*long.*

TABLE 6.5: Inferred third arguments of `memcpy` [165]

We proceed to generate queries for the sink `memcpy`, resulting in 38 queries, 14 of which specify a source for `memcpy`'s third argument. As the third argument specifies the amount of data to copy, cases where this argument is attacker-controlled are particularly interesting when searching for buffer overflows and overreads. The third arguments to `memcpy` inferred by our method are shown in Table 6.5. In particular, it contains the source `n2s`. As this is the only source that is attacker-controlled with certainty, only the traversal shown in Figure 6.6 needs to be executed.

```
arg3Source = sourceMatches('.*n2s.*');

arg2Sanitizer = { it, symbol ->
     conditionMatches(".*%s (==|!=) NULL.*", symbol)
};

arg3Sanitizer = { it, symbol ->
     conditionMatches(".*%s.*\+(\d+).*", symbol)
};

getCallsTo("memcpy")
.taintedArgs([ANY, ANY, arg3Source])
.unchecked([ANY_OR_NONE, arg2Sanitizer, arg3Sanitizer])
```

FIGURE 6.6: Generated traversal encoding the vulnerable programming pattern leading to the Heartbleed vulnerability [165].

Reviewing the generated query, we see that it specifies the desired sink `memcpy` on line 11, and describes the data flow from the attacker-controlled source `n2s` to the third argument of this sink on line 1. In addition, it enforces two sanitization rules. On line 4, it is ensured that second arguments to `memcpy` are checked against `NULL`, as the source buffer for the copy operation may not be a `NULL` pointer. Second, line 8 ensures that the third argument to `memcpy`, i.e., the amount of data to copy should be checked in an expression containing an integer. The automatically generated thus already expresses the vital validation of the third argument to the copy-operation along with an additional rule that is less important. Of course, an analyst can easily edit this query, e.g., to exclude the less important rule, however, for the purpose of this evaluation, we leave the query as is to assess the methods ability without the analyst intervention. In fact, even without the analyst's modifications, the traversal returns only 7 call sites of 738, that is, only 0.81%. Table 6.6 shows the returned functions, among which two correspond to the "Heartbleed" vulnerability. Finally, Figure 6.7 shows the code property graph that the two vulnerable functions have in common, including the check placed between source and sink to patch the vulnerability. As this graph shows, the traversal returns the function if this check is removed, and does not return it, if the check is in place.

Filename	Function
ssl/d1_both.c	dtls1_process_heartbeat
ssl/s3_clnt.c	ssl3_get_key_exchange
ssl/s3_clnt.c	ssl3_get_new_session_ticket
ssl/s3_srvr.c	ssl3_get_client_key_exchange
ssl/t1_lib.c	ssl_parse_clienthello_tlsext
ssl/t1_lib.c	tls1_process_heartbeat
crypto/buffer/buf_str.c	BUF_memdup

TABLE 6.6: The seven hits returned by the generated traversal. Vulnerable call sites are shaded [165].

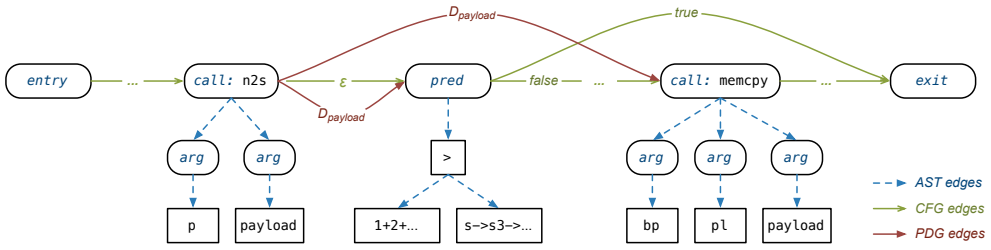


FIGURE 6.7: Excerpt of the code property graph for the Heartbleed vulnerability [165]

6.4.2.2 Case Study: The VLC Media Player

We now employ our method to uncover five previously unknown vulnerabilities in the popular open-source media player VLC. To this end, we again generate queries for the sink `memcpy`, and select two queries that look particularly interesting, as they encode dangerous programming practices. Figure 6.8 shows the first of the two queries, which describes calls to `memcpy` where the destination buffer, that is, the first argument, is defined as a local stack buffer of type `char`. In addition, the size of this buffer is dynamically calculated as part of its declaration. This way of declaring a local buffer already constitutes a dangerous programming practice in itself as it is not possible to verify whether at runtime, the available stack memory is sufficient to allocate the buffer. In particular, if the amount of memory requested is attacker-controlled, this can possibly be leveraged to corrupt memory in a subsequent call to `memcpy`.

```
arg1Src = sourceMatches('.*char \[ .*len \+ .* \].*')
arg3Src = { sourceMatches('.*size_t.*')(it) ||
            sourceMatches('.*str.*len.*')(it)}

getCallsTo("memcpy")
.taintedArgs([arg1Src, ANY_SOURCE, arg3Src])
```

FIGURE 6.8: Traversal for dynamic allocations of stack buffers passed as first arguments of memcpy [165].

We run this query and obtain three call sites, all of which are problematic (Table 6.7). As an example, Figure 6.9 shows the vulnerable function `rtp_packetize_xiph_config` where the variable `len` is equal to the length of an attacker controlled string (line 14). This value is subsequently used to allocate the buffer `b64` on the stack (line 14). Finally, `len` bytes are copied into the buffer on line 16. We have successfully verified the existence of this vulnerability by triggering an invalid memory access on a 64 bit Linux system.

The second interesting traversal is shown in Figure 6.10, where the third argument of `memcpy` stems from a source matching the regular expression `.*Get.*`, a family of macros

Traversal	Filename	Function	Line	CVE Identifier
Traversal 1	modules/services_discovery/sap.c	ParseSDP	1187	CVE-2014-9630
Traversal 1	modules/stream_out/rtpfmt.c	rtp_packetize_xiph_config	544	CVE-2014-9630
Traversal 1	modules/access/ftp.c	ftp_SendCommand	122	CVE-2015-1203
Traversal 2	modules/codec/dirac.c	Encode	926	CVE-2014-9629
Traversal 2	modules/codec/schroedinger.c	Encode	1554	CVE-2014-9629

TABLE 6.7: The call sites extracted by our traversals. All call sites are vulnerable [165].

```
int rtp_packetize_xiph_config( sout_stream_id_t *id, const char *fmtp,
                               int64_t i_pts )
{
  if (fmtp == NULL)
    return VLC_EGENERIC;

  /* extract base64 configuration from fmtp */
  char *start = strstr(fmtp, "configuration=");
  assert(start != NULL);
  start += sizeof("configuration=") - 1;
  char *end = strchr(start, ';');
  assert(end != NULL);
  size_t len = end - start;
  char b64[len + 1];
  memcpy(b64, start, len);
  b64[len] = '\0'; // [...]
}
```

FIGURE 6.9: Previously unknown vulnerability found using the first traversal [165].

in VLC that read data directly from media files possibly under the attacker's control. This amount of control over a copy-operation is a common source for buffer overflows, and thus, the query catches our attention.

The two functions returned by the query are shown in Table 6.6. Both functions are vulnerable to a buffer overflow. In particular, Figure 6.11 shows the identified buffer overflow in the source file `modules/codec/dirac.c` where the 32 bit variable `len` is initialized by `GetDWBE` on line 5, and thus possibly attacker-controlled. This variable in subsequently used to allocate the buffer `p_extra` on line 7, however, not before adding the fixed value of `sizeof(eos)` to it. Unfortunately, when choosing `len` to be large enough, this causes an integer overflow in the allocation and a subsequent buffer overflow when data is copied into the buffer on line 10.

```
arg20Source = sourceMatches('.*Get.*');
arg21Source = sourceMatches('.*uint.*_t.*');

getCallsTo("memcpy")
.taintedArgs([ANY_SOURCE,ANY_SOURCE,
              {arg20Source(it) && arg21Source(it)}])
```

FIGURE 6.10: Traversal to identify third arguments of `memcpy` defined by `.*Get.*` [165].

In total, our method played the key role in uncovering 5 previously unknown vulnerabilities that can possibly be exploited to execute arbitrary code. This was achieved by selecting only two promising automatically generated queries, showing that our tool is useful for security analysts reviewing code in practice.

```
static block_t *Encode(encoder_t *p_enc, picture_t *p_pic)
{
 if( !p_enc->fmt_out.p_extra ) {
   // [...]
   uint32_t len = GetDWBE( p_block->p_buffer + 5 );
   // [...]
   p_enc->fmt_out.p_extra = malloc( len + sizeof(eos) );
   if( !p_enc->fmt_out.p_extra )
     return NULL;
   memcpy( p_enc->fmt_out.p_extra, p_block->p_buffer, len);
   // [...]
 }
}
```

FIGURE 6.11: Previously unknown vulnerability found using the second traversal [165].

6.5 Related Work

The idea of instantiating static code checkers using information scattered across a code base can be traced back to the seminal paper by Engler et al. [39]. They show that templates expressing programming rules can be automatically tailored to a target code base. For example, a template may express that *a call to* `A` *must always be followed by a call to* `B`, where `A` and `B` are the template parameters that their approach automatically infers. This allows them to uncover different types of bugs, including security vulnerabilities such as certain types of use-after-free vulnerabilities, double-free vulnerabilities, and immediate dereferenciations of user-space pointers in kernel code. However, in contrast to the queries generated by our approach, none of the presented rule templates is able to model checks in source-to-sink systems, and thus, taint-style vulnerabilities remain out of reach.

Kremenek et al. [77] provide a method that combines different sources of information in a factor graph to automatically infer functions that allocate or deallocate resources. More recently, Livshits et al. [86] present Merlin, a method also based on factor graphs that allows information flow specifications to be inferred for Web applications written for the Microsoft .NET framework. Unfortunately, Merlin is limited to modeling the flow of information between functions, and hence, sources, sinks, and sanitizers are required to be calls to functions. This assumption holds for many important incarnations of Web application vulnerabilities such as SQL injection, and cross site scripting vulnerabilities, however, missing bounds checks for vulnerabilities such as buffer overflows or null pointer checks typical for system code cannot be detected in this way. In contrast, our method allows us to derive sanitizers from arbitrary statements, making it possible to encode a wider range of checks, and missing bound checks in particular.

7 Conclusion and Outlook

Providing analysts with practical methods to assist in the discovery of vulnerabilities is crucial for securing computer systems. Even with improvements of programming languages in sight that can eventually eliminate certain types of vulnerabilities completely, the systems created today and in the past will remain important for the security of our communication infrastructure for years to come. Reviewing recent history and the advent of Web applications also suggests, that new types of vulnerabilities specific to these technologies will surface, based on attacker-control, sensitive operations, and failure to restrict the attacker's capabilities.

This thesis has introduced *pattern-based vulnerability discovery*, a family of methods based on machine learning and graph mining which assist analysts in day-to-day auditing. Rooted in pragmatism, our methods are specifically designed to be operated interactively by an analyst, allowing her to benefit from the machine's pattern recognition capabilities without surrendering the ability to guide the analysis and make final security critical decisions. Moreover, our methods trade the precision offered by methods such as symbolic execution for the speed and scalability of lightweight analysis, and simplicity in operation.

First and foremost, these approaches highlight the merits of learning techniques for vulnerability discovery, allowing us to reduce the necessary effort to uncover severe vulnerabilities in mature real world code bases. In addition, their realization has lead to the development of several new techniques for robust code analysis such as *refinement parsing*, the *code property graph*, and in particular, *feature maps for source code* that enable machine learning algorithms to uncover complex patterns in programs. These techniques are interesting for code analysis in general and have already been applied by other researchers for identifying code reuse [3], de-anonymization programmers [19], and decompiling binary code [161].

In the following, we discuss the results of this thesis in greater detail and proceed to point out limitations of our pattern-based approaches. Finally, we outline ideas for future work based on the results presented in this thesis.

7.1 Summary of Results

The results of this thesis are twofold. On the one hand, we have presented a novel platform for robust code analysis that can be used to mine for vulnerabilities using graph database queries. On the other, we have studied the ability of unsupervised machine learning algorithms to assist in the discovery of vulnerabilities, and have provided a practical method for real world audits for each of the three main capabilities of unsupervised learning. Finally, the last of the three approaches closes the loop by showing that graph database queries for our code analysis platform can in fact be inferred from the code using learning techniques, allowing us to express patterns identified by the learner in an explicit and editable form. Zooming in on the concrete methods, the following main contributions have been made.

Mining for vulnerabilities with Graph Databases. We have presented a platform for robust source code analysis that combines ideas from fuzzy parsing, classic compiler design, and the emerging field of graph databases. In particular, we have introduced *refinement parsing* as a practical approach for parsing possibly incomplete code robustly, and the code property graph, a joint representation of a program's syntax, control flow and data flow. Storing code property graphs in graph databases has allowed us to effectively mine source code for vulnerabilities using manually crafted search patterns. We have showed in an empirical evaluation that these search patterns are expressive enough to create queries for the majority of vulnerabilities reported in the Linux kernel in 2012. Moreover, we have conducted a real world audit employing our method where we uncover 18 previously unknown vulnerabilities in the Linux kernel (Chapter 2).

Feature maps for source code. We have proceeded to introduce different ways for embedding source code in a vector space, a prerequisite for the application of many machine learning algorithms. These maps provide the bridge between our code analysis platform on the one hand, and machine learning on the other, making it possible to determine patterns in code associated with its syntax, control flow, and data flow. In addition to defining these feature maps abstractly, we have discussed how they can be realized at scale via feature hashing, and have provided a generic procedure for implementing these embeddings based on the code property graphs. This makes it possible to implement all techniques for pattern-based vulnerability discovery presented in this thesis directly for our architecture (Chapter 3).

A method for vulnerability extrapolation. We consider the scenario where an analyst is interested in finding vulnerabilities similar to a known vulnerability. To this end, we have presented a method that identifies similar functions to a known function based on dimensionality reduction, and a representation of code in terms of the subtrees of its syntax tree. Comparing several variants of this representation, we find empirically that limiting ourselves to subtrees of API symbols allows to narrow the set of functions to inspect down to 8.7% on average. Moreover, we have shown that we are able to leverage this method in practice to identify previously unknown vulnerabilities in the media decoding library FFmpeg, and the instant messenger Pidgin (Chapter 4).

A method for missing-check detection. We have proceeded to present a method that points out particularly pronounced deviations from programming patterns by extending our method for extrapolation by contextual anomaly detection (see Section 5.2). The main idea is to identify functions that deviate notably in the way they handle input when compared to functions operating in a similar context. The method thus focuses on the detection of missing checks by automatically determining the checks typically associated with a source or sink via lightweight data flow analysis. We have also shown in an empirical analysis on known vulnerabilities that our method is capable of detecting missing checks accurately with few false alarms, albeit these may not be security critical. In addition, we have again employed the resulting method in practice, allowing us to uncover previously unknown vulnerabilities in Pidgin and LibTIFF (Chapter 5).

A method for inference of search patterns. Finally, we have presented a method to automatically extract common source-to-sink systems from source code and the checks associated with their parameters to generate graph database queries. These queries allow source-sink-systems to be identified with insufficient validation of input. Moreover, they express the identified patterns in an explicit form, allowing the analyst to understand which patterns were identified, and to augment these search patterns with information that cannot be derived from the statistics of the code base. In a controlled setting, we find that this method allows us to create accurate signatures for taint-style methods that reduce the amount of code to inspect for these particular flaws by 94.9%. Moreover, we have shown that our method allows us to uncover eight previously unknown vulnerabilities in the VLC media player by enumerating problematic taint-style systems (Charpter 6).

7.2 Limitations

The results presented in the previous chapters demonstrate the merits of pattern-based techniques for the discovery of vulnerabilities in real world applications. Nonetheless, there are several limitations of our approach, both of inherent and of technical nature.

First of all, it needs to be pointed out that the problem of vulnerability discovery is undecidable in the general case [115], meaning that we cannot devise an automated method that determines all vulnerabilities in a program without any false positives. Moreover, it is equally unrealistic to prevent vulnerable code to be created in the first place: vulnerabilities are the result of oversights, misjudgment of the attacker's control over the program, and unfortunate design decisions. While improvements introduced by programming languages and environments may alleviate the possibility of introducing severe vulnerabilities, it will not be possible to entirely eliminate human error.

Second, our work builds on machine learning techniques such as embedding of code in a feature space, dimensionality reduction, anomaly detection, and clustering. These approaches are useful to narrow in on *potentially* vulnerable code, but provide no guaranties for the existence of a vulnerability. This is an inherent problem of machine learning techniques as they focus mainly on the code's statistics as opposed to its semantics. Nonetheless, finding interesting starting points for more precise analysis in the large amounts of code contemporary systems are composed of, is a useful feature, as our experiments show.

Third, our techniques do not account for the situation where a programmer purposely introduces a vulnerability to place a backdoor inside the code. In this scenario, the programmer may make a significant effort to hide the vulnerability, for example, by obfuscating code or dynamically generating it. To improve upon this situation, it may be possible to combine techniques from malware analysis with our approaches in the future.

Fourth, the applicability of our methods has been shown for different programs, but not for different programming languages. It is an open question whether our method can for example be applied to dynamic programming languages such as PHP or Javascript. Moreover, it is also unclear whether they can be operated on binary code.

Finally, our techniques focus on vulnerabilities that manifest in few functions, and can be localized. However, different types of design flaws may arise due to the interplay of system components, and it may not be possible to pinpoint exact locations in the code that can be associated with the vulnerability. These types of flaws cannot be identified using our methods, albeit they may be helpful in gaining the necessary understanding of the code base to uncover them.

7.3 Future Work

It is the authors hope that pattern-based vulnerability discovery as introduced in this thesis will provide a fruitful ground for further research. In particular, starting from the work presented, the following directions seem interesting for the development of improved systems to assist in vulnerability discovery.

Combination with precise techniques. Precise techniques of program analysis such as symbolic execution provide a useful tool to check code for specific properties, but they are also hard to scale to today's systems. In the future, it may be interesting to combine our approaches with these precise techniques, for example, by using pattern-based techniques to select interesting code, which is subsequently analyzed using symbolic execution. This could drastically reduce the amount of false positives produced by inexact methods, and possibly even provide the analyst with information necessary to determine whether a defect is exploitable.

Exploiting new data sources. A strength of the property graph in the setting of code analysis is that it allows us to combine different aspects of the code in a single data structure, at scale. However, the approaches presented so far rest entirely on the results of static analysis of the code in question. In the future, overlaying additional information obtained by other sources, such as traces of program executions, revision histories, or even documents found on the Web may provide further hints towards the existence of vulnerabilities.

Extension to other forms of code. The presented approaches have been evaluated entirely on C/C++ code. While this allows many security critical code bases to be analyzed, it is currently unknown whether our approaches can be employed as-is for the analysis of code written in dynamic languages such as PHP or Javascript. Exploring this would allow our approaches to be applied in a much wider setting, making it possible to identify vulnerabilities in Web applications in particular. Finally, Pewny et al. [109, 110] provide evidence that our concept for vulnerability extrapolation is indeed also applicable

to binary code. Whether the remaining methods proposed in our work can be equally well applied in this setting is an open question.

Semi-supervised approaches for long-term settings. The machine learning approaches employed in our work are all unsupervised, that is, we assume that the analyst does not have time to provide any sorts of labels. This is a reasonable assumption when considering a third-party analyst task with finding many vulnerabilities in a program in a short amount of time. In contrast, for manufacturers who maintain software over a longer period of time, it may be possible to provide at least some labels. Employing semi-supervised approaches that allow this information to be taken into account may therefore be an interesting direction for future research.

Scaling to entire systems. Finally, while our evaluation considers large code bases such as the Linux kernel, we have not yet explored whether our techniques scale to entire OS distributions, that is, the source code of programs along with all libraries they employ. This is a particularly interesting question as programmers using a library typically do not possess an intimate knowledge of its implementation, and therefore, they may not be aware of those subtleties relevant for its secure operation.

A

Operations on Property Graphs

In the following, we define basic and intuitive operations on property graphs, which are made use of in the thesis. In particular, the construction of code property graphs requires an operation to merge property graphs to be defined. While for traditional graphs, this can be achieved easily by merging node sets and edge sets, for property graphs, we need to additionally take care of preserving edge labeling, and endpoint functions. To this end, we define the union of endpoint functions as follows.

Definition A.1. (*Union of endpoint functions.*) For $s_1 : E_1 \mapsto X_1$, $s_2 : E_2 \mapsto X_2$ and $E_1 \cap E_2 = \emptyset$, we define the union of s_1 and s_2 as $s = (s_1 \cup s_2) : (E_1 \cup E_2) \mapsto X_1 \cup X_2$ where $s(e) := s_1(e)$ if $e \in E_1$, and $s(e) := s_2(e)$ otherwise.

In this definition, we demand that the two node sets as well as the two edge sets are disjoint to ensure that the union of endpoint functions is actually a function as well. Analogously. we can define the union for attribute functions as follows.

Definition A.2. (*Union of attribute functions.*) For $\mu_1 : (V_1 \times K_1) \mapsto S_1$, $\mu_2 : (V_2 \times K_2) \mapsto S_2$, and $V_1 \cap V_2 = \emptyset$, we define the union of μ_1 and μ_2 as $\mu = (\mu_1 \cup \mu_2) : ((V_1 \cup V_2) \times (K_1 \cup K_2)) \mapsto (S_1 \cup S2)$ where for all $k \in K_1 \cup K_2$, $\mu(x, k) = \mu_1(x, k)$ if $x \in V_1$ and $\mu(x, k) = \mu_2(x, k)$ otherwise.

With these two definitions at hand, we can now define the union of property graphs.

Definition A.3. (*Union of property graphs.*) Let $g_1 = (V_1, E_1, \lambda_1, \mu_1, s_1, d_1)$ and $g_2 = (V_2, E_2, \lambda_2, \mu_2, s_2, d_2)$ be property graphs with $V_1 \cap V_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$, then the union $g_1 \cup g_2$ is obtained by element-wise application of the union operator.

The union of property graphs can be employed in particular to add nodes, edges, and labels to graphs. For removal, we define the difference of property graphs as follows.

Definition A.4. (*Difference of functions.*) Let $f : X_f \to Y_f$ and $g : X_g \to Y_g$ be two functions. Then we define the difference $f \setminus g$ to be given by the function corresponding to the set of pairs $\{(x, f(x)) : x \in X_f \text{ and } x \notin X_g\}$.

Definition A.5. (*Difference of property graphs.*) Let $g_1 = (V_1, E_1, \lambda_1, \mu_1, s_1, d_1)$ be a property graph and $g_2 = (V_2, E_2, \lambda_2, \mu_2, s_2, d_2)$ is a sub graph for $g1$, then the difference $g_1 \setminus g_2$ is obtained by element-wise application of the set-minus operator.

Closely related, we often need to restrict a property graph to a sub set of its nodes while preserving attributes and edges between these nodes. To this end, we define the restriction as follows.

Definition A.6. (*Restriction.*) Let $g = (V, E, \lambda, \mu, s, d)$ be a property graph and let $U \subseteq V$ be a subset of its nodes, and $D \subseteq E$ be a subset of its edges. Then the restriction $g|_{U,D}$ is defined as the property graph containing all nodes of U and all edges of D, that is, $g|_{U,D} = (U, D, \lambda|_D, \mu|_{(U \cup D}, s|_D, d|_D)$.

With the restriction at hand, we denote the restriction of a property graph to the single node x as $\mathcal{N}(x)$. Finally, to simplify notation, we define the *node set* $\mathcal{V}(g)$ and *edge set* $\mathcal{E}(g)$ of a property graph $g = (V, E, \lambda, \mu, s, d)$ to be given by V and E respectively.

B

Linux Kernel Vulnerabilities - 2012

Vulnerability types	Description	#
Memory Disclosure	A structure is copied to user space and not all fields or padding bytes are properly initialized resulting in memory disclosure.	21
Buffer Overflows	A length field involved in a copy operation is not checked resulting in buffer overflows.	16
Resource Leaks	A function creates a resource, but it is not destroyed on all error paths.	10
Design Errors	Program design does not sufficiently implement security policies.	10
Null Pointer Dereference	A pointer controlled by an attacker is dereferenced without checking whether it is null.	8
Missing Permission Checks	A security sensitive operation can be accessed without undergoing a prior permission check.	6
Race Conditions	Concurrently running processes cause various types of vulnerabilities.	6
Integer Overflows	A length field involved in a copy operation is checked, but the check is insufficient as integer overflows are not accounted for.	3
Division by Zero	An attacker-controlled value is a denominator in a division and it is allowed to be zero.	3
Use After Free	An allocated block of memory is used after being freed by the allocator.	3
Integer Type Vulnerabilities	A length field involved in a copy operation is checked, but the check is insufficient as the length field is a signed integer.	1
Insecure Arguments	Passing arguments to a function results in an implicit, insecure type cast.	1
Total vulnerabilities		88

TABLE B.1: Vulnerabilities discovered in the Linux kernel in 2012 sorted by vulnerability type [162].

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc.

[2] Allen, F. E. (1970). Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19.

[3] Alrabaee, S., Shirani, P., Wang, L., and Debbabi, M. (2015). Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71.

[4] Anderberg, M. (1973). *Cluster Analysis for Applications*. Academic Press, Inc., New York, NY, USA.

[5] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1.

[6] Anley, C., Heasman, J., Lindner, F., and Richarte, G. (2011). *The Shellcoder's Handbook: Discovering and exploiting security holes*. John Wiley & Sons.

[7] Appleby, A. (visited, March 2015). MurmurHash. `https://sites.google.com/site/murmurhash/`.

[8] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[9] Arp, D., Yamaguchi, F., and Rieck, K. (2015). Torben: A practical side-channel attack for deanonymizing tor communication. In *Proc. of ACM Symposium on Information, Computer and Communications Security(ASIACCS)*.

[10] Aycock, J. and Horspool, R. N. (2001). Schrödinger's token. *Software: Practice and Experience*, 31(8):803–814.

[11] Baier, C., Katoen, J.-P., et al. (2008). *Principles of model checking*, volume 26202649. MIT Press Cambridge.

[12] Batra, S. and Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512.

[13] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance (ICSM)*.

[14] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591.

[15] Berry, M. (1994). Computing the sparse singular value decomposition via svdpack. In *Recent Advances in Iterative Methods*, volume 60 of *The IMA Volumes in Mathematics and its Applications*, pages 13–29. Springer.

[16] Bishop, C. M. et al. (2006). *Pattern recognition and machine learning*, volume 4. Springer New York.

[17] Cadar, C., Dunbar, D., and Engler, D. R. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[18] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *Proc. of the International Conference on Software Engineering*.

[19] Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., and Greenstadt, R. (2015). De-anonymizing programmers via code stylometry. In *Proc. of USENIX Security Symposium*.

[20] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394.

[21] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3).

[22] Chang, R.-Y., Podgurski, A., and Yang, J. (2008). Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596.

[23] Chazelas, S. (visited June, 2015). Re: Stephane chazelas: How *did* you find shellshock? `http://seclists.org/oss-sec/2014/q4/224`.

[24] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

[25] Cooper, K. D., Harvey, T. J., and Kennedy, K. (2001). A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10.

[26] Csurka, G., Dance, C., Fan, L., Willamowski, J., and Bray, C. (2004). Visual categorization with bags of keypoints. In *Workshop on statistical learning in computer vision*.

[27] Cunningham, P. (2008). Dimension reduction. In *Machine learning techniques for multimedia*, pages 91–112. Springer.

[28] CVE (2006). CVE-2006-3459: Multiple Stack-Based Buffer Overflows in the TIFF library (libtiff). Common Vulnerabilities and Exposures, The MITRE Corporation.

[29] CVE-2014-0160 (2014). The Heartbleed Bug, `http://heartbleed.com/`.

[30] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*.

[31] DB, A. (visited May, 2015). Arangodb query language (aql). `https://docs.arangodb.com/Aql/`.

[32] Deerwester, S., Dumais, S., Furnas, G., Landauer, T., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

[33] DeKok, A. (visited February, 2013). Pscan: A limited problem scanner for c source files. `http://deployingradius.com/pscan/`.

[34] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

[35] Dowd, M., McDonald, J., and Schuh, J. (2006). *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education.

[36] Drake, J. J., Lanier, Z., Mulliner, C., Fora, P. O., Ridley, S. A., and Wicherski, G. (2014). *Android Hacker's Handbook*. John Wiley & Sons.

[37] Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification (2Nd Edition)*. John Wiley & Sons.

[38] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., et al. (2014). The matter of heartbleed. In *Proc. of the Conference on Internet Measurement Conference*.

[39] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001). Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72.

[40] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51.

[41] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349.

[42] Foundation, A. S. (visited June, 2015). Apache mahout. `http://mahout.apache.org/`.

[43] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., and Rieck, K. (2015). Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Proc. of the International Conference on Security and Privacy in Communication Networks (SECURECOMM)*.

[44] Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. (2013). Structural detection of android malware using embedded call graphs. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISEC)*, pages 45–54.

[45] Godefroid, P., Levin, M. Y., and Molnar, D. (2012). SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44.

[46] Goldsmith, S. F., O'Callahan, R., and Aiken, A. (2005). Relational queries over program traces. In *Proc. of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[47] Greenberg, A. (2015). Forbes: Shopping for zero-days: A price list for hackers' secret software exploits.

[48] Gruska, N., Wasylkowski, A., and Zeller, A. (2010). Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*.

[49] Gubichev, A. and Then, M. (2014). Graph pattern matching: Do we have to reinvent the wheel? In *Proceedings of Workshop on Graph Data management Experiences and Systems*.

[50] Hackett, B., Das, M., Wang, D., and Yang, Z. (2006). Modular checking for buffer overflows in the large. In *Proc. of the International Conference on Software engineering*.

[51] Hallem, S., Chelf, B., Xie, Y., and Engler, D. (2002). A system and language for building system-specific, static analyses. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*.

[52] Haller, I., Slowinska, A., Neugschwandtner, M., and Bos, H. (2013). Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proc. of USENIX Security Symposium*.

[53] Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301.

[54] Harmeling, S., Dornhege, G., Tax, D., Meinecke, F. C., and Müller, K.-R. (2006). From outliers to prototypes: ordering data. *Neurocomputing*, 69(13–15):1608–1618.

[55] Hartig, O. (2014). Reconciliation of rdf* and property graphs. *arXiv preprint arXiv:1409.3288*.

[56] Hartig, O. and Thompson, B. (2014). Foundations of an alternative approach to reification in rdf. *arXiv preprint arXiv:1406.3399*.

[57] Haykin, S. (2009). *Neural networks and learning machines*, volume 3. Pearson Education Upper Saddle River.

[58] Heelan, S. (2011). Vulnerability detection systems: Think cyborg, not robot. *IEEE Security & Privacy*, 9(3):74–77.

[59] Heintze, N. and Riecke, J. G. (1998). The slam calculus: Programming with secrecy and integrity. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*.

[60] Herman, D. (visited, March 2015). The c typedef parsing problem. `http://calculist.blogspot.de/2009/02/c-typedef-parsing-problem.html`.

[61] Hido, S. and Kashima, H. (2009). A linear-time graph kernel. In *Proc. of the IEEE International Conference on Data Mining (ICDM)*.

[62] Holzschuher, F. and Peinl, R. (2013). Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM.

[63] Horwitz, S., Reps, T., and Binkley, D. (1988). Interprocedural slicing using dependence graphs. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*, pages 35–46.

[64] Jain, A. K. and Dubes, R. C. (1988). *Algorithms for clustering data*. Prentice-Hall, Inc.

[65] Jang, J., Agrawal, A., , and Brumley, D. (2012). ReDeBug: finding unpatched code clones in entire os distributions. In *Proc. of IEEE Symposium on Security and Privacy*.

[66] Jaro, M. A. (1989). Advances in record linkage methodology as applied to the 1985 census of Tampa Florida. *Journal of the American Statistical Association*, 84(406):414–420.

[67] Joseph, A. D., Laskov, P., Roli, F., Tygar, J. D., and Nelson, B. (2012). Machine learning methods for computer security. *Dagstuhl Manifestos 3.1*.

[68] Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy*, pages 6–263.

[69] Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670.

[70] Karampatziakis, N., Thomas, A., Marinescu, M., and Stokes, J. W. (2012). Using file relationships in malware classification. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[71] Kinloch, D. A. and Munro, M. (1994). Understanding C programs using the combined C graph representation. In *Proc. of International Conference on Software Maintenance (ICSM)*.

[72] Klein, T. (2011). *A bug hunter's diary: a guided tour through the wilds of software security*. No Starch Press.

[73] Knapen, G., Lague, B., Dagenais, M., and Merlo, E. (1999). Parsing C++ despite missing declarations. In *Proc. of the International Workshop on Program Comprehension*.

[74] Kontogiannis, K. A., Demori, R., Merlo, E., Galler, M., and Bernstein, M. (1996). Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:108.

[75] Koppler, R. (1996). A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27:649.

[76] Koziol, J., Litchfield, D., Aitel, D., Anley, C., Eren, S., Mehta, N., and Hassell, R. (2004). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons.

[77] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D. (2006). From uncertainty to belief: Inferring the specification within. In *Proc. of the Symposium on Operating Systems Design and Implementation*.

[78] Krinke, J. (2004). Advanced slicing of sequential and concurrent programs. In *Proc. of the IEEE International Conference on Software Maintenance*.

[79] Krinke, J. and Snelting, G. (1998). Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11):661–675.

[80] Kruegel, C., Mutz, D., Valeur, F., and Vigna, G. (2003). On the detection of anomalous system call arguments. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, pages 326–343.

[81] Lam, M. S., Whaley, J., Livshits, V. B., Martin, M. C., Avots, D., Carbin, M., and Unkel, C. (2005). Context-sensitive program analysis as database queries. In *Proc. of Symposium on principles of database systems*.

[82] Larus, J. R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S. K., and Venkatapathy, R. (2004). Righting software. *IEEE Software*, 21(3):92–100.

[83] Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141.

[84] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192.

[85] Li, Z. and Zhou, Y. (2005). PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of European Software Engineering Conference (ESEC)*, pages 306–315.

[86] Livshits, B., Nori, A. V., Rajamani, S. K., and Banerjee, A. (2009). Merlin: specification inference for explicit information flow problems. In *Proc. of ACM SIGPLAN International Conference on Programming Languages Design and Implementation (PLDI)*.

[87] Livshits, B. and Zimmermann, T. (2005). Dynamine: finding common error patterns by mining software revision histories. In *Proc. of European Software Engineering Conference (ESEC)*, pages 296–305.

[88] Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *Proc. of USENIX Security Symposium*.

[89] Maier, A. (2015). Chucky-ng – a modular approach to missing check detection. Master's thesis, University of Goettingen.

[90] Maloof, M., editor (2005). *Machine Learning and Data Mining for Computer Security: Methods and Applications*. Springer.

[91] Manning, C. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

[92] Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge.

[93] Marcus, A. and Maletic, J. I. (2001). Identification of high-level concept clones in source code. In *Proc. of International Conference on Automated Software Engineering (ASE)*, page 107.

[94] Martin, M., Livshits, B., and Lam, M. S. (2005). Finding application errors and security flaws using pql: Program query language. In *Proc. of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[95] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320.

[96] Microsoft (visited July, 2015). Data execution prevention (dep). `https://support.microsoft.com/en-us/kb/875352`.

[97] Miller, C., Blazakis, D., DaiZovi, D., Esser, S., Iozzo, V., and Weinmann, R.-P. (2012). *iOS Hacker's Handbook*. John Wiley & Sons.

[98] Moonen, L. (2001). Generating robust parsers using island grammars. In *Proc. of Working Conference on Reverse Engineering (WCRE)*.

[99] Mozilla (visited, July 2015). Security advisories for firefox. `https://www.mozilla.org/security/known-vulnerabilities/firefox/`.

[100] Myers, A. C. (1999). Jflow: Practical mostly-static information flow control. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*.

[101] Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. (2001). Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif*.

[102] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. (2007). Predicting vulnerable software components. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

[103] Newsome, J., Karp, B., and Song, D. (2005). Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of IEEE Symposium on Security and Privacy*, pages 120–132.

[104] Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[105] Organization, T. M. (2014). Cve-2014-6271. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`.

[106] Parr, T. (2013). *The definitive ANTLR 4 Reference*. Pragmatic Bookshelf.

[107] Paul, S. and Prakash, A. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*.

[108] Perl, H., Arp, D., Dechand, S., Yamaguchi, F., Fahl, S., Acar, Y., Rieck, K., and Smith, M. (2015). Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

[109] Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. (2015). Cross-architecture bug search in binary executables. In *Proc. of IEEE Symposium on Security and Privacy*.

[110] Pewny, J., Schuster, F., Rossow, C., Bernhard, L., and Holz, T. (2014). Leveraging semantic signatures for bug search in binary programs. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*.

[111] Prosser, R. T. (1959). Applications of boolean matrices to the analysis of flow diagrams. In *Eastern Joint Computer Conference*.

[112] rats (visited April, 2012). Rough auditing tool for security. Fortify Software Inc., `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`.

[113] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. (2014). Optimizing seed selection for fuzzing. In *Proc. of USENIX Security Symposium*.

[114] Reps, T. (1998). Program analysis via graph reachability. *Information and Software Technology*.

[115] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366.

[116] Rieck, K. (2009). *Machine Learning for Application-Layer Intrusion Detection*. Doctoral thesis, Berlin Institute of Technology (TU Berlin).

[117] Rieck, K., Holz, T., Willems, C., Düssel, P., and Laskov, P. (2008). Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 108–125.

[118] Rieck, K., Krueger, T., Brefeld, U., and Müller, K.-R. (2010a). Approximate tree kernels. *Journal of Machine Learning Research (JMLR)*, 11(Feb):555–580.

[119] Rieck, K., Krueger, T., and Dewald, A. (2010b). Cujo: Efficient detection and prevention of drive-by-download attacks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 31–39.

[120] Rieck, K. and Laskov, P. (2008). Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9(Jan):23–48.

[121] Risky.Biz (visited July, 2015). Risky business 339 - neel mehta on heartbleed, shellshock. `http://risky.biz/RB339`.

[122] Rivest, R. L. (visited April, 2015). S-expressions. `http://people.csail.mit.edu/rivest/Sexp.txt`.

[123] Robinson, I., Webber, J., and Eifrem, E. (2013). *Graph databases*. "O'Reilly Media, Inc.".

[124] Rodriguez, M. A. and Neubauer, P. (2011). The graph traversal pattern. *Graph Data Management: Techniques and Applications*.

[125] Roweis, S. and Saul, L. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

[126] Sabelfeld, A. and Myers, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19.

[127] Salton, G. and McGill, M. J. (1986). *Introduction to Modern Information Retrieval*. McGraw-Hill.

[128] Salton, G., Wong, A., and Yang, C. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620.

[129] Sassaman, L., Patterson, M. L., Bratus, S., and Shubina, A. (2011). The halting problems of network stack insecurity. *login:, December*.

[130] Scandariato, R., Walden, J., Hovsepyan, A., and Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006.

[131] Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319.

[132] Schwartz, E., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE Symposium on Security and Privacy*, pages 317–331.

[133] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. (2001a). Detecting format string vulnerabilities with type qualifiers. In *Proc. of USENIX Security Symposium*, pages 201–218.

[134] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. (2001b). Detecting format string vulnerabilities with type qualifiers. In *Proc. of USENIX Security Symposium*.

[135] Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., and Vishwanathan, S. (2009). Hash kernels for structured data. *Journal of Machine Learning Research (JMLR)*, 10(Nov):2615–2637.

[136] Shirey, R. (2007). Internet security glossary, version 2, fyi 36, rfc 4949.

[137] Standard, E. S. S. (visited, March 2015). Ebnf: Iso/iec 14977: 1996 (e). `http://www.cl.cam.ac.uk/mgk25/iso-14977.pdf`.

[138] Stuttard, D. and Pinto, M. (2008). *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons.

[139] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[140] Symantec (2015). Internet security threat report. Volume 20 (April 2015), Symantec Corporation.

[141] Synytskyy, N., Cordy, J. R., and Dean, T. R. (2003). Robust multilingual parsing using island grammars. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research*.

[142] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. (2008). Autoises: automatically inferring security specifications and detecting violations. In *Proc. of USENIX Security Symposium*.

[143] Team, P. (visited July, 2015). Address space layout randomization (aslr). `http://pax.grsecurity.net/docs/aslr.txt`.

[144] Technology, N. (visited May, 2015). Cypher query language. `http://neo4j.com/docs/stable/cypher-query-lang.html`.

[145] Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

[146] Thummalapenta, S. and Xie, T. (2009). Alattin: Mining alternative patterns for detecting neglected conditions. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, pages 283–294.

[147] TinkerPop (visited March, 2015). Blueprints. `http://blueprints.tinkerpop.com`.

[148] Vanegue, J., Heelan, S., and Rolles, R. (2012). Smt solvers in software security. In *Proc. of USENIX Workshop on Offensive Technologies (WOOT)*.

[149] Vanegue, J. and Lahiri, S. K. (2013). Towards practical reactive security audit using extended static checkers. In *Proc. of IEEE Symposium on Security and Privacy*.

[150] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. In *Proc. of the annual Southeast regional conference*.

[151] Viega, J., Bloch, J., Kohno, Y., and McGraw, G. (2000). ITS4: A static vulnerability scanner for C and C++ code. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, pages 257–267.

[152] (W3C), W. W. W. C. (visited March, 2015a). Resource description framework (rdf). `http://www.w3.org/RDF/`.

[153] (W3C), W. W. W. C. (visited March, 2015b). Sparql 1.1 overview. `http://www.w3.org/TR/sparql11-overview/`.

[154] Wang, K. and Stolfo, S. (2004). Anomalous payload-based network intrusion detection. In *Recent Adances in Intrusion Detection (RAID)*, pages 203–222.

[155] Wang, T., Wei, T., Lin, Z., and Zou, W. (2009). IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[156] Wasylkowski, A., Zeller, A., and Lindig, C. (2007). Detecting object usage anomalies. In *Proc. of European Software Engineering Conference (ESEC)*, pages 35–44.

[157] Weiser, M. (1981). Program slicing. In *Proc. of International Conference on Software Engineering*.

[158] Welc, A., Raman, R., Wu, Z., Hong, S., Chafi, H., and Banerjee, J. (2013). Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*.

[159] Wheeler, D. A. (visited April, 2012). Flawfinder. `http://www.dwheeler.com/flawfinder/`.

[160] Williams, C. C. and Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31:466–480.

[161] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. (2015). No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Proc. of Network and Distributed System Security Symposium (NDSS)*.

[162] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[163] Yamaguchi, F., Lindner, F., and Rieck, K. (2011). Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[164] Yamaguchi, F., Lottmann, M., and Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*.

[165] Yamaguchi, F., Maier, A., Gascon, H., and Rieck, K. (2015). Automatic inference of search patterns for taint-style vulnerabilities. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[166] Yamaguchi, F., Wressnegger, C., Gascon, H., and Rieck, K. (2013). Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*.

[167] Yudkowsky, E. (2008). Artificial intelligence as a positive and negative factor in global risk. *Global catastrophic risks*, 1:303.

[168] Zalewski, M. (visited June, 2015). Symbolic execution in vuln research. `http://lcamtuf.blogspot.de/2015/02/symbolic-execution-in-vuln-research.html`.

[169] Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending API usage patterns. In *Proc. of the European Conference on Object-Oriented Programming(ECOOP)*, pages 318–343.

[170] Zimmermann, T., Nagappan, N., and Williams, L. (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. In *International Conference on Software Testing, Verification and Validation (ICST)*.