CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Cross-Language Dependency Analysis for VS Code Extension Ecosystem

Master's Thesis in Computer science and engineering

Alexander Brunnegård, Malte Carlstedt

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Cross-Language Dependency Analysis for VS Code Extension Ecosystem

Alexander Brunnegård, Malte Carlstedt

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Cross-Language Dependency Analysis for VS Code Extension Ecosystem
Alexander Brunnegård, Malte Carlstedt

Supervisor:
Mohannad Alhanahnah, Department of Computer Science and Engineering
Examiner:
Jennifer Horkoff, Department of Computer Science and Engineering
Examiner in practice:
Yi Peng, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2025

Cross-Language Dependency Analysis for VS Code Extension Ecosystem

Alexander Brunnegård, Malte Carlstedt
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Visual Studio Code (VS Code) is currently the most popular integrated development environment (IDE), primarily due to its highly modular architecture facilitated by third-party extensions. These extensions can rely on dependencies spanning multiple programming languages, notably JavaScript and native languages such as C and C++. Such cross-language interactions introduce complexity and potential security vulnerabilities due to differences in memory management, type safety, and crash resilience between languages. While previous research has identified the inherent security risks in cross-language bindings within individual packages in the `npm` ecosystem, the implications of such vulnerabilities within the VS Code extension ecosystem have yet to be explored.
This thesis investigates cross-language dependencies in VS Code extensions, specifically focusing on the interactions between JavaScript and native code. A methodology is presented to systematically discover, construct, and analyse the dependency tree from an extension to native code. The study uncovers patterns, characteristics, and potential security risks associated with native dependencies in VS Code extensions. This research provides insights into the lack of security practices within the VS Code ecosystem by addressing the gap between current knowledge about cross-language vulnerabilities and VS Code extensions. The results show that 455 (14.7%) out of the investigated 3,078 extensions either implemented native code directly or depend on a package including cross-language cooperation. While only two extensions had direct production code in a native language, they amassed 171 potential vulnerabilities. Additionally, 211 extensions depended on 228 dependencies containing native code that amassed 8,732 potential vulnerabilities in total, showing the potential risks of using such packages.

Keywords: Cross-Language Dependency, VS Code Extensions, Vulnerability Analysis, npm, CodeQL, Static Analysis

Acknowledgements

We would like to thank our supervisor, Mohannad Alhanahnah, for his continuous input, feedback, insightful discussions, and help.

We acknowledge the use of AI tools, including OpenAI's ChatGPT and Anthropic's Claude, for assistance with writing and code debugging. All outputs were critically reviewed to ensure accuracy and academic integrity.

Alexander Brunnegård, Malte Carlstedt, Gothenburg, June 2025

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1

Introduction

Modern programming languages like JavaScript and Python allow developers to interact with or import libraries written in lower-level languages, such as C or C++. This is often achieved through Foreign Function Interfaces (FFIs), enabling high-level scripts to call low-level code directly. While the capabilities and benefits of low-level languages can be utilised, they can also introduce severe security bugs and vulnerabilities due to the vast possibilities and unchecked nature of C and C++. Grichi et al. [27] found that these capabilities increase the risk of bugs threefold, while security vulnerabilities are doubled. These risks have previously been studied in the `npm` ecosystem [47] and browser extensions [14], but not within the context of Visual Studio Code (VS Code) extensions. This domain presents new angles of attack and challenges, ranging from direct access to the network, kernel intercommunication, and device use with the same permissions as the user opening VS Code [1]. At the same time, cross-language interactions put additional emphasis on the knowledge of extension and package maintainers, requiring expertise in multiple languages and their bindings to catch potentially vulnerable or buggy code.

The rest of this chapter states the problem that was investigated (section 1.1), give the purpose of the study (section 1.2), and outline the remaining structure of the thesis (section 1.3).

1.1 Problem Description

According to Stack Overflow's 2024 survey of over 65,000 participants [6], VS Code is the most widely used integrated development environment (IDE), with more than 73% adoption. Its minimal core architecture and modularity allow users to integrate with different languages, tools and customisations through extensions. This approach by the creators enables the application to be effectively used for all languages, whereas other IDEs usually differ depending on the language used. Hence, a developer can use the same application and ecosystem for numerous languages across different projects, though at the expense of relying on third-party extensions to achieve this. Due to the lack of proper vetting and verification of an extension, many extensions with malicious intent or vulnerabilities are available for download on the VS Code marketplace [23], [35], [29].

Previous works have studied security vulnerabilities in VS Code [30], its extensions [23], [35], [29]. and cross-language analysis within other domains [47], [44], [15], [22], but none have explored cross-language vulnerabilities in VS Code extensions. This study shows potential cross-language vulnerabilities in extensions, which can propagate to VS Code and the host system. These distinctions make our research particularly relevant for understanding security risks in development environments, where developers routinely install third-party extensions with no privilege restrictions compared to the user. In addition to exploiting developers who download these extensions, corporations can have their secrets, source code and potentially credentials stolen by the extension developer, due to the open nature of the VS Code architecture, and lack of tightened permissions. Extensions do not explicitly have to state their needed permissions nor are run in the confines of a sandbox, but rather acquire the same permissions as VS Code [1], enabling communication to both the Node runtime and with the underlying host system via native FFI modules.

To illustrate an occurrence of a cross-language inconsistency, Listing 1.1 presents the `divide` function from the `int64-napi` package, where the crash safety of JavaScript is compromised, as shown initially by Staicu et al. [47]. In contrast to dividing by zero in JavaScript, where `infinity` is returned, the program crashes in the Node.js runtime, never reaching the catch clause as expected.

```
1 const int64 = require("int64-napi");
2
3 try {
4     int64.divide(10, 0); // hard crash of Node.js
5 } catch(e) { } // never invoked
```

Listing 1.1: Motivating example of a hard crash using FFI

1.2 Purpose of the Study

This study aims to analyse cross-language dependencies in VS Code extensions, focusing on their security implications and unforeseen side effects. By examining extensions directly from the VS Code marketplace, we aim to identify common patterns and potential vulnerabilities that arise from interactions between JavaScript and native C/C++ modules in real-life scenarios. These dependencies were statically analysed to find potential security risks and taint sinks.

For this thesis, we state two assumptions: **(I)** The workspace is not restricted. Restricting a workspace limits the source code within the workspace and the extensions used. Although it is suggested to restrict foreign code and, therefore, automatic code execution from tasks, extensions, or workspace settings, this renders VS Code nearly useless. When working with one's code and installing an extension, the workspace is already trusted; thus, this assumption is often implied. **(II)** For any VS Code version $\geq$1.97, the installed extension (and all dependent extensions) are also assumed to be trusted, as they otherwise cannot function in VS Code. Though necessary for this study, these assumptions are rarely broken in a real-life scenario, as the entire installation of said extension relies on the user trusting it.

The suggested threat model assumes that the developer is not malicious, but that buggy and incorrect code can occur and contain vulnerabilities. We do not focus on exploiting malicious use, but rather identifying potential misuses in the ecosystem and spreading awareness of the core problem.

A single benchmark has been created to validate the creation of the dependency tree structure, and an additional eight benchmarks have been created to validate the static analysis tool and possible vulnerabilities within the dependency tree. From the proposed methodology, the research questions sought to be answered in this study are the following:

RQ1 Do VS Code extensions contain cross-language dependencies, and what are their characteristics?

RQ2 Do cross-language dependencies introduce security vulnerabilities in VS Code extensions, and how prevalent are their occurrences?

1.3 Thesis Outline

Chapter 2 introduces foundational background topics that help understand the rest of the thesis.

Chapter 3 presents previous works on relevant topics and identifies the insights gathered from them. This thesis will cover a subject that spans several specific fields and thus includes a broad range of previous work.

Chapter 4 outlines the methodology for collecting, analysing, and presenting cross-language dependencies in VS Code extensions and their dependents using a pipeline. It also details the implementation of tools from Chapter 2 for this specific use case.

Chapter 5 presents the objective results that were gathered from the investigation.

Chapter 6 introduces points for discussion, such as threats to validity, assumptions and their implications, and potential factors that might have affected the outcome.

Chapter 7 concludes from the results gathered in chapter 5.

2

Background

As mentioned previously, this thesis investigates cross-language interactions within VS Code extensions and examines their associated security implications. Accordingly, this chapter provides the necessary background information on relevant subjects, introducing all the necessary elements for understanding the remainder of the thesis. All non-trivial tools used in Chapter 4 will be introduced, along with underlying theoretical backgrounds.

2.1 IDEs and VS Code

Integrated development environments, or IDEs for short, are widely used among developers to increase their productivity by combining and packaging a variety of tools in a single application [7], in contrast to traditional text editors. Tools for syntax highlighting, testing, debugging, building, and packaging software are examples that generally can be included. Visual Studio, IntelliJ, WebStorm, and PyCharm are examples of IDEs that include these capabilities. IDEs are often defined for particular languages (e.g. Visual Studio for the .NET framework, PyCharm for Python, and IntelliJ for Java), and whilst they do work for other languages than intended, specific capabilities are not translated. JetBrains has created several IDEs, such as IntelliJ for Java, PyCharm for Python, Webstorm for web development, and GoLand for Go. While these IDEs share similar interfaces, they differ in language-specific features, such as in-application building and out-of-the-box syntax highlighting.

While Visual Studio Code (VS Code) shares certain features with full-fledged IDEs like Visual Studio, it is more accurately described as a lightweight, extensible code editor. By default, VS Code offers built-in support for JavaScript, TypeScript, HTML, and CSS, including syntax highlighting and IntelliSense. Additional support for other languages and advanced features, such as debugging and building, requires installing extensions from the VS Code Marketplace or using external tools. This delimitation was made to create a one-serves-all code editor, stating that "most VS Code users will need to install additional components depending on their specific needs" [16]. This modular approach allows developers to customise their environment by adding extensions for version control, additional language support, interface customisation, and AI-assisted coding. While many of these extensions are essential for a comprehensive development experience, users should install third-party

extensions cautiously due to potential security risks. However, as the application's functionality largely depends on the use of extensions and the broad encouragement to download them in their documentation [17], it's easier said than done.

VS Code is an application built on Electron[1], which is a framework that includes both Chromium[2] and Node.js[3] functionality [3], and is used for making cross-platform desktop applications. Electron provides an environment by enabling web technologies (HTML, CSS, and JavaScript) to be used for application development, bridging the gap between web and desktop software.

Electron applications are prone to security vulnerabilities, as highlighted by Jin et al. [30]. Their study identifies risks emerging from unintended DOM mutations (e.g., via malformed HTML/CSS inputs), which can lead to script injection, privacy leaks, or UI deception, even without direct Node.js integration. For example, improperly sanitised user inputs in apps like Microsoft Teams or Visual Studio Code allow attackers to inject malicious scripts or exfiltrate data through crafted DOM elements.

2.2 Software Dependencies and Package Managers

In all dependency systems, dependencies come in two flavours: *direct* or *transitive*. A program directly uses its direct dependencies, whilst transitive dependencies refer to libraries on which a library may depend. For example, if program **P** depends on library **L**, and library **L** inherently depends on another library, **K**, **P** *transitively* depends on **K**. This means that when a package is imported, the program also depends on every imported package used within it. Thompson stated as early as 2003 that insecurities and failures in software dependencies, such as packages, contribute to the complicated nature of testing software [48]. As additionally found by Zimmermann et al. [51] on the `npm` ecosystem, trusting a single package implicitly involves trusting an average of 79 transitive dependencies. In addition to their unveiling that 40% of all packages depend on a package having at least one known vulnerability, this cascading transitive dependency tree causes vulnerability propagation through the program unbeknownst to the developer.

Package managers enable developers to import functionality already written to accommodate specific purposes, encouraging module reuse. In most cases, the choice of package manager is not enforced, giving developers the freedom to use their preferred tools. For example, Java developers often use `Maven`, while C and C++ developers may prefer `vcpkg`. On the other hand, Go uses a built-in package manager maintained by its creators.

For the Node.js ecosystem, several package managers are available, each with their benefits and disadvantages. According to themselves, the *node package manager* (`npm`) is the "world's largest software registry", allowing users to create, share and

[1]Link to Electron
[2]Link to Chromium
[3]Link to Node.js

browse Node packages. It is the default package manager and is included when installing Node. Whilst other package managers, such as `yarn`[4] and `pnpm`[5], can be installed and used if preferred; they are not included by default. `yarn` proposes improvements to resource usage and security, but its repository is relatively small in comparison [46]. *Performance* `npm`, or `pnpm`, uses the `npm` registry and also boasts significant efficiency in both storage and installation time, as compared to its predecessor [10]. A commonality for both these alternatives to `npm` is their relatively small communities in comparison, making the community support less extensive.

2.3 `Node.js` and Package Managers

While Node.js and its package managers are closely correlated, essential distinctions must be stated for future understanding when reading this study. Additionally, there are multiple choices and factors for deciding on a package manager for Node.js, which will be discussed below.

2.3.1 `Node.js`

Node.js is, according to its creator Ryan Dahl, a "server-side runtime platform, built on Google's V8[6] [engine]", and is largely written in C and C++ [31]. It runs JavaScript code outside the web browser, thus allowing developers both server-side and client-side JavaScript, sparking the "JavaScript everywhere" concept. According to several blog posts [9], [33], [2], JavaScript is used for over 97% of all websites. Allowing it to also run as the backend, which is not inherently possible with plain JavaScript, lowers the barrier of entry to backend development. With Node.js, a developer previously only accustomed to JavaScript can start developing a backend without learning a new language, which is especially impactful for inexperienced developers.

Recent competitors to Node.js have emerged over the last few years, including Bun[7] and Deno[8]. These alternatives provide certain benefits over the much older architecture of Node.js, ranging from build times for Bun to the increased security for Deno [5]. Although these alternatives are not addressed in more detail than this brief introduction in this thesis, the shortcomings of Node.js have led to their creation and recent boost in use.

2.3.2 Packages in Node.js

A Node.js package can extend the functionality of an application by importing it into the application, encouraging the reuse of the modules. This way, the developer

[4]Link to the `yarn` package manager
[5]Link to the `pnpm` package manager
[6]Link to the V8 Engine
[7]Link to Bun
[8]Link to Deno

can reuse already implemented functionality by importing it as pre-built packages, instead of building it from the ground up.

2.3.3 Semantic Versioning

Semantic versioning, or `semver`[9], is the used versioning in the `npm` ecosystem. Instead of declaring a distinct dependency version, `semver` allows version windows, where the versions are compliant. In Listing 2.1, some common `semver` version ranges are given, but for the complete listing explanation, see the `node-semver` GitHub repository [11].

```
1  * := >=0.0.0 // Any non-prerelease version
2
3  1.x := >=1.0.0 <2.0.0-0 // Matching on major version
4
5  1.2.3 - 2.3.4 := >=1.2.3 <=2.3.4 // Inclusive range matching
       on
6  1.2.3 - 2 := >=1.2.3 <3.0.0-0
7
8  ~1.2.3 := >=1.2.3 <1.(2+1).0 := >=1.2.3 <1.3.0-0 //
9
10 ^1.2.3 := >=1.2.3 <2.0.0-0 // Matching on major version
11 ^0.2.3 := >=0.2.3 <0.3.0-0 // Matching on minor version
```

Listing 2.1: Common Selection of Semver Range Definitions

2.3.4 Anatomy of `package.json`

A Node.js package is structurally described by its metadata file, `package.json`, where the package's name, versioning and other relevant metadata are included. While the full description of the structure of `package.json` can be found in the `npm` documentation [41], some fields in the metadata are important for this study and will thus be covered. Other than the mandatory *name* and *version* fields for publishing a package, the metadata also defines the entry point(s) to the package, its dependencies, and the repository link. However, additional historical data tied to specific versioning can also be fetched from the `npm` registry.

The *entry points* can be described in two separate ways, either using *main* or *exports* [39]. The *main* field declares the single entry point to the package, like the *main* function in C, C++ or Java, and has been supported from the inception of Node. Meanwhile, the *exports* field is a newer mechanism that can specify multiple entry points, which allows for better decoupling of separate modules within a package. Depending on the version of Node in which the package is used, it may need the declaration of *main*, as older versions do not recognise the *exports* field. However, for all newer versions, *exports* is preferred [39]. They use the same prerogative,

[9]Link to semver website

targeting a JavaScript export file to serve as the package interface(s), exposing all reachable functions and variables within the package.

Package dependencies come in multiple types, depending on where and how they are needed. Examples include *dependencies*, *devDependencies*, *peerDependencies*, *bundleDependencies*, and *optionalDependencies*, though the first-mentioned is the most commonly used. While *dependencies* target all dependencies needed for using a package in your code, *devDependencies* target those required for the development phase of that particular package, e.g., transpilers, unit testing packages, or minification. The others, namely *peerDependencies*, *bundleDependencies*, and *optionalDependencies*, are far less straightforward. The nature and use of *peerDependencies* are to enable bundling versions (See Listing 2.2), such that if the application **A** uses dependency **D** with version **V**, whilst another dependency, $\hat{\mathbf{D}}$, that is used in application **A** uses dependency **D** with version $\hat{\mathbf{V}} \neq \mathbf{V}$, then if there is any `semver` version that covers both cases, this can be used as a *peerDependency* [21]. In Listing 2.2, two dependencies on different versions of the `lodash` package are used for direct use in application **A** and Dependency $\hat{\mathbf{D}}$. However, Listing 2.3 shows the resolved version for both instances of the `lodash` package.

```
1 Application A
2 ├── lodash @ ^4.17.0
3 └── Dependency D̂
4     └── lodash @ ^4.15.0
```

Listing 2.2: Conflicting Dependency Versions

```
1 Application A
2 ├── lodash @ 4.17.21
3 └── Dependency D̂
4     └── lodash @ 4.17.21
```

Listing 2.3: Resolved `semver` versions for both uses of `lodash`

The *bundleDependencies* field declares any bundled packages that can be extracted from the Node package (see the example in the official documentation [41]). If a dependency does not fetch or install, but the code can be run either way, this can be done with *optionalDependencies*. However, this means that there should be multiple ways for the code to work; one with the package installed and one without. For example, a package for "pretty printing" an exception could be used as an *optionalDependency*. If the package can be fetched, pretty print the exception; otherwise, print it without the formatting. In summary, all cases of dependencies have their own set of benefits depending on the situation.

2.4 Cross-Language Interaction

Cross-language interaction offers advantages from one language to another. Most commonly, high-level languages like Python, JavaScript, or Java can utilise the capabilities of a low-level language, such as C or C++ (known as native extensions), for several use cases. For instance, cross-language interaction can allow JavaScript to incorporate modules like `bcrypt` and `sqlite` without requiring reimplementation, allowing low-level efficiency and optimisations. Additionally, native extensions can use the performance benefits of the other language, improving the efficiency of low-

level languages for compute-heavy tasks, as shown by Lion et al. [36], where the JavaScript counterpart of a C++ application runs over 8 times slower. Furthermore, modules that interact with the system can be accessed from higher-level languages, which may otherwise be unreachable.

2.4.1 Foreign Function Interfaces

There are various ways to establish interaction across languages. Foreign Function Interfaces (FFI) are one common way to enable cross-language interaction and binding APIs. An FFI is a mechanism that allows the loading of code compiled in a foreign language into a program. The Java Native Interface (JNI) is an example in which modules written in native code can be embedded in a Java program and used to extend the functionality of the high-level Java language. The `Win32RegKey` application illustrated by Horstmann [19], where the Windows registry is not accessible directly in Java. The Windows registry is used via a native C implementation and extracted using the JNI, an FFI between Java and C/C++. This approach proves useful in certain situations, but as identified by Horstmann previously in that same chapter, care must be taken on implementation. The native language has no protection against overwriting memory through invalid pointer usage, unlike Java applications [19], further claiming that it is easy to write native methods that corrupt your program or infect the operating system, as aligned by Zimmermann et al. [51].

2.4.2 FFI Security

Although cross-language interactions bring many benefits as discussed earlier, Grichi et al. [27] show that bugs within interlanguage dependencies are three times more common than in intralanguage (i.e., single language) dependencies, and vulnerabilities are twice as common. This implies the issue-prone nature of such dependencies and should deter the developer from using them excessively.

Brown et al. [15] show that the underlying engine for the binding can bypass the crash, type, and memory safety of JavaScript, resulting in unexpected behaviour. Staicu et al. [47] further demonstrated that this could propagate by reads of uninitialised memory, hard crashes in their applications, or memory leaks. This implies that issues within the FFI can break the guarantees of the scripting language and can thus lead to uncertain behaviour in an application. Opposed to both mentioned, where they focused on cross-language dependencies in `npm` packages, this study will explore cross-language dependencies for VS Code extensions, presenting unique challenges compared to `npm` packages. Unlike a stand-alone `npm` package, VS Code extensions function within the host environment (see Figure 2.1), interacting with different APIs, user inputs, and system-level functionalities. This introduces additional complexity in dependency management, as extensions may invoke native modules for debugging, file system operations, and language processing tasks. As previously mentioned, VS Code extensions are a core part of VS Code's modular approach, placing much of the application's capabilities in the hands of third-party extension developers.

2.4.3 FFIs in Node.js

Node.js offers multiple ways for developing native addons, but the two currently most prevalent are using Node-API or the C++ wrapper library, `node-addon-api`. These are typically built using the `node-gyp` build tool, maintained by Node.js. The Node-API is an API for developing native C modules in Node applications, maintained as a part of Node.js [40]. In contrast to its predecessors, NAN and direct V8 APIs, Node-API guarantees *application binary interface* (ABI) stability and is maintained separately from the underlying engine, allowing compiled native modules to work across Node.js versions without recompilation. This ensures that any update to the underlying JavaScript engine does not impact the compatibility of a program and its native Node-API module, addressing a key disadvantage to previous methods in the Node.js ecosystem: the fragility of native addons tied to engine-specific implementations. By abstracting the underlying JavaScript engine, Node-API promotes cross-version compatibility, in addition to future-proofing native modules against changes in the JavaScript engine itself. In contrast, NAN and the V8 APIs are tightly coupled with the underlying engine; thus, engine updates can break implementations.

The Node-API toolkit equips developers with a robust set of tools for creating native extensions in Node.js, offering a comprehensive collection of functions to interact with JavaScript values and objects. This includes capabilities for creating and manipulating JavaScript data types, invoking functions, and handling errors through a consistent C-based interface. It additionally enables developers to focus on implementing functionality rather than managing intricate interoperability details.

On the other hand, the C++ wrapper version offers the use of C++ abstractions, exception handling and object-oriented features to allow a simplified development process, and it is also maintained as part of Node.js. According to the Node-API documentation, the `node-addon-api` "is more efficient for writing code that calls Node-API" [40], emphasising its efficiency. The implementations made using the C++ wrapper make use of the underlying ABI, but package it in a more comprehensive and efficient way, at the expense of a low overhead.

2.5 Extensions in VS Code

Extensions are available for IDEs, such as VS Code and IntelliJ, and browsers, such as Google Chrome and Mozilla Firefox. Although not used in the same context, they resemble each other greatly; both are written mainly in JavaScript, gain access to an application-specific API, and can manipulate the DOM, but there are also differences between them. In contrast to VS Code extensions, browser extensions do not get direct access to the host system or the Node runtime and are sandboxed within the confines of their respective applications.

An extension in VS Code follows the underlying structure of the Electron appli-

cation, running in the Node runtime. It also has access to the VS Code API[10], enabling manipulation of the application and the host.

The architecture of VS Code is visualised in Figure 2.1. Extensions in VS Code are run in a non-sandboxed environment, called the extension host, and are run with the same privileges as VS Code itself [1], allowing significant manipulation of the host system. This means that an extension can be used as an attack vector to infiltrate the host to leak sensitive data or inject code to trigger supply chain poisoning attacks. While VS Code extensions traditionally communicate with the underlying runtimes and the operating systems via predefined Node APIs, such as the `os`, `fs` and `worker_threads` packages, custom interactions are accessible using FFIs, displayed in purple in Figure 2.1.

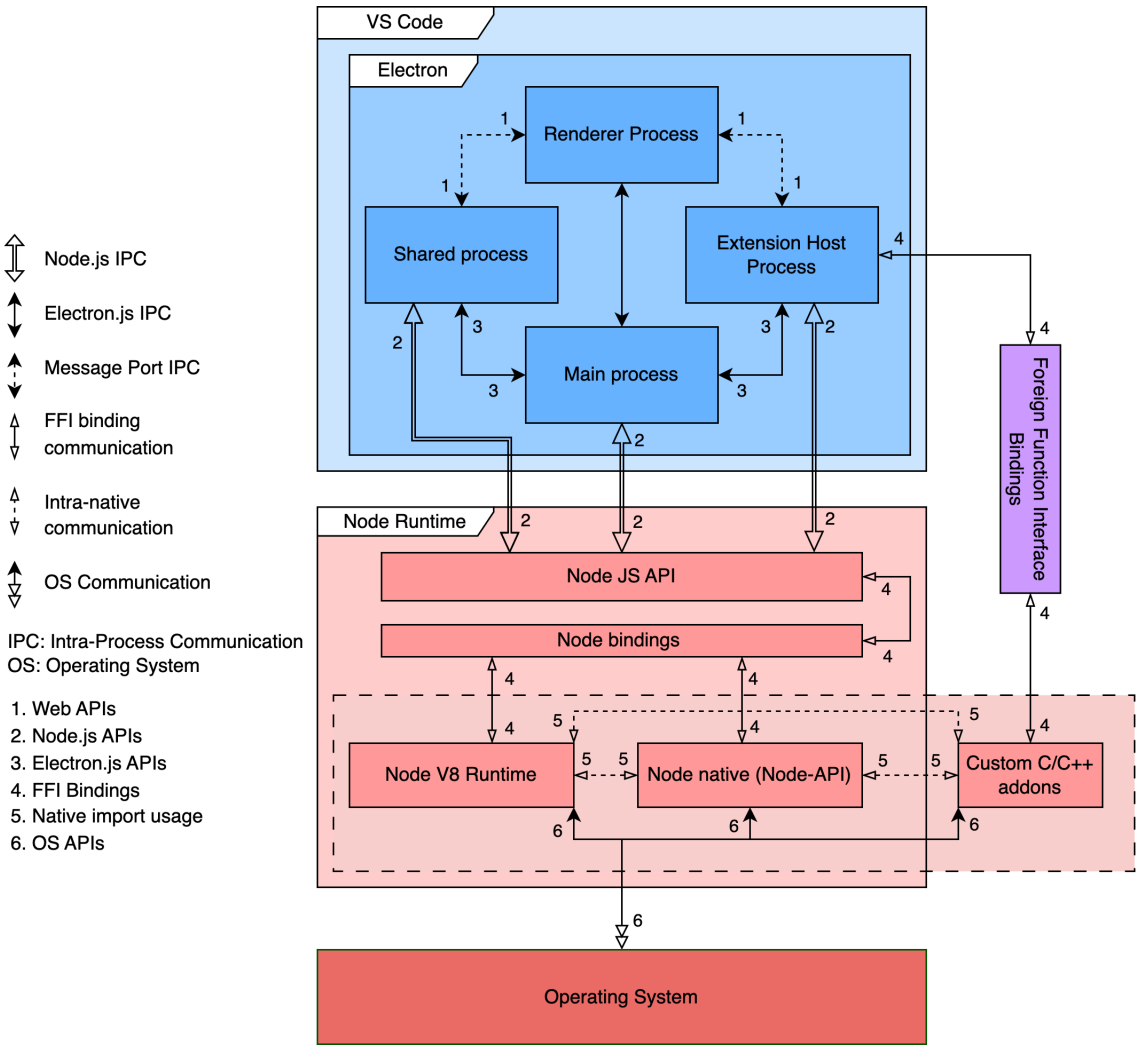

Figure 2.1: Architectural structure of VS Code, including the binding between JS and the native modules, heavily inspired by the visualisation from Edirimannage et al. [23].

[10]Link to the VS Code API

As shown in Assaraf's blog post [12], the VS Code marketplace can be manipulated to promote unvetted extensions, potentially exposing its users to malicious intents from adversaries. They created a proof-of-concept extension that exposes the code of opened files to a remote server, masquing itself as a popular VS Code theme. Unknowing victims downloaded the extension simply by misspelling the name of the intended theme extension, thus possibly exposing valuable company data to the extension's creator.

Recent updates (VS Code versions $\geq$19.7) to the security model of *extension publisher trust* states that the application shows a dialog when installing an extension, requiring manual confirmation of trusting the publisher and any dependent extensions (shown in Figure 2.2). This is an attempt to give the user the knowledge of possible threats within third-party extensions, yet merely adds another step in the process of installation. One can argue that this does not combat the core problem of malicious or vulnerable extensions but only shifts the responsibility toward the end-user, relieving all possible liability from the maintainers.

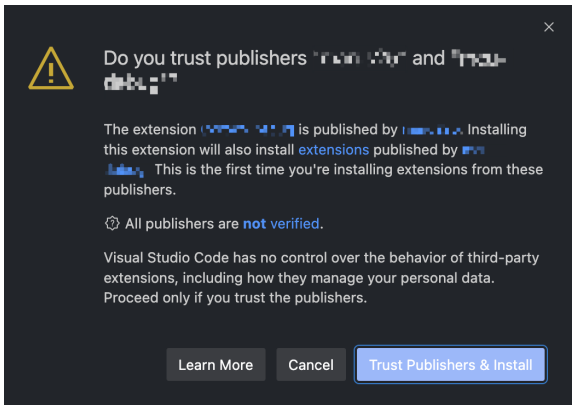

Figure 2.2: Dialog when installing extension in VS Code, requiring confirmation of trusting the publisher, and all dependent extension publishers.

The entry point of a VS Code extension, as described by the *Extension Anatomy* [18], is defined in the `package.json` manifest file. Here, the developer states the entry point file using the `main` field, and defines the extension commands present in the `commands` field, located within the outer `contributes` field. This means that when an extension is run, it targets the output JavaScript file located at `main`'s path. This will differ between plain JavaScript extensions (see Listing 2.4, taken from their JavaScript minimal sample [4]) and TypeScript extensions (See Listing 2.5, taken from Extension Anatomy [18]), as the TypeScript version must first be compiled to a JavaScript file before being run in VS Code.

```
{
"main": "./extension.js"
"contributes": {
    "commands": [
        "command": "example.helloWorld"
        "title": "Hello World Example"
        ]
    },
... //More package.json data
}
```

Listing 2.4: `package.json` for JavaScript VS Code Extension

```
{
"main": "./out/extension.js"
"contributes": {
    "commands": [
        "command": "example.helloWorld"
        "title": "Hello World Example"
        ]
    },
"scripts": {
    "vscode:prepublish": "npm run compile",
    "compile": "tsc -p ./",
    "watch": "tsc -watch -p ./"
    },
... //More package.json data
}
```

Listing 2.5: `package.json` for TypeScript VS Code Extension

An extension's entry file commonly exports two functions; **(i)** *activate*, and **(ii)** *deactivate* [18]. The *activate* function serves as the entry point function of the extension, starting with the extension itself, while *deactivate* is executed before the extension is exited. All commands are declared within the *activate* function, essentially serving as a `main` function in C. Whether being delegated to other functions or files, this is the entry point of the extension. As shown in the examples below, there are minor differences between the JavaScript implementation and exporting of the functions (see Listing 2.6) in contrast to TypeScript (see Listing 2.7), though the general structure is closely similar.

```
const vscode = require('vscode');

function activate(context) {
    let disposable = vscode.commands.
        registerCommand('js.yourCommand',
        () => {
        // Your command code
        }
    );
}

function deactivate() {
    //Cleanup code if needed
}

module.exports = {
    activate,
    deactivate
}
```

Listing 2.6: JavaScript VS Code Extension Entry Point

```
import * as vscode from 'vscode';

export function activate(context: vscode.
    ExtensionContext) {
    let disposable = vscode.commands.
        registerCommand('ts.yourCommand',
        () => {
        // Your command code
        }
    );
}

export function deactivate()  {
    // Cleanup code if needed
}
```

Listing 2.7: TypeScript VS Code Extension Entry Point

2.6 Static Analysis

Static analysis is the process of examining source code or compiled code without executing it [8]. It analyses the structure, syntax, and semantics for potential flaws, ranging from security vulnerabilities to coding praxis violations and maintainability deficiencies. While this approach excels at finding structural patterns that indicate issues, runtime-based inconsistencies can go unnoticed. Another consideration is the possibility of false positives, skewing the attack vectors from the actual vulnerabilities.

CodeQL is an open-source semantic code analysis engine developed by GitHub, primarily designed for detecting vulnerabilities within a codebase [26]. However, its capabilities extend beyond security analysis, making it a powerful tool for program analysis, dependency analysis, and cross-language code flow tracking. Its compatibility with multiple languages and the possibility of working with multilingual repositories by using flags to determine the included languages, in addition to its included language-specific example queries for "relevant and interesting problems" [25], gives the user a good foundation to start from.

Before running queries, CodeQL needs to generate a database, which it does by converting source code into a relational database containing a structured representation of the codebase. This database includes multiple program analysis representations such as the abstract syntax tree, the data flow graph, and the control flow graph [24].

The database schema varies for each supported language and defines tables for different languages. CodeQL provides an abstraction layer over these tables, making it easier to write queries using QL, an object-oriented query language [24].

For compiled languages such as C and C++, database extraction involves monitoring the build process, capturing compiler invocations, and extracting relevant syntactic and semantic data. For interpreted languages like JavaScript and TypeScript, the extractor analyses the source code directly, resolving dependencies to create an accurate representation of the code base [24].

3

Related Work

As previously mentioned, this section will include works from multiple domains. To ensure clarity for the reader, it has been split into specific sections, where relevant previous works are categorised based on their main association.

3.1 Cross-language Security

Brown et al. [15] focuses on detection and exploitation in the bindings of the Node and Chrome runtimes using static analysis, finding that there are numerous exploitable security holes in both runtimes, propagating as violations of memory-, crash-, or type-safety.

Staicu et al. [47] instead broadened their study to include the cooperation between several programming languages and their native APIs. With static analysis using Joern and Google's Closure Compiler, they concluded that issues ensue in all languages, but the misuses differ in nature. However, they found that the Node APIs have lenient handling of most misuses, compared to Python's and Ruby's alternatives.

Roth et al. [44] created cross-language analysis by integrating multiple single-language analyses using a coordinator bridging the languages, serving as the monitor of the entire application. Although not directly investigating the cooperation between JavaScript and native code, but instead using Java as an intermediary layer, this approach proved effective for cross-language cooperation, for both JavaScript-to-Java interactions, and bindings from Java to native code.

Scholtes et al. [45] presented CHARON, the first interprocedural, polyglot static analysis for detecting vulnerabilities in scripting languages and across the boundary to native code. It supports crossing the boundaries between languages multiple times, tracking the data flow between them. They found over 5,800 vulnerable data flows across 116 packages, from both `npm` and `PyPI`. They demonstrate that many real-world security bugs lie not within one language, but in the connections between them.

Zhu et al. [50] perform the first systematic study on the potential abuse of TensorFlow's cross-language API surface, demonstrating how legitimate Python-facing

APIs to C++ implementations can embed stealthy malware into AI models. By analysing TensorFlow v2.15.0, they identify over 1,000 persistent APIs and categorise their capabilities, revealing overlooked security-critical behaviours such as file and network access. Their attack, TensorAbuse, exploits the persistence of such cross-language APIs and their hidden semantics to construct malicious behaviours that activate during model inference. Crucially, the attack does not rely on vulnerabilities or serialisation flaws and bypasses detection by Hugging Face Hub, TensorFlow Hub, and existing scanning tools.

In contrast to static analysis, *fuzzing* explores the boundaries of a program by automatically constructing inputs to find edge cases in the program; thus, potentially generating crashes, memory leaks or violated runtime assertions. While previous fuzzing tools, such as Skyfire [49] and Langfuzz [28], solely focus on a single language, Dinh et al. presented Favocado [22], the first fuzzing tool that detects bugs in the binding layer by creating semantic-aware test cases using the API references. They found 61 previously unknown vulnerabilities across 4 JavaScript runtime systems, though the Node runtime was not investigated.

3.2 VS Code Security

Jin et al. [30] analysed Electron applications, including VS Code, and found vulnerabilities such as XSS and scriptless attacks, which can lead to information leakage and other security issues. Their study identifies risks emerging from unintended DOM mutations (e.g., via malformed HTML/CSS inputs), which can lead to script injection, privacy leaks, or UI deception—even without direct Node.js integration. For example, improperly sanitised user inputs in apps like Microsoft Teams or Visual Studio Code allow attackers to inject malicious scripts or exfiltrate data through crafted DOM elements.

Lin et al. [35] used proof-of-concept exploits in VS Code extensions to identify 716 dangerous data flows and 21 verified extension vulnerabilities. They identified both untrusted inputs and code targets for code injection and file integrity attacks, using these insights to design taint analysis rules for CodeQL. Their study highlights that some extensions can start local web servers to provide specific functionalities, which, if not properly secured, can become vectors of attacks.

Edirimannage et al. [23] found that out of 52,880 extensions taken from the marketplace, 2969 were potentially malicious. They investigated the issue from a developer's perspective, focusing on the potential of a software supply chain attack, where sensitive data can be exposed for malicious intent. Furthermore, they demonstrate how VS Code's architecture, particularly its lack of extension sandboxing and permission model, enables risks by granting extensions unchecked access to host systems and developer workspaces. Their conclusion covers recommendations for mandatory sandboxing, runtime permission controls, and marketplace transparency to highlight the current gaps in the underlying software.

In summary, research regarding VS Code security exists, but no previous work has handled the issue from a cross-language perspective. This shows the current lack of investigation in this area.

3.3 Node.js Security

Huang et al. [29] used both static and dynamic analysis to analyse malicious intent in the `npm` ecosystem from the angle of software supply chain attacks. The detector is built with static analysis, identifying suspicious API call sequences, and dynamic analysis, which captures runtime behaviours that might be obfuscated or evasive. They found that their dynamic execution significantly complements static detection in the detector, identifying packages that employed obfuscation techniques to avoid static analysis. They amassed 325 previously unknown malicious packages and discovered novel API call patterns, grouping them in five different categories; **(i)** sensitive information theft, **(ii)** sensitive file operations, **(iii)** malicious software import, **(iv)** reverse shell, and **(v)** suspicious command execution.

Decan et al. [20] investigated security vulnerabilities in the `npm` dependency network of over 610 thousand packages, and found that 72,470 packages depended on potentially vulnerable packages. They found that it took between 20 and 39 months to discover 50% of vulnerabilities, depending on their severity. While some packages are fixed before the vulnerabilities are found and publicly announced, others remain unfixed.

Zimmermann et al. [51] emphasise the risks posed by densely connected dependencies and maintainers, demonstrating that a single compromised package or account can have cascading effects across thousands of packages. They found that a package, on average, implicitly trusts 79 third-party packages; with the most influential packages impacting more than 100,000 dependents. They also found that up to 40% of packages depend on code containing at least one known vulnerability and that the `npm` ecosystem's lack of systematic vetting increases these risks, leaving developers to rely on ad-hoc tools like `npm` audit, which only addresses direct dependencies. They suggest vetting the top 1500 packages to reduce the attack surface significantly, though this would cost substantial resources. Their findings align with earlier work by Lauinger et al. [34], which revealed that outdated JavaScript libraries are pervasive in web applications, exposing users to known vulnerabilities.

3.4 Browser Extension Security

Kapravelos et al. [32] proposed Hulk, a dynamic analysis system for detecting malicious Chrome extensions by combining two factors: *HoneyPages*, dynamic pages that mimic DOM structures, and *event handler fuzzing* to trigger malicious behaviour. Out of 48,000 analysed extensions, they identified 130 being malicious, including extensions with millions of users abusing privileged APIs like the `webRequest`-API.

Bandhakavi et al. [13] proposed the tool VEX for detecting vulnerabilities in Firefox extensions. By tracking flows from untrusted sources (e.g. page content) to executable sinks (e.g. `eval`), VEX discovered 12 vulnerabilities across 2,460 extensions. They assumed that the extensions, though flawed, were not malicious by intent, but instead that the developer could write vulnerable code due to insufficient knowledge.

Pantelaios et al. [43] address the challenge of detecting malicious browser extensions using an update delta-analysis approach. The authors analyse 922,684 extension versions over six years, identifying malicious updates by clustering code deltas based on abused APIs, for example, ad injection and history tracking. Their two-stage method first detects "seed" extensions via anomalies in user ratings and comments by taking sudden negative reviews containing keywords like "malware" into account. Second, they propagate these findings to uncover 143 malicious extensions across 21 clusters. A key insight is that 44% of detected extensions evaded Chrome Web Store's protections, highlighting limitations in current vetting systems, much like in the VS Code marketplace.

4

Methods

Our approach combines repository mining techniques with static security analysis to systematically investigate cross-language dependencies in the VS Code ecosystem. The methodology addresses two primary challenges: **(i)** identifying which extensions contain or depend on native code, and **(ii)** assessing the security implications of these cross-language interactions. Each step in our pipeline builds upon previous results, enabling both analysis of dependency prevalence and assessment of security risks.

Our analysis identifies vulnerabilities present within an extension or its dependencies, regardless of whether they are reachable from extension code. This provides a conservative assessment of potential security exposure. This chapter outlines the methodology employed to conduct the present analysis systematically and rigorously.

Methodology Overview

We constructed a five-step pipeline to address our research questions mentioned in section 1.1, as shown in Figure 4.1. The first two steps aim to answer RQ1, while the last three address RQ2.

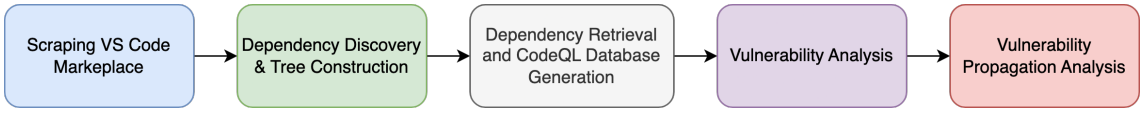

Figure 4.1: Overview of our methodology pipeline

Below, we provide a brief overview of each step, followed by a detailed explanation in the subsequent subsections.

1. **Scraping VS Code Marketplace:** We collect VS Code extensions from the Microsoft Visual Studio Marketplace to gather relevant data for our research. This is further detailed in section 4.1.

2. **Dependency Discovery and Tree Construction:** We parsed each extension's manifest file to identify direct and transitive dependencies, and constructed a dependency tree from the gathered structure. We then collected

the usage of compiled languages (C/C++) from the GitHub API (see Appendix A.1) and filtered out all dependency branches not including native code. This is detailed further in section 4.2.

3. **Dependency Retrieval and CodeQL Database Generation:** We retrieve source code for each dependency and create CodeQL databases for both extensions and dependencies. By resolving semantic versions and automating the build process, we ensured that each database accurately reflects the code under analysis. This is further detailed in section 4.3.

4. **Vulnerability Analysis:** We detect vulnerabilities in VS Code extensions and their native dependencies using custom CodeQL queries. We analyse both VS Code extensions and their native C/C++ dependencies individually to identify potential vulnerabilities. This is further detailed in 4.4.

5. **Vulnerability Propagation Analysis:** We use the previously constructed dependency trees to map vulnerabilities discovered in native dependencies back to the extensions that depend on them, determining which extensions are potentially affected and at what dependency depth the vulnerabilities occur. This is further detailed in 4.5.

4.1 Scraping VS Code Marketplace

To collect relevant data, we scraped extensions from the Visual Studio Code Marketplace[1], sorted by the number of downloads. We chose the install count as our selection criteria because it represents actual usage impact in the ecosystem, ensuring our analysis focuses on extensions that affect the most users in practice. While this approach may introduce sampling bias by potentially missing vulnerability patterns present in less popular extensions, it allows us to assess security risks where they have the greatest real-world impact. Our data collection was conducted on 2025-05-05, ensuring our analysis reflects the extension ecosystem state at that time.

A custom script was developed to automate the retrieval of extensions. The Marketplace API allowed us to utilise pagination to fetch data systematically. This allowed incremental fetching and provided flexibility to pause and resume data collection across multiple sessions. The script incorporated rate-limiting and exponential backoff strategies to efficiently manage network resources and reduce the risk of being rate-limited.

Furthermore, parallelising the cloning operation of each extension's GitHub repository using Python's `ThreadPoolExecutor` made it possible to clone multiple repositories concurrently. In addition, mechanisms for tracking progress were incorporated so that the script could save incremental progress and pick up where it had left off from interrupted scrapes. Extension metadata, including publisher details, cloning

[1]Link to VS Code Marketplace

status, timestamps, and repository URLs, was saved in CSV files for traceability and reproducibility purposes.

Having obtained the source code for our target extensions, the next step was to systematically analyse their dependency structures to identify which extensions rely on native code.

4.2 Dependency Discovery and Tree Construction

To analyse VS Code extensions systematically for dependencies, we crafted a structured process with two main steps: initial discovery of dependencies and construction and pruning of a dependency tree. Together, these steps aim to provide a complete and efficient analysis.

Below, a workflow is given for creating a single dependency tree, starting from an extension, and being finalised as a pruned dependency tree only containing native branches. Thus, this is done for every extension in the dataset, and each extension containing at least one native dependency is output as a `json` file.

4.2.1 Extension Dependency Tree Structure

Before describing how the dependency tree is built, pruned and used in the methodology, the building blocks of the dependency tree will be defined. The term "dependency tree" will describe the dependency structure during the methodology and the following chapters. In the dependency tree, we have three descriptive entities, that is exemplified in Listing 4.1, that are defined as:

- **Nodes**: A *node* is either an extension node or a `npm` dependency node. In contrast to a dependency, an extension will always and only serve as a root. Moreover, an extension node requires less information tied to it, as it is the root node.
- **Edges**: An *edge* is the dependency a node puts on another node. For example, this can mean that either **(i)** an extension node depends on a dependency node, or **(ii)** a dependency node depends on another dependency node. As the extension node is always the root node, there is no scenario where an extension can be depended upon in the dependency tree.
- **Properties**: A *property* is additional data that describe the nodes and edges. Extension nodes only have a single property for including native code directly within. This stems from always being a root node, thus the starting point when creating the tree. Dependency nodes have more properties attached than extension nodes.
  - **Metadata**: Package name and GitHub repository URL.
  - **Language Analysis**: Programming languages detected via GitHub API

with byte counts (see Appendix A.1), and a boolean value for whether it has native code within, based on the response from the API request, called `has_cpp_directly`. When pruning the dependency tree, the `contains_compiled` is updated based on whether the node or any outgoing branches have native code.

- **Hierarchical Structure**: Nested outgoing dependency edges and the depth of the dependency node.

Although the version of a dependency node is stated in the node itself for better readability, this is a property of the edge from the source node to the target node.

Listing 4.1 shows an example of a dependency tree structure for an extension with a transitive C/C++ dependency:

```
{
  "ext_native": false,
  "dependencies": {
    "example-dependency@3.5.3": {
      "name": "example-dependency",
      "version": "3.5.3",
      "github": "https://github.com/owner/repo",
      "languages": {
        "TypeScript": 134725,
        "JavaScript": 352
      },
      "contains_compiled": true,
      "has_cpp_directly": false,
      "depth": 0,
      "dependencies": [
        {
          "name": "example-dependency-2",
          "version": "2.1.0",
          "github": "https://github.com/owner/repo",
          "languages": {
            "C++": 12888,
            "JavaScript": 11408,
            "Python": 716
          },
          "contains_compiled": true,
          "has_cpp_directly": true,
          "depth": 1,
          "dependencies": []
        }
      ]
    }
```

```
32    }
33 }
```

Listing 4.1: Example of extension dependency tree structure showing hierarchical relationships and language detection

The resulting extension dependency tree is stored as JSON files, with one JSON file per extension. This structured representation enables systematic analysis of how native code dependencies propagate through the extension ecosystem. These trees form the foundation for subsequent analysis.

4.2.2 Dependency Discovery

The first step of our dependency discovery involves identifying all direct and transitive dependencies for a given extension using npm's built-in dependency resolution. Rather than manually traversing the `npm` registry, we leverage `npm`'s `npm ls` command to obtain the complete dependency tree. This dependency tree corresponds to what `npm install` resolves, ensuring we capture the exact dependencies that would be installed in practice [42].

Cross-language dependencies can occur in VS Code extensions through three mechanisms: **(i)** direct implementation of native code within the extension itself, **(ii)** direct dependencies on packages containing native code, or **(iii)** transitive dependencies where native code exists deeper in the dependency chain. Since these dependencies can be deeply nested, our approach must analyse the complete dependency tree to identify all potential cross-language interactions.

Our dependency discovery algorithm (see Algorithm 1) explores all dependencies used by each extension. The algorithm traverses all downloaded extensions, automatically installing their dependencies using `npm install` (See Algorithm 1, row 9), with fallback mechanisms that handle partial installation failures while allowing dependency extraction to proceed. The extraction process then recursively parses `npm`'s JSON output from the `npm ls` command (See Algorithm 1, row 10) to capture all transitive dependencies from an extension, with their exact resolved versions. This approach follows `semver` principles while maintaining the complete dependency hierarchy. This version-specific precision allows us to assess the current state of VS Code extensions' dependencies, as future potential vulnerabilities may be present in one version of a dependency but patched in later releases, enabling us to identify which extensions are currently using potentially vulnerable dependency versions.

The algorithm uses GitHub's API (see Appendix A.1) to determine programming languages for each dependency repository (see Algorithm 1, row 17), using rate limiting and graceful handling of repository redirects and renames. To increase efficiency and reduce repeated network calls, dependency metadata such as programming language and repository URLs is cached, reducing processing time in subsequent analysis runs.

Algorithm 1: Dependency Discovery for VS Code Extensions.

Input: Scraped extension E
Output: Complete dependency tree T_E for E

```
1  T_E ← {};
2  if package.json not found in E then
3  |  return T_E;
4  end
5  ExploreExtension(E);
6  Function ExploreExtension(E):
7  |  Run npm install in E;
8  |  E.dependencies ← npm ls -omit=dev -depth=50 -json in E;
9  |  ExtractDependencies(E, 1)
10 Function ExtractDependencies(node, depth):
11 |  foreach child d in node.dependencies do
12 |  |  repo ← npm info(d).repository;
13 |  |  langs ← GitHubAPI.getLanguages(repo);
14 |  |  d.depth = depth;
15 |  |  Add d to T_E;
16 |  |  ExtractDependencies(d, depth +1)
17 |  end
18 return T_E
```

To exemplify the output of Algorithm 1, we will use the simple dependency structure for a single extension from Listing 4.2. As this is what is returned by `npm ls`, the following example starts here (See Algorithm 1, row 10) and describes the row-wise operations. The next-coming paragraph will heavily refer to both Listing 4.2 and Algorithm 1, and these referrals will thus be implicit to make the instructions more readable.

```
1 Extension E
2 ├── D1 @ 1.0.0
3 ├── D2 @ 5.0.0
4 |   └── D3 @ 2.5.0
5 |       └── D4 @ 4.12.0
6 └── D5 @ 1.0.0
7     └── D6 @ 2.2.2
```

Listing 4.2: Output from `npm ls` dependency structure.

Row 9 starts the traversal of the entire tree of extension E. Every direct child of E, i.e., $D1$, $D2$, and $D5$, will be iterated through on row 11. We collect the repository data and languages, and attach the respective depth to the corresponding dependency nodes on rows 12, 13, and 14, respectively, and then add the child to the extension tree. On row 16, we extract the dependencies of the currently handled dependency, i.e., $D1$, $D2$ and $D5$, respectively, on `depth`=1. $D1$ has no direct

children; thus, nothing is done here, and this dependency branch is exhausted. $D2$ and $D5$ have one direct child each; we extract them ($D3$ and $D6$), and do the same for them as the previous dependencies. When we explore these dependencies on row 16, only $D3$ has a direct child, and $D6$ is exhausted. When all dependency branches are exhausted, the output is the complete tree, with all additional properties added.

4.2.3 Native Dependency Detection and Tree Pruning

The second stage uses previously collected dependency data to prune dependency trees systematically. The process is detailed in Algorithm 2, and uses a pruning approach that preserves the complete path to any dependency containing native code. This ensures that intermediate dependencies without native code are retained if they lead to native dependencies, maintaining the full dependency chain for security analysis.

Our analysis focuses on standard `dependencies` as declared in `package.json`, representing packages required for runtime functionality. We exclude `devDependencies` as these affect only developers, not end-users, aligning with our research focus on user-facing security risks. Other dependency types (`peerDependencies`, `bundleDependencies`, `optionalDependencies`) were not analysed due to time constraints, though they represent opportunities for future work.

4.3 Dependency Retrieval and CodeQL Database Generation

CodeQL was chosen due to its multi-language support, as the original plan was to create a cross-dependency, cross-language call-graph that mapped the function calls from the extension entry point, all the way to the potential native vulnerabilities. This is further discussed in Section 6.8.

To enable the security analysis of cross-language dependencies, CodeQL databases were created for all dependencies containing C/C++ code. This involves two main challenges: **(i)** obtaining the exact source code versions specified by each extension's dependency constraints, and **(ii)** successfully compiling diverse native dependencies across different build environments and Node.js versions. This section describes our systematic approach to addressing these challenges and constructing the databases that provide the foundation for vulnerability analysis.

4.3.1 Retrieving dependencies source code

To enable security evaluation of VS Code extensions' cross-language dependencies, we first needed to obtain the source code for each identified dependency. This process utilised the dependency trees constructed in Section 4.2, which had already identified the relevant dependencies and their corresponding GitHub repository URLs.

The retrieval strategy was targeted at achieving exact versions of dependencies as

Algorithm 2: Dependency Tree Construction and Pruning

Input: Dependency Tree T_E for an extension E
Output: Pruned tree T'_E with only C/C++-relevant paths

```
E ← T_E.get(E);
MarkCpp(E);
PropagateCpp(E);
PruneTree(E);
Function MarkCpp(node):
    if node.languages contains C or C++ then
        node.has_cpp_directly ← true
    end
    foreach child in node.dependencies do
        MarkCpp(child)
    end
Function PropagateCpp(node):
    compiled ← node.has_cpp_directly;
    foreach child in node.dependencies do
        compiled ← compiled OR PropagateCpp(child)
    end
    node.contains_compiled ← compiled;
    return compiled
Function PruneTree(node):
    node.dependencies ← [child ∈ node.dependencies | PruneTree(child)];
    return node.contains_compiled OR node.has_cpp_directly
return T'_E with only nodes where contains_compiled = true
```

required by `semver` constraints mentioned in each extension's dependency tree from section 4.2. Our algorithm tried to clone the repository for the version corresponding to these constraints. More specifically, the following were the steps taken:

- **First Cloning Attempt**: We tried to clone the GitHub repository from a specific version tag using the resolved version from the dependency tree. This cloning used shallow cloning methods (`git clone --depth 1 --branch <version>`) to quickly fetch the required data.

- **Fallback Mechanism**: Where the precisely specified version was not available in the repository, we resorted to cloning the repository in its most up-to-date available state (default branch). An effort was then made to check out the particular version tag after cloning. In case it also failed, the repository was left in the latest version state.

Following this methodology, we prioritised our analysis based on exact, version-specific source code to improve accuracy and validity in subsequent security evaluations.

4.3.2 Creating CodeQL Databases

After successfully obtaining the source code for the dependencies, we constructed the CodeQL databases necessary for performing our security analyses. Our database creation process employed a multi-environment approach to handle diverse compilation requirements across different dependency ages. We configured three Node.js environments with corresponding Python and `node-gyp` versions: modern (Node.js 22.13.0), intermediate (Node.js 14.20.1), and legacy (Node.js 8.10.0).

We automatically selected the appropriate `conda` environment based on the `node-gyp` version specified in each dependency's or extension's `package.json`. Nodes without specific version requirements defaulted to attempting compilation across all environments in descending order of modernity. Environment-specific configurations were maintained and restored after each build attempt to ensure consistency. This environment selection approach was important for handling the varied nature of the extension ecosystem, where dependencies often have different compilation requirements based on their age and target platforms. Nodes specifically designed for Windows or macOS platforms were excluded due to our Linux-based build environment.

Each node, both extension and dependencies, was scanned for `binding.gyp` files (excluding those in `node_modules` directories), which serve as configuration files for `node-gyp` to define how C/C++ code should be compiled and linked into Node.js addons. Only nodes containing valid `binding.gyp` files proceeded to C++ database creation, as this indicates an available native build process. Dependencies without `binding.gyp` files were excluded from C++ analysis.

The build process for each dependency or extension consisted of three steps: **(i)** updating git submodules recursively, **(ii)** installing dependencies using `npm install`, and **(iii)** creating CodeQL databases using `node-gyp rebuild` in the appropriate binding directory. Our environments focused on the standard `node-gyp` toolchain rather than supporting every possible build system. Failed builds were retried across multiple environments before being marked as unsuccessful.

For nodes where our automated build process failed across all environments, we implemented a fallback strategy of attempting to download pre-built CodeQL databases directly from GitHub. GitHub automatically generates CodeQL databases for many public repositories through its security scanning infrastructure. However, we found that GitHub's `autobuild` process often failed to capture the complete scope of the native code, and the version correspondence between the downloaded databases and our specific versions remained unclear. Despite these limitations, this fallback approach allowed us to extend our analysis coverage, albeit with reduced confidence in the completeness of the resulting databases.

With CodeQL databases constructed for both extensions and their native dependencies, we proceeded to systematic vulnerability detection using custom security queries.

4.4 Vulnerability Analysis

Following the creation of CodeQL databases for both VS Code extensions and their native dependencies, we conducted a systematic vulnerability analysis targeting the C/C++ components. Due to time constraints, we implemented CodeQL queries for 8 of the 13 identified vulnerability types, prioritising those most prevalent in related work and most feasible to implement reliably. The implemented vulnerability types were: MEM-01, MEM-02, MEM-03, MEM-04, MEM-05, MEM-07, INJ-01, and RES-01, as categorised in Table 4.1. This analysis employed custom CodeQL queries designed to detect specific vulnerability patterns across memory safety issues (MEM), injection vulnerabilities (INJ), and resource management flaws (RES).

While CodeQL provides built-in security queries, we developed custom implementations to better integrate with our analysis pipeline and address our specific research context. These custom queries target vulnerability patterns relevant to the JavaScript-to-C++ interface found in VS Code extensions and their native dependencies, and generate structured output for subsequent vulnerability propagation analysis. The queries were tested against created extension benchmarks (detailed in Section 5.3) to verify their detection capabilities before use across the ecosystem.

The vulnerability detection process was automated through a Python script that systematically executed each query against all available CodeQL databases. Each vulnerability-specific query (e.g., MEM-01, INJ-01) was executed against every database containing C/C++ code, including both extension databases and dependency databases. For successful executions that identified potential vulnerabilities, structured results were captured in CSV format, with each vulnerability type generating a dedicated file (e.g., `custom_MEM-01.csv`) containing file paths, line numbers, vulnerability descriptions, and specific context information.

To ensure result quality, we implemented several filtering mechanisms. Duplicate findings were removed, and results originating from `node_modules` directories were excluded to focus on the projects' own code. This filtering process helped eliminate noise and concentrate on vulnerabilities directly relevant to the extension ecosystem.

The structured output from this vulnerability analysis serves as input for the subsequent vulnerability propagation analysis, where we assess which specific extensions are affected by identified vulnerabilities. This approach enables detection of security issues across the ecosystem while maintaining traceability from individual code locations to affected extensions.

Table 4.1 presents a set of vulnerabilities that may arise from the improper or malicious use of cross-language interactions.

Table 4.1: Classification of Cross-Language Dependency Vulnerabilities, along with their corresponding CWE-labelling.

ID	Category	Description	CWE
Memory Safety Vulnerabilities (MEM)			
MEM-01	Inadequate Type Validation	Converting JavaScript values to C/C++ types without proper validation; not checking the number of arguments passed to native functions	*CWE-20* *CWE-685*
MEM-02	Uninitialised Memory Exposure	Returning uninitialised buffer contents to JavaScript; improper string termination in C/C++ code	*CWE-908* *CWE-170*
MEM-03	Memory Leaks via Persistent Handles	Creating persistent handles without proper cleanup mechanisms; missing destructor calls or reference counting issues	*CWE-401*
MEM-04	Buffer Overflow Vulnerabilities	Buffer operations without bounds checking; fixed-size buffers handling variable-sized input from JavaScript	*CWE-120* *CWE-787*
MEM-05	NULL Terminator Handling Issues	String operations assuming NULL-terminated strings from JavaScript; length calculations not accounting for embedded NULL characters	*CWE-170* *CWE-464*
MEM-06*	Integer Overflow/Underflow	Numeric conversions without range checks; size calculations based on JavaScript numeric inputs without validation	*CWE-190* *CWE-191*
MEM-07	Use-After-Free and Double-Free	Objects freed in one extension function but still used in another; improper lifecycle management of objects; mismatched memory allocation patterns	*CWE-415* *CWE-416*
Injection Vulnerabilities (INJ)			
INJ-01	Command Injection	Native functions that call system(), exec(), spawn() with insufficient sanitisation; JavaScript strings concatenated directly into command line arguments	*CWE-78*

Continued on next page

Table 4.1: Classification of Vulnerabilities (continued)

ID	Vulnerability	Description	CWE
INJ-02*	Path Traversal	Functions that access the filesystem without path normalisation; insufficient validation of path components; potential to break out of intended directory boundaries	*CWE-22* *CWE-23*
INJ-03*	Format String Vulnerabilities	Format strings containing user-controlled input passed to printf-family functions; potential for information disclosure or memory corruption	*CWE-134*
Race Conditions and Concurrency Issues (RACE)			
RACE-01*	TOCTOU Race Conditions	File operations that verify permissions/existence before usage without proper locking; exploitable window between checking and using	*CWE-367*
Resource Management Vulnerabilities (RES)			
RES-01	Insecure Temporary File Creation	Predictable temporary file paths; inadequate permissions on temporary files; failure to clean up temporary files	*CWE-377* *CWE-379*
RES-02*	Cross-Extension Resource Consumption	Native code that consumes excessive resources; lack of resource limiting when processing user input; potential denial of service	*CWE-400* *CWE-770*
Information Disclosure (INFO)			
INFO-01*	Improper Error Handling	Errors exposing stack traces or memory addresses; crashes leaking internal state; unsanitised error information passed from C++ to JavaScript	*CWE-200* *CWE-209*

** These vulnerabilities are not explicitly analysed in this thesis but are identified as potential attack vectors within VS Code extensions.*

4.5 Vulnerability Propagation Analysis

The vulnerability propagation analysis phase determines how security vulnerabilities identified in native dependencies propagate through the dependency chain to affect VS Code extensions. This step addresses RQ2 by quantifying the security exposure of extensions and analysing how cross-language dependencies introduce vulnerabilities at the ecosystem level.

After identifying security vulnerabilities in individual native dependencies through CodeQL analysis (Section 4.4), we needed to determine which extensions were affected by these vulnerabilities. The vulnerability propagation analysis phase translated vulnerability findings from the dependency to the extension, allowing us to assess the scope of security issues and identify distribution patterns.

The vulnerability propagation process leveraged the extension dependency trees from Section 4.2 and comprised of three main steps:

1. **Vulnerability Aggregation:** The vulnerability analysis results were initially stored in individual CSV files per dependency, then combined into a unified report summarising the number and types of vulnerabilities found in each dependency version (see Listing 4.3).

2. **Dependency Matching:** We created an automated process that traversed each extension's dependency tree to identify vulnerable dependencies. The matching process normalised dependency names by replacing special characters (e.g., `@types/node` becomes `types-node`) and used exact version information when available. When specific version information was missing, we employed a "latest" fallback strategy, which provided broader coverage by matching dependencies regardless of version constraints while maintaining conservative security assessment.

3. **Recursive Propagation:** Each extension's dependency tree was checked recursively to find all nested dependencies, ensuring that vulnerabilities in indirect dependencies were properly linked to affected extensions through the complete dependency chain.

For each extension, we calculated several key metrics to quantify vulnerability exposure:

- **Direct exposure:** vulnerabilities present in immediate (depth-0) dependencies

- **Indirect exposure:** vulnerabilities propagated through transitive dependencies

- **Vulnerability distribution:** how vulnerabilities spread across different dependency depths

- **Dependency density correlation:** whether extensions with more dependencies exhibited proportionally higher vulnerability counts

The process was automated using Python scripts that matched and combined results for both extensions and dependencies. Different versions of the same dependency were treated separately, since different extensions might use different versions with varying security issues.

Each extension's vulnerability exposure was recorded in a structured vulnerability propagation report containing comprehensive vulnerability details and propagation statistics. The report structure captures both the vulnerabilities themselves and metadata about how they propagate through the dependency chain. Listing 4.3 shows an example of the output generated for an extension with identified vulnerabilities:

```
{
  "extension": "example_extension",
  "total_vulnerabilities": 4,
  "vulnerability_breakdown": {
    "MEM-07": 4
  },
  "affected_dependencies": {
    "package-a@1.2.3": {
      "vulnerabilities": { "MEM-07": 4 },
      "depth": 2,
      "match_type": "exact"
    }
  },
  "depth_analysis": {
    "max_depth": 3,
    "vulnerabilities_by_depth": { "2": 4 }
  }
}
```

Listing 4.3: Simplified vulnerability propagation report structure showing vulnerability attribution and dependency chain metadata

In the above example, `depth: 2` indicates that the vulnerability was found in a dependency two levels deep in the dependency chain, while `match_type: "exact"` shows that the dependency version was matched precisely rather than using the fallback strategy. Depth analysis was used to calculate vulnerability distribution across different depths.

These vulnerability propagation reports enabled analysis for both individual extensions and their dependencies. The structured output provides the foundation for answering RQ2 by quantifying how cross-language dependencies introduce security risks and identifying patterns in vulnerability distribution across the VS Code extension ecosystem, as presented in our results (Chapter 5).

4.6 Methodological Limitations

Our analysis depends on GitHub repository metadata to determine programming language usage, which may not accurately reflect the actual languages used in `npm` packages. Repository URLs in `package.json` files are not validated, potentially leading to false positives or negatives in native code detection.

Static analysis inherently produces both false positives and false negatives. Our conservative approach may flag benign code patterns as vulnerabilities while missing subtle security issues that require runtime context. The absence of call-graph analysis means we cannot distinguish between vulnerabilities that are reachable from extension code versus those that exist but are never invoked.

Our focus on the most popular extensions (top 3,078 by install count) may not represent vulnerability patterns present in the broader ecosystem of 70,000+ available extensions. Additionally, our analysis is limited to open-source extensions with accessible GitHub repositories. Extensions and dependencies hosted on alternative platforms such as GitLab, Bitbucket, or self-hosted repositories were excluded from the analysis, potentially omitting relevant cross-language dependencies.

The automated build process for CodeQL database creation succeeded for most dependencies but failed for some, requiring specialised build environments. Dependencies without `binding.gyp` files were excluded from C/C++ analysis, potentially missing native code that uses alternative build systems.

4.7 Reproducibility

We provide all source code, scripts, CodeQL queries, and datasets used in this study to enable reproduction of our results. The materials include the complete analysis pipeline, dependency trees, vulnerability detection queries, and setup details.

Our methodology can be applied to current extension data, though exact reproduction requires the same extension versions from our analysis date (2025-05-05). GitHub link

5

Results

This chapter presents results categorised by the respective research question. First, we answer the quantitative question of cross-language dependencies within the dataset. Second, we answer the question of any vulnerabilities occurring within the gathered cross-language dependencies.

5.1 RQ1: Presence and Characteristics of Cross-Language Dependencies

To answer RQ1, "Do VS Code extensions contain cross-language dependencies, and what are their characteristics?", an analysis was conducted of the top 3,078 Visual Studio Code extensions regarding downloads, selected to represent over 95% of all installs on the Marketplace, as of 2025-05-05. The aim was to identify whether extensions include native code, either directly or through dependencies written in native languages, and to characterise how such dependencies are integrated. The analysis was performed by recursively traversing each extension's dependency tree, identifying transitive packages, and examining whether they include native source code.

In the following section, we present the prevalence of cross-language dependencies and describe their common characteristics.

5.1.1 Data Collection and Processing Completeness

The analysis pipeline processed extensions through three distinct stages:

Stage 1 - Download: Of the initial 3,078 extensions, 2,873 (93.3%) were successfully downloaded from GitHub repositories. Meanwhile, 205 failed to download due to missing GitHub URLs or inaccessible private repositories.

Stage 2 - Parsing: Among downloaded extensions, 75 (2.6% of downloaded, 2.4% of initial) experienced parsing failures due to corrupted package.json files or npm installation issues.

Stage 3 - Analysis: The remaining 2,798 extensions (91.0% of the initial dataset)

were successfully analysed and form the basis for all reported statistics. Of these analysed extensions, 1,599 (57.1%) had no declared dependencies, leaving 1,199 (42.9%) that declared at least one dependency and formed the further analysed dataset for constructing the dependency tree.

Extensions without dependencies represent a substantial portion of the ecosystem and include cases such as theme extensions containing only JSON color schemes, snippet collections with static code templates, language definition extensions providing syntax highlighting grammar, icon themes, extension packs that bundle other extensions, and simple configuration extensions that rely solely on VS Code's contribution points without requiring external packages.

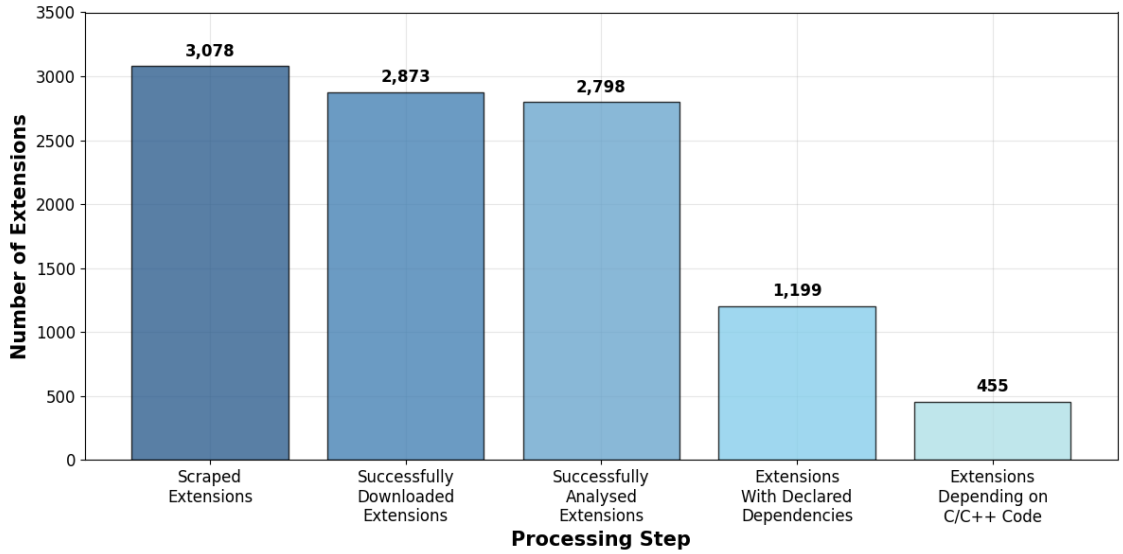

Figure 5.1: Count of VS Code extensions collected and processed during the scraping phase.

The analysis discovered 208,826 total dependency instances across all processed extensions during dependency tree traversal. Of these, 1,189 dependencies (0.6%) were skipped during processing due to missing package information in `npm`'s dependency resolution, typically indicating packages that were declared but not properly installed. The remaining 207,637 dependencies were successfully processed, though 83 dependencies could not be analysed for language usages due to missing repository information. These dependencies with incomplete language data were kept in paths leading to nodes containing native language, though their specific language composition remains undetermined in the final analysis.

5.1.2 Prevalence of Cross-Language Dependencies

Having established the scope and quality of our dataset, we now examine the prevalence of cross-language dependencies within the VS Code extension ecosystem. Of the 2,798 extensions that were successfully downloaded, 95 extensions (3.4% of downloaded extensions) contained native C/C++ code directly within their own codebase.

After pruning dependency trees to retain only branches leading to C/C++ dependencies, 455 extensions (16.3% of analysed extensions) remained, including those with native code directly in the extension. This breaks down as follows: 86 extensions contain native C/C++ code directly but have no external C/C++ dependencies, 9 extensions have both direct native code and external C/C++ dependencies (representing the overlap between categories), and 360 extensions rely on compiled languages exclusively through their dependency chains without implementing native code themselves.

The dependency analysis identified 129 extensions (4.6% of analysed extensions) that have direct dependencies on packages containing C/C++ code. However, the reach of cross-language dependencies extends much further through transitive dependency chains. A total of 369 extensions (13.2% of analysed extensions) were found to have C/C++ code somewhere in their dependency tree, meaning an additional 240 extensions are exposed to compiled languages only through transitive dependencies. This demonstrates that most cross-language exposure occurs indirectly, often without explicit developer awareness.

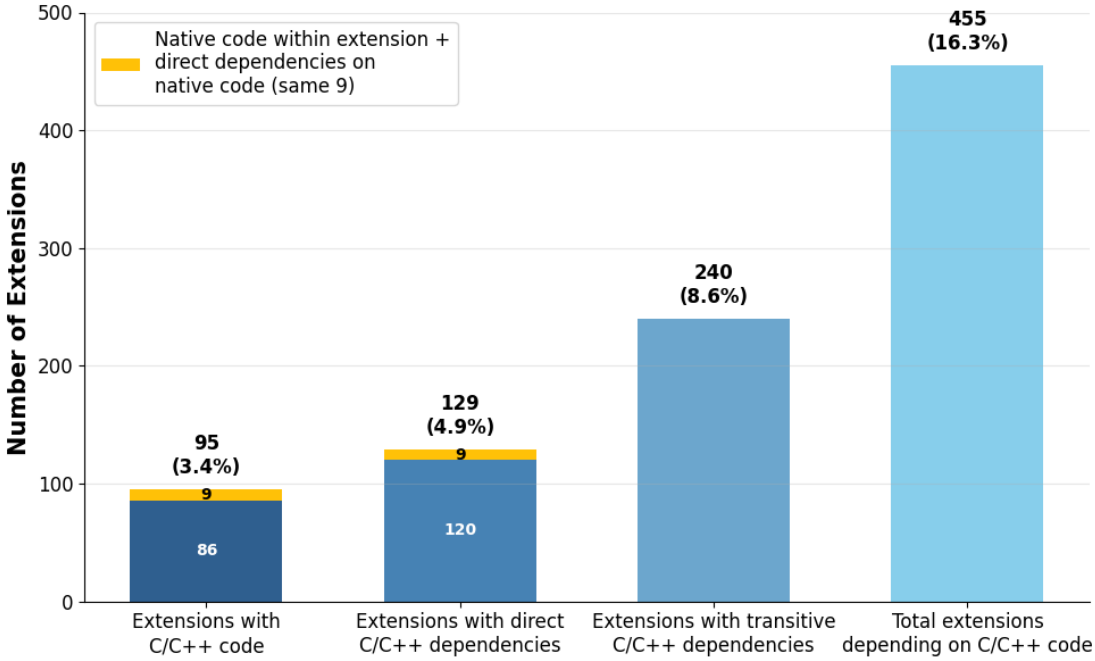

Figure 5.2: Breakdown of C/C++ use in VS Code extensions.

5.1.3 Dependency Tree Characteristics

The dependency analysis processed substantial volumes of data across the extension ecosystem. At the direct dependency level, 18,502 dependency instances were identified, representing 7,914 distinct packages by name, with 508 instances (2.7%) containing C/C++ code.

The complete dependency trees before pruning revealed the full scope of the ecosystem's interconnectedness: 208,826 total dependency instances, encompassing 36,698

unique package-version combinations across 12,235 distinct packages.

After pruning branches without native languages, the resulting dependency trees contained 6,803 dependency instances representing 1,703 unique packages, including non-native intermediaries. As expected, given the pruning algorithm's design to retain only paths leading to C/C++ dependencies, 4,283 of these instances (62.96%) contained C/C++ code directly, across 550 unique instances. However, this proportion also demonstrates that a substantial number of non-compiled dependencies (2,520 instances, 37.04%) serve as intermediary packages in the paths to native code integration, indicating that cross-language dependencies often involve multi-step dependency chains rather than use of native modules in the direct dependency.

The dependency trees exhibited significant structural complexity, with the maximum observed depth reaching 10 levels of transitive dependencies within the pruned trees. Notably, the distribution in Figure 5.3 shows an interesting pattern where transitive dependencies containing native code at depth 2 (898), 3 (1124), and 4 (965) exceed direct dependencies at depth 1 (508), indicating that many cross-language dependencies are accessed through intermediate packages rather than directly. The distribution then follows a decaying pattern with progressively fewer instances at greater depths, culminating in only 3 dependencies at the maximum depth of 10, as shown in Figure 5.3.

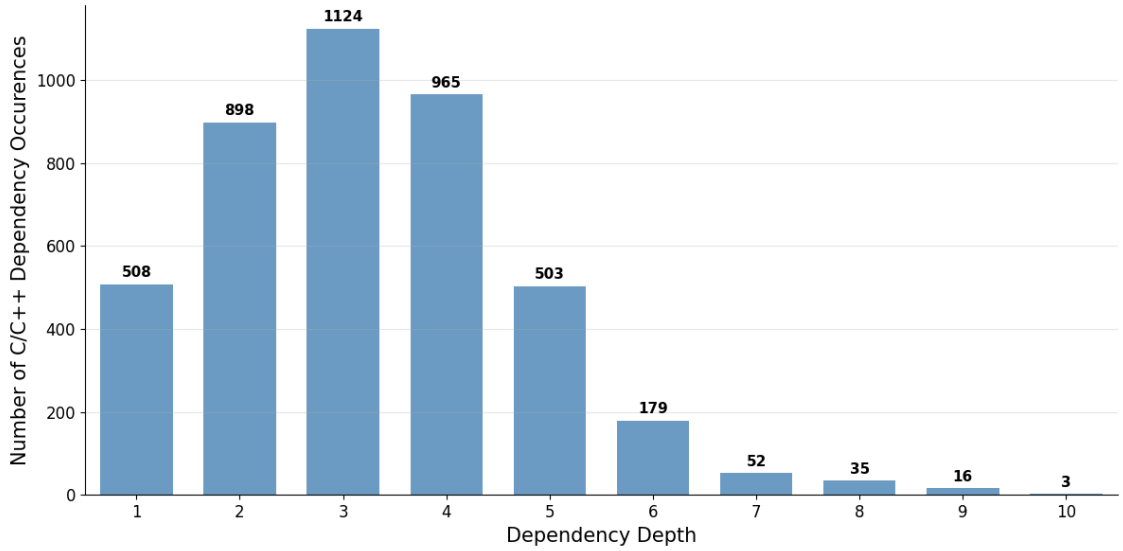

Figure 5.3: Distribution of C/C++ dependency occurrences by depth level in dependency trees

Figure 5.4 shows the distribution of dependency depths across extensions with C/C++ dependencies. The left panel shows that 108 extensions (23.7% of extensions with C/C++ dependencies) rely only on direct dependencies (depth 1) containing native code, with the frequency decreasing as depth increases. The maximum observed dependency depth reaches 10 levels, with only 3 extensions (0.7%) reaching depths of 9-10 levels.

The right panel shows the cumulative distribution of dependency usage across depth levels. 82.9% of all C/C++ dependencies are found within the first three levels, dropping to 33.1% by depth 4 and 5.5% by depth 6.

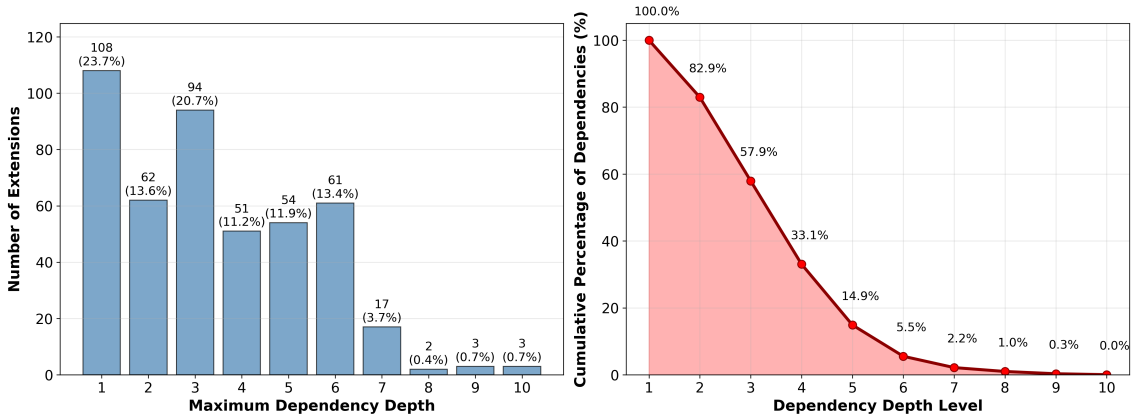

Figure 5.4: Distribution of extensions by maximum dependency depth (left) and cumulative decay of dependency usage across depth levels (right). Analysis includes only pruned dependency trees containing paths to C/C++ dependencies.

Summary of Findings for RQ1

Key Finding: Cross-language dependencies are prevalent in the VS Code extension ecosystem, affecting 455 extensions (16.3% of successfully analysed extensions).
Distribution: Only 95 extensions (3.4%) contain native C/C++ code directly, while 369 extensions (13.2%) are exposed to compiled languages through dependency chains. Most cross-language exposure (240 extensions) occurs through transitive dependencies without explicit developer knowledge.

Ecosystem Scale: The analysis processed 208,826 total dependency instances, identifying 4,283 (2.1%) containing C/C++ code across 550 distinct packages.
Structural Complexity: Dependency chains extend up to 10 levels deep, but 66.9% of cross-language dependencies occur within the first three levels, concentrating security exposure in manageable dependency relationships.

5.2 RQ2: Security Implications of Cross-Language Dependencies

Addressing RQ2, we conducted a static analysis of native code used by VS Code extensions. Our investigation examined both the extensions themselves and their dependency trees to identify potential vulnerabilities in the subject native code. Below are the general structural tendencies of native code used in VS Code extensions, split into sections regarding occurrences directly in the extensions (Section 5.2.1), and native code in their dependencies (Section 5.2.2).

Due to the shortcomings of static analysis in general, brought up in Section 2.6, among the true positives caught there will also be false positives, and there might also be false negatives missed. However, it still shows tendencies of potentially vulnerable or buggy native code present in the extensions' dependencies.

In Figure 5.5, the top 5 contributors to the potential vulnerabilities encountered are visualised, with one of the two extensions taking the top spot with 168 occurrences, despite the low number of extension entries.

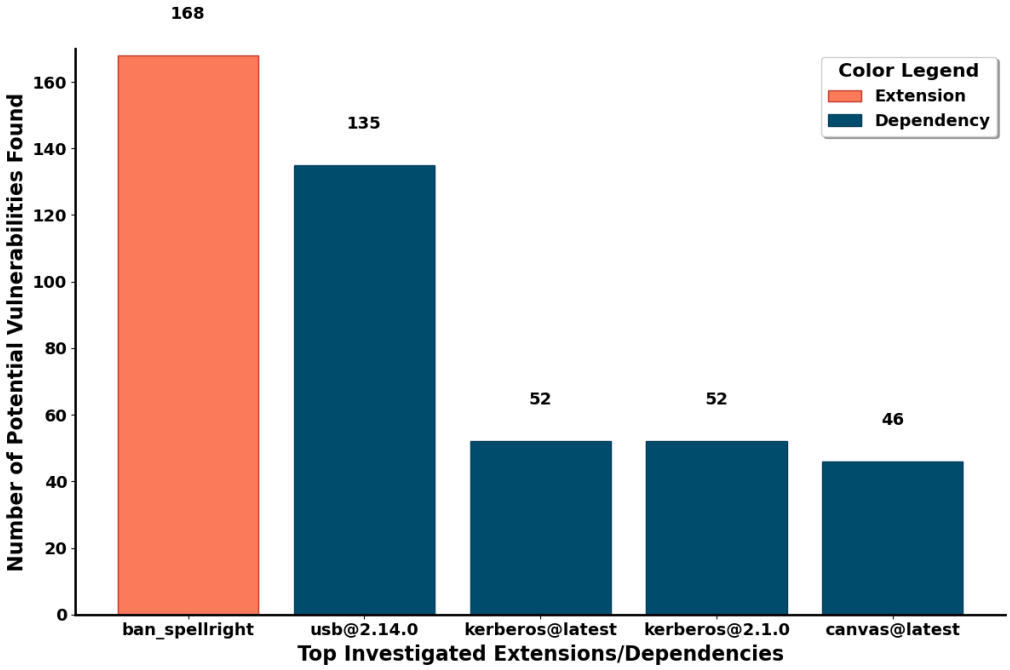

Figure 5.5: Top 5 potential vulnerability occurrences by extension or dependency in isolation, e.g., intra-package vulnerabilities.

5.2.1 Cross-Language Dependencies in VS Code Extensions

As mentioned in RQ1 (Section 5.1), we found 95 extensions containing native code within them. Of these 95, 90 only contained native code as test-code, or had misconfigured or missing `binding.gyp` files, and were thus filtered out. This filtration of misconfigured or empty `binding.gyp` files was due to not being able to build the native code using the suggested `node-gyp` commands. After this filtration, only five extensions remained; and after further filtering of non-Linux (due to incompatibility with the host computer) specific functionality, only two extensions (id: `ban.spellright` with 505,361 installs, and id: `HaxeFoundation.haxehl` with 74,133 installs) remained. Figure 5.2 shows that the overwhelming majority was in MEM-07 (96.5%).

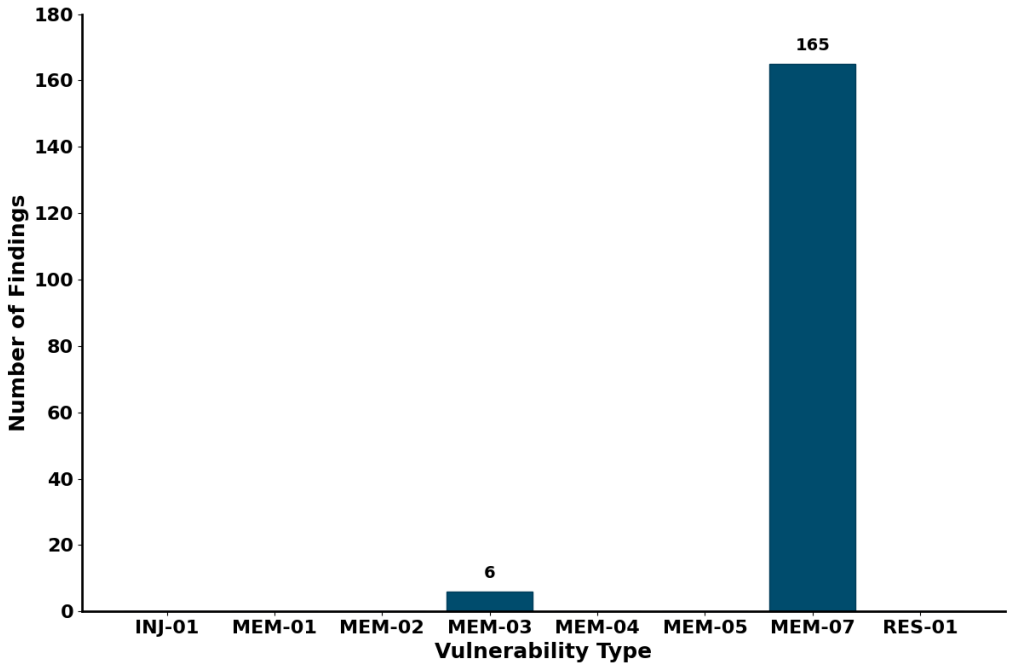

Figure 5.6: Vulnerability distribution among extensions of all the vulnerabilities listed and analysed in Table 4.1

5.2.2 Cross-Language Dependencies in `npm` Packages used by VS Code Extensions

The conducted vulnerability analysis on the dependencies reveals potential security vulnerabilities with cross-language interaction in VS Code extensions' dependencies. From the 721 unique dependencies identified and cloned during the dependency extraction process in section 4.3, 228 successfully underwent CodeQL database creation and analysis. The reduction primarily resulted from build system incompatibilities, including missing `binding.gyp` files and other build configuration requirements.

The analysis of 228 dependencies revealed 8,732 potential vulnerabilities affecting 153 dependency versions. Thus, 75 dependencies revealed no potential vulnerabilities from the conducted analysis. As illustrated in Figure 5.7, the distribution of vulnerability types demonstrates that memory-related issues dominate the security landscape, with MEM-02 representing the most prevalent category at 7626 findings (87.3% of total vulnerabilities). However, most MEM-02 occurrences came from a single dependency, `leveldown`, spanning two distinct versions, amassing 7,392 (96.9%) of all found occurrences. In Figure 5.8, a better representation is given for the general theme of the vulnerabilities. Due to this outlier dependency, we hereby exclude this package to better prove the general tendencies of cross-language dependencies, rather than only being influenced by this single instance, albeit its concerning nature.

Collectively, excluding the outlier, memory-related vulnerabilities (MEM-01 through MEM-07) account for 95.1% of all identified security issues (see Figure 5.8). The

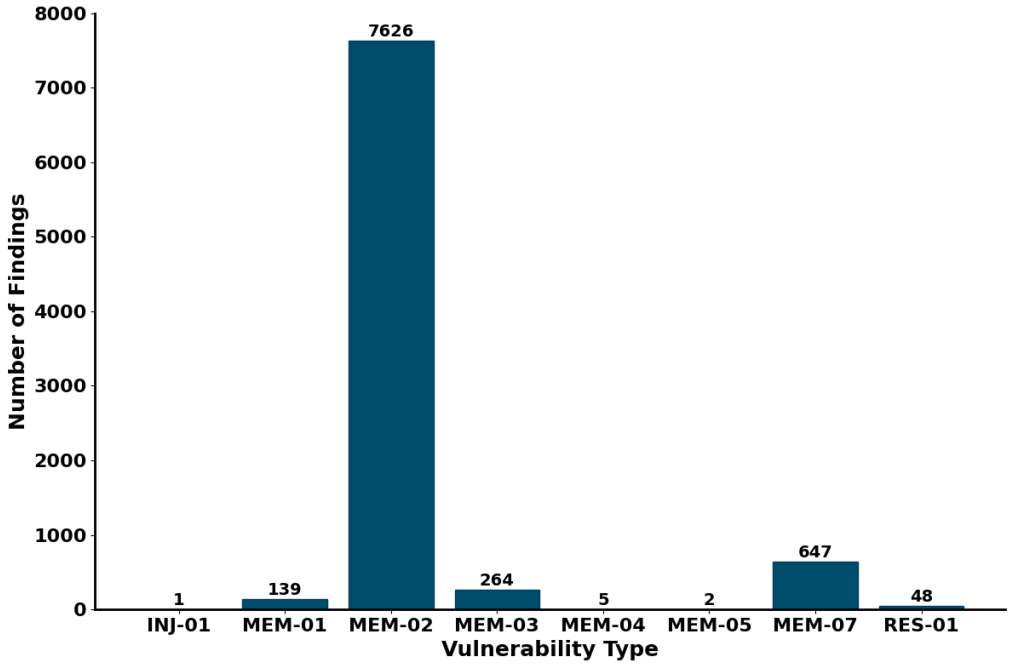

Figure 5.7: Vulnerability distribution among dependencies of all the vulnerabilities listed and analysed in Table 4.1

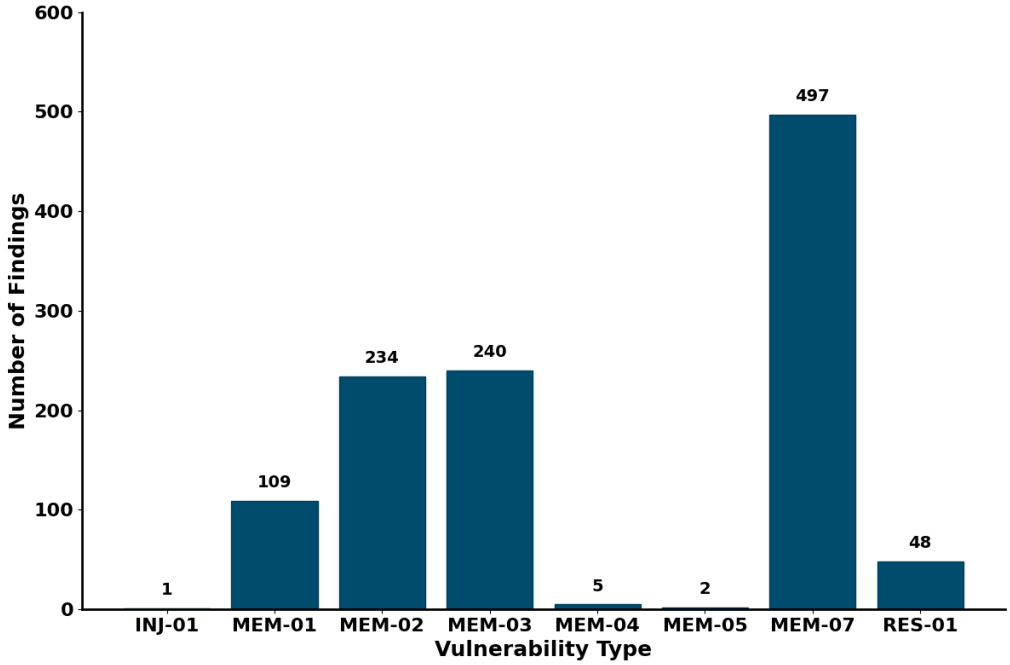

Figure 5.8: Vulnerability distribution of all the vulnerabilities listed and analysed in Table 4.1, excluding one over-represented dependency

analysis of the remaining 226 dependencies (i.e., excluding the two versions of leveldown) revealed 1,136 potential vulnerabilities affecting 151 dependencies (66.8%), indicating that approximately two-thirds of all cross-language dependencies introduce some form of security risk. RES-01 and INJ-01 represent smaller but notable categories, contributing 48 and 1 findings, respectively.

5.2.3 INJ-01 Vulnerability

The single INJ-01 vulnerability occurred in a dependency that did result in a true positive. The input to the C++ function is directly translated from the dependent JavaScript program and used without sanitisation. In Listing 5.1, we present an anonymised version of the exploitable part of the dependency to discern it from being targeted for malicious use. However, given the snippet below, we can deduce that the input from a program can be whatever a developer chooses.

```
1  extern "C" bool function(const char* s) {
2    // Other code ...
3
4    char command[1024];
5    snprintf(command, sizeof(command), "text %s.text-2/%s", s, s); //
         <- text segments here are identifiable and application
        specific, thus anonymised (same exploitability in original).
6
7    if (system(command) < 0) // INJ-01
8      return false;
9
10   return false;
11 }
```

Listing 5.1: Dependency with native command injection and potential buffer overflow.

5.2.4 Extension Impact of Cross-Language Vulnerabilities

Vulnerability Type	Total Count	Extensions Affected
INJ-01	1	1
MEM-01	826	97
MEM-02	271	15
MEM-03	615	28
MEM-04	5	3
MEM-05	8	8
MEM-07	1664	150
RES-01	42	2
Total	3432	211 (Unique)

Table 5.1: Unique occurrences of vulnerabilities for affected extensions.

The vulnerabilities found in a dependency can be used by multiple extensions, and thus, the occurrences of such a vulnerability are cumulative among all possible usages. The impact analysis presented in Table 5.1 demonstrates the effect of vulnerabilities in shared dependencies across the VS Code extension ecosystem. Each vulnerability instance in a dependency propagates to every dependent, creating a cascade of potential security risks that spreads beyond the original vulnerable component.

The data in Table 5.1 reveals distinct cumulative propagation patterns across vulnerability types. MEM-07 and MEM-01 vulnerabilities exhibit the highest impact, amassing 1,664 and 826 total vulnerability instances, respectively. Simultaneously, a few extensions dependent on packages containing MEM-02 and MEM-03 have large quantities of occurrences per extension impacted.

In Figure 5.9, we see the relationship between the number of native dependencies of a VS Code extension and the number of occurring vulnerabilities. We can see that the data points tend toward a quadratic pattern, visualised by the dotted line. The dataset is, naturally, skewed toward extensions having fewer native dependencies, yet adhering tendencies reflect the pattern found. To cement the statistical pattern of vulnerabilities, more extensions that contain more than 10 packages containing native code would have to be investigated. However, as shown, such data points are few and far between, as the use of dependencies containing native code is not homogeneous.

Figure 5.10 demonstrates a clear asymmetry in vulnerability distribution between direct and transitive dependencies within VS Code extensions. Direct dependencies (depth 1) account for 1,162 vulnerability occurrences, representing 25.3% of total vulnerabilities, whilst transitive dependencies spanning depths 2 through 9 contribute 3,439 occurrences, constituting 74.7% of all identified vulnerabilities. The distribution exhibits a decaying pattern with peak vulnerability concentration at depth 2, followed by a gradual decrease across deeper transitive relationships.

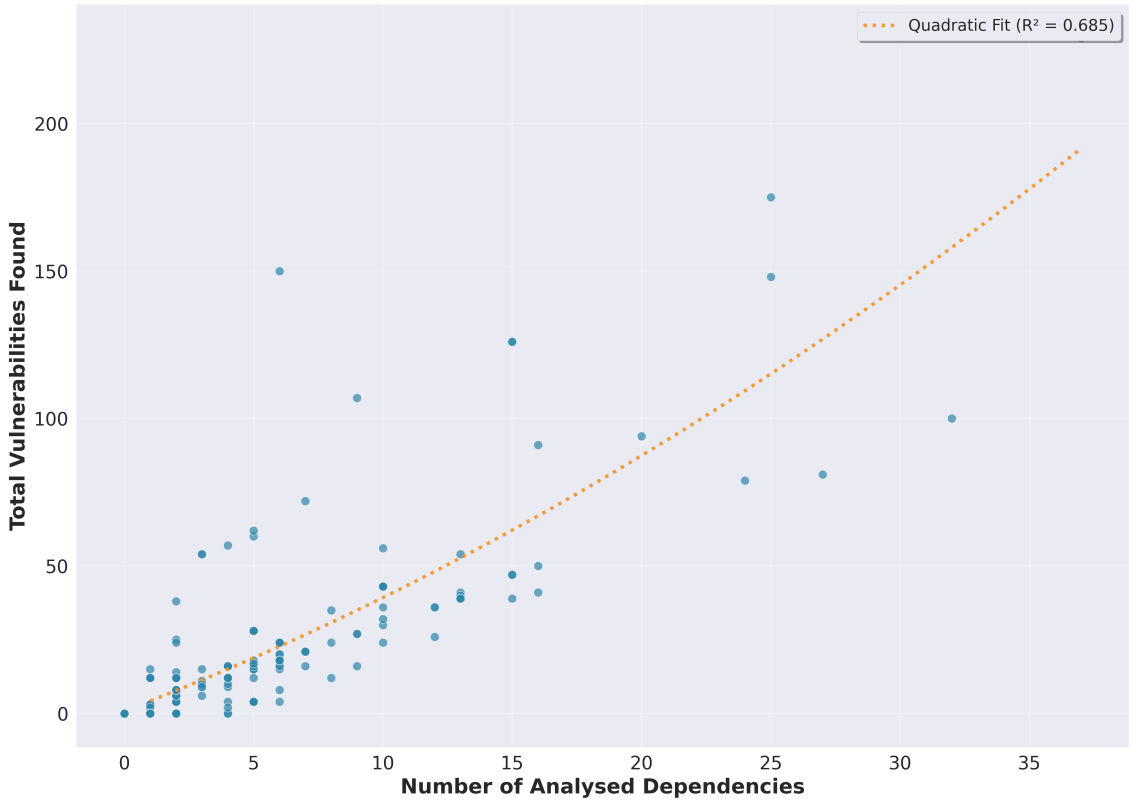

Figure 5.9: Relationship between the number of analysed dependencies and total vulnerabilities found across VS Code extensions.

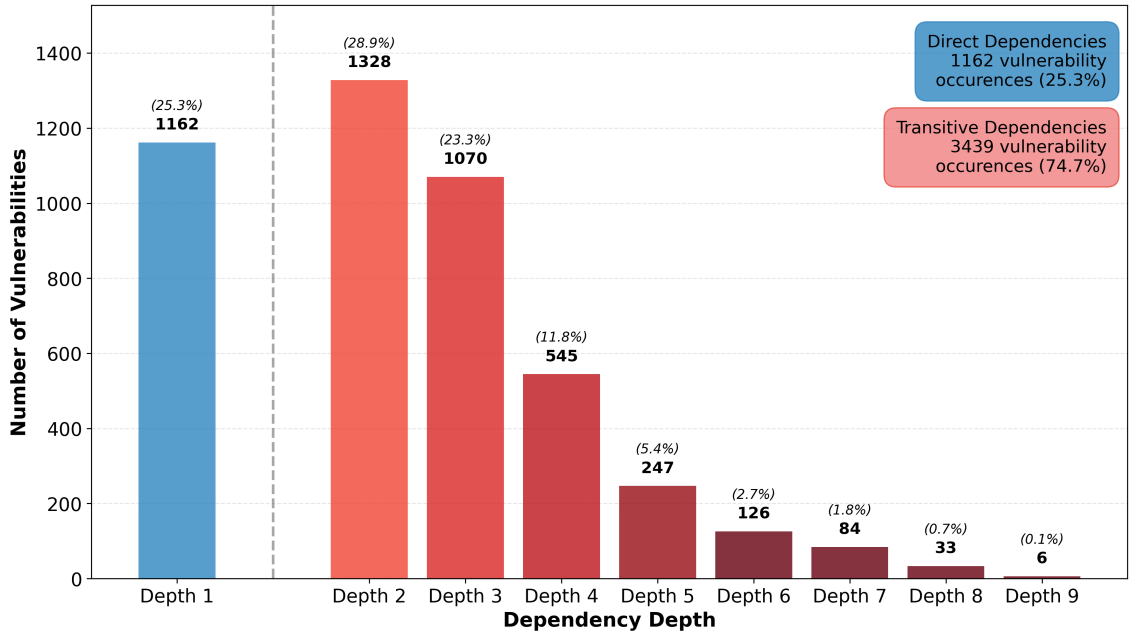

Figure 5.10: Security vulnerability distribution by dependency depth in VS Code extensions.

Summary of Findings for RQ2

Outlier: There is a clear outlier for the occurrences of potential, where two versions of a dependency (leveldown) represented 7,596 of the encountered 8,732 potential vulnerabilities.
Key Finding: Static analysis identified 1,136 potential vulnerabilities across 226 (excluding the outliers) analysed dependencies, with 151 dependencies (66.8%) introducing some form of security risk to dependent extensions.
Vulnerability Landscape: Memory-related vulnerabilities dominate, comprising 95.1% of all identified issues. Only 2 extensions contained compilable native code, whilst most vulnerabilities occurred in dependencies.

Extension Impact: Vulnerabilities propagate to 3,432 total occurrences across affected extensions, with MEM-07 (1,664 instances, 150 extensions) and MEM-01 (826 instances, 97 extensions) showing the highest impact through shared dependency usage.
Transitive Risk Concentration: 74.7% of vulnerability occurrences stem from transitive dependencies (depths 2-9) rather than direct dependencies (25.3%), with peak concentration at depth 2.

5.3 Validation

To evaluate the effectiveness of our static analysis, we created a set of simple extensions where the expected output was compared against CodeQL's results. This allowed us to verify that the generated graphs accurately captured dependencies on FFIs, cross-language bindings, and potential taint sinks. These dependencies fall into two categories based on how the cross-language interaction occurs. The first category includes a single extension that implements native code directly within its scope, using the Node-API[1]. This category will benchmark vulnerabilities studied, where each benchmark will adhere to one type of vulnerability mentioned in Table 4.1.

The second category includes a single extension that depends on external packages with cross-language implementations, both direct and transitive. To demonstrate this, an `npm` package with both direct and transitive native code was selected and incorporated into our analysis. The package, `usb`[2], directly implements the native code, and also depends on `node-addon-api` that implements its own native code. This category validates the construction of our dependency tree and finds the occurrences of native code for analysis.

Our benchmarks, which can be accessed in the reproducibility package found in Section 4.7, successfully detected all vulnerabilities purposefully planted in our benchmark extensions. The complete results and validation data are available in the package.

[1]Link to the Node-API documentation
[2]Link to the `usb npm` package

5.3.1 Category 1

To demonstrate the benchmarks' validity and the queries' robustness, we will exemplify some of them in this subsection to show the use of both benchmarks and the CodeQL queries built. As mentioned in Section 2.4, implementations using the Node-API (or NAPI) come in two flavours, either using the C-pointer based `Node-API` ABI, or using the C++ wrapped version from the `node-addon-api`[3]. The C++ wrapper package allows for a simplified implementation due to its extended functionalities that comes with C++ as a superset of the C language. It is maintained and developed by Node.js, thus closely resembling the original ABI. As both implementation strategies are equally valid, both are covered for the benchmarks, to ensure that the queries collect all possible Node-API implementations. However, even though the benchmarks are created to cover all these different cases, there is still a possibility of false negatives that can evade the analysis. This is deducted by the fact that the developer has freedom of implementation, which makes covering every possible outcome nearly impossible.

To make the benchmarks as realistic as possible for this study, one subset within this vulnerability analysis group consists of working VS Code extensions that can be used and exploited within the application. The other subgroup consists of template Node-API C implementations as stand-alone C++ files, entirely separated from the Node.js runtime. This was done to mimic the structural misuses of C-based implementations, which are still detectable for static analysis, as code samples rather than working Node modules. Both these groups create the benchmarks to evaluate the common setup of extensions containing native code.

5.3.2 Category 2

As mentioned in the introduction of this chapter, this category mainly focuses on finding the dependency structure of said benchmarks to include transitive and direct dependencies, disregarding all dependency branches that do not include native code. This is investigated within the expected confines of the extension benchmarks in this category to validate the branch-stepping of the implemented script. To properly validate this, a single extension with multiple possible dependency branches will be traversed, which covers native code in a direct dependency and a transitive dependency, in addition to a dependency that does not include any native code within its dependency tree. Thus, the expected outcome of this dependency tree will be represented by a tree similar to Listing 5.2, but with the actual extension names, dependency keys and repository-specific data.

```
1 "extension": {
2     "dependencies": {
3         "direct_dep": {
4             "languages": {...}, //<- Including either C/C
                ++
5             "contains_compiled": true,
```

[3]Link to the `node-addon-api npm` package

```
6               "has_cpp_directly": true,
7               "dependencies": [
8                   "transitive_dep": {
9                       "languages": {...}, //<- Including
                            either C/C++
10                      "contains_compiled": true,
11                      "has_cpp_directly": true,
12                      ... //More dep data
13                  }
14              ],
15              ... //More dep data
16          }
17          //No more direct dep, as other dep branch has no
                native code
18      },
19      "ext_native": false //<- No native directly in the
            extension
20  }
```

Listing 5.2: Dependency tree of the extension benchmark.

6

Discussion

Throughout this study, several choices impacted its outcome. This chapter discusses the results gathered and their impact, as well as choices made during the study, argues in favour of them, and suggests possible future angles to build on the research conducted. It also considers the threats to this research's validity and important delimitations that shaped the results.

6.1 Selection of Extensions for Analysis

As of April 28, 2025, there were 72,924 extensions available on the Visual Studio Code Marketplace. Analysing every single one of them would not only be time-consuming and computationally expensive, but also unnecessary for our purposes. Instead, we decided to focus on the top 3,078 extensions based on their total number of installs.

This decision is backed by the distribution of installs across extensions, which is heavily skewed: a small number of popular extensions account for the vast majority of installs. As shown in Figure 6.1, the top 3,078 extensions alone cover about 95.96% of all installs, which translates to over 4.16 billion installs out of a total of 4.34 billion. Beyond this point, each additional extension contributes very little in terms of user coverage.

The data used to calculate install counts was fetched directly from the Visual Studio Code Marketplace. This data was collected programmatically on April 28, 2025, and includes all publicly listed extensions available at that time.

Focusing on these most-used extensions allows us to carry out a more efficient and relevant analysis for the vast majority of VS Code users. It ensures we are capturing the extensions that are being used in practice, while avoiding spending resources on rarely used or inactive ones.

However, of the 3,078 extensions selected for analysis, 1,674 could not be processed due to missing package.json files or the absence of any declared dependencies. Upon manual inspection, we found that many of these extensions belonged to categories that do not typically include executable code or rely on external packages. These included theme extensions defined by static JSON colour schemes, snippet collec-

tions containing static code templates, language grammar definitions used for syntax highlighting, icon themes, extension packs that bundle other extensions, and configuration-focused extensions that rely solely on VS Code's built-in contribution points.

We deliberately chose not to filter out any categories in advance, relying solely on install count as our selection criterion. This decision was made by previous findings indicating that even seemingly benign extension types can be used in unexpected or potentially harmful ways [12]. Including them in the dataset ensured a broader and more representative sample of the VS Code extension ecosystem, avoiding premature assumptions about which extensions might contain relevant dependencies or code paths.

Although this excluded a portion of the dataset from the final analysis, it allowed us to maintain an unbiased starting point other than install count.

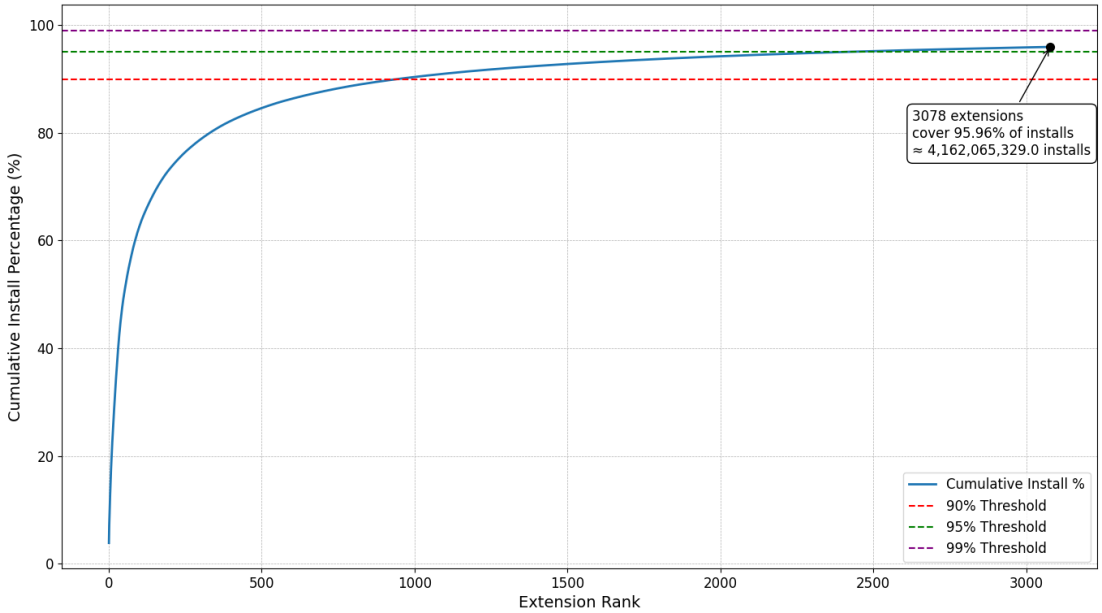

Figure 6.1: Cumulative install coverage of Visual Studio Code extensions sorted by install count.

6.2 Implications of RQ1 Results

Many extensions were filtered out by not using any dependencies. This shows that a significant portion of the extension ecosystem consists of lightweight, dependency-free extensions that provide themes or specific functionality through VS Code's built-in extension mechanisms. Although the VS Code extension ecosystem is primarily JavaScript-based, a considerable minority of extensions (over 16%) rely on compiled native modules to deliver their functionality, either through direct implementation or in their transitive dependency chains. This shows the prevalence of cross-language dependencies in the extension ecosystem.

The dependency depth analysis provides further insight into the structure of these cross-language relationships. While dependency chains can extend up to 10 levels deep, the concentration of 82.9% of C/C++ dependencies within the first three levels indicates that most cross-language relationships occur through direct or near-direct connections. This pattern suggests that the majority of potential cross-language security considerations are concentrated in dependencies that are relatively close to the extension itself, which may simplify dependency management and security assessment for extension developers.

Due to the small sample size of compilable VS Code extensions with direct native implementations, no general patterns can be drawn from this group, other than these two instances containing these potential vulnerabilities. However, this shows the relative lack of direct native exploitation among top extensions, as only 2 of 3,078 extensions (≈0.06%) had direct implementations of native code that could be compiled, and were not tests. Widening the scope of investigation to less popular extensions might open the possibility of direct malicious native exploitations. Many extension creators for the larger extensions are generally trusted actors, such as corporations like Microsoft or GitHub, and are not likely to have malicious intentions.

Combined with the dependency depth patterns, the collected data demonstrates that most cross-language integration occurs indirectly through the `npm` ecosystem at relatively shallow depths, rather than through explicit choices by extension developers to include compiled language components. This should concern developers of VS Code extensions and `npm` packages alike, as they might expose users to packages they do not know they depend on.

6.3 Implications of RQ2 Results

The vulnerabilities found in Section 5.2 show that the most frequent security challenge in cross-language dependencies stems from the complexity of managing memory across runtime boundaries, where traditional JavaScript memory safety guarantees no longer apply. This does not come as a surprise, as many of the complexities of writing native code stem from these considerations, which are generally handled automatically in higher-level languages.

The distribution of vulnerabilities in affected extensions, depending on their respective type, shown in Table 5.1, suggests that potential MEM-07 vulnerabilities are more commonly encountered than other findings, followed by MEM-01. However, both MEM-02 and MEM-03 vulnerabilities are far more prevalent in their dependent extensions, as they boast greater average inclusion in the extension where they occur.

The findings revealed in 5.10 show that most potential cross-language vulnerabilities within VS Code extensions originate from transitive dependencies, rather than direct dependency relationships. Transitive dependencies represent nearly three-quarters of the total vulnerability exposure across the analysed extension ecosystem. This

suggests that developers often introduce extension risks implicitly, without their explicit knowledge or intent.

INJ-01 Exploit

The motivating example gathered (See Listing 5.1), being the only found INJ-01 vulnerability, was investigated further to see its implications. Using a simple input for input parameter `s = "test; rm -rf /; echo"`, we exit the confines of the presumed structure, giving complete control to the implementer. The given example, however, assumes basic removal of the root's files. Removing all files in the system can harm the host, but extracting personal data and information, or downloading malicious files can be far more malevolent. Another scheme can be used as input for connecting to, sending data to, or downloading malicious payloads from a remote actor. For example, if `s = "test; curl remote.com/malware.sh | bash; echo "`, the adversary can download a malicious script from the remote server and execute it on the host system, without the knowledge of the host user.

However, by manual investigation, we found that even though the encompassing dependency is included in the dependency tree, it is not used anywhere in the only dependent package. This means that the vulnerability is never reached. Still, it shows the potential inclusion of such vulnerable and exploitable code implementations, and the lack of vetting in the ecosystem.

6.4 Comparison to Related Work

Our results partially align with earlier findings on the risks posed by native code. Similar to Staicu et al. [47], we observed that memory-related issues, such as uninitialized memory reads and potential crashes, can propagate from native components into high-level languages. However, while their study focuses on intentionally malformed inputs and missing safety checks at the language boundary, our analysis indicates that memory issues could arise from seemingly non-malicious VS Code extensions due to cross-language misuse or unsafe transitive dependencies.

Scholtes et al. [45] similarly identify memory vulnerabilities, including buffer overflows and use-after-free bugs, and go further by demonstrating exploitability in real packages. While we do not attempt exploit generation, we do observe several native extensions within VS Code extensions with comparable risks. Unlike Scholtes et al, who report issues primarily in general-purpose libraries, our work reveals that such vulnerabilities also occur in unchecked third-party additions to their development environments, often due to developers depending on low-level bindings indirectly or with limited knowledge.

6.5 Threats to Validity and Delimitations

This study was enabled by the openness of the `npm` ecosystem, the transparency of its underlying source code, and prior research in the neighbouring areas. However, some packages lack repository data, do not have a public GitHub repository, or point to the wrong repository. As the entry of this data is not mandatory nor vetted in the `package.json` file before submission to `npm`, the package developer can insert any information they choose. This was further exemplified by Assaraf [12], who used a malicious proof-of-concept extension that used the GitHub repository link of the extension that it mimicked. If the developer wishes not to share this information or keep the repository private, there is no reasonable way to check the inclusion of compiled languages for this study. Additionally, since the repository link is not vetted, the final location could be another repository, which may misrepresent the languages used in the source code. Any such mislabeling could either produce false negatives or false positives of native code inclusion, depending on the differences between the dependency's source code and the provided repository. As constructing a CodeQL database is time-consuming, we want to filter out all dependency branches of execution that do not include native code before building it. This seconds the notion that mislabeled repositories may result in false negatives, as entire branches of execution may be filtered out before the databases are created.

Some extensions, such as Pylance[1] do not, surprisingly, include any `package.json` metadata file at all, making the dependency structure difficult to traverse. For the above example, this is due to the extension not being open-source; thus, the source code is not disclosed. Such cases are not evaluated in this study.

Although not widely adopted, a developer can theoretically invoke native code through an intermediary layer in an additional language. For example, a native extension can theoretically be wrapped within a compile `Rust` module, as the JavaScript code imports and uses the compiled module, not caring about the underlying language used. This implementation does not have direct interaction between JavaScript and native code, but lets Rust act as the middleman. Such occurrences will not be parsed or evaluated in this study, as they fall outside the scope; however, they can still produce similar issues, as described by Mergendahl et al. [37].

6.6 Encountered Misuse Preventions

All actors included in the VS Code structure, e.g. Node.js, Electron, and VS Code itself, strive to keep their users safe. However, completely safeguarding the users is difficult and breaks the openness and modularity of VS Code's philosophy. Several attempts to prevent malicious intent have been encountered during this study, both whilst creating the benchmarks and when researching the underlying architecture. For example, Node.js continuously deprecate vulnerable and buggy code from their documentation and the runtime itself [38]. Their current latest ver-

[1] Link to the Pylance extension

sion, Node 24.0.0 (released 2025-05-06) includes deprecation of passing arguments to `node:child_process`' `spawn` and `execFile` functions containing the `shell`-option set to `true` to combat potential shell injection. A discovery from this is that the native-side of such consequences is not mentioned, leaving the possibility for shell injection if executed by a native-implemented equivalent (brought up as `INJ-01` in Table 4.1).

The C++ language outputs warnings to the user when using historically unsafe and vulnerable functions, e.g `sprintf` and `gets`, when compiling the native code. This warning lets the developer of the native extension know that the used function is indeed problematic. However, that warning is not presented to the end-user of a dependency or extension that uses such native extensions.

6.7 Recommendations

To improve the security of VS Code extensions that rely on cross-language interaction, we recommend that users exercise caution when installing third-party extensions. Due to the open nature of the VS Code extension ecosystem, numerous potential attack vectors may place users at risk. This risk is particularly high for extensions incorporating native languages such as C or C++, as these can introduce a greater number of, and often unforeseen, vulnerabilities, as discussed throughout this thesis.

From a security perspective, we advocate for the creators of VS Code to introduce sandboxing for all extensions running inside the extension host process, as is done for the renderer process, shown in Figure 2.1. Such a sandbox could restrict access to the underlying system by exposing only selected APIs through a strict, permission-based model. This would limit cross-language extensions from accessing sensitive system resources, such as executing shell commands with privileges inherited from the user, unless explicitly granted.

However, it is also important to acknowledge that the architecture of VS Code is probably intentionally open, since this enables small-scale developers to create extensions without requiring extensive domain knowledge. This openness helps to create a large and active community, encouraging more developers to contribute and, in turn, attracting more users to the platform due to the wide variety of available extensions. To strike a balance between a fully open and a fully restricted model, we recommend more transparent communication from the VS Code Marketplace to the users and developers. By collaborating with `npm` to retrieve both direct and transitive dependencies, Microsoft could display a complete Software Bill of Materials (SBOM) to users before extension installation. This would help users make more informed decisions about what they are installing. Additionally, based on these dependency trees, we suggest closer engagement with extension developers, for example, issuing warnings if any dependency in their SBOM is flagged as vulnerable by automated tools such as SNYK, or if it contains native code written in C or C++, as this may warrant a more thorough security review.

6.8 Future Works

Despite our best efforts, the vulnerability analysis itself does not convince us enough to trust it entirely. As discussed, the risk of false negatives led us to weaken the constraints to try to catch all possible positives. However, this can come at the expense of a significant number of false positives, thus complicating the identification of the true positives due to the large quantities of data. In retrospect, we believe using an existing static analysis tool, such as Snyk, would be beneficial in exploring the native vulnerabilities. Other analysis methods could also be considered for more robust identification of vulnerabilities, using either dynamic analysis or machine learning models.

One consideration for this study is the prevalence of the attempted call-graph creation for both cross-dependency and cross-language reachability from an extension. Due to the time constraints and open nature of the `npm` ecosystem, the complete pipeline for constructing the call-graphs was not finalised. This means that, instead of mainly analysing the reachable vulnerabilities from an extension, all vulnerabilities within any dependency, direct or transitive, are examined. Creating call-graph queries took far longer than expected due to all possible ways of tracking the data, particularly in JavaScript, and between languages. For example, when tracking the cross-dependency function calls, the dependencies are imported using either the `import` or the `require` keyword. However, these can be imported in multiple ways, and most need to be explicitly handled. Finding the entry point functions of a dependency requires the entry point files and all the exported functions from those files. The entry point files are declared in the `package.json`, but depending on the dependency's setup, the entry point files may differ. For example, a TypeScript dependency that needs compilation to JavaScript code before use may exclude the actual compiled output directory from the source code, resulting in compilation before accessing the actual entry point. To access the entry point file, we must compile the TypeScript into JavaScript. However, the compilation script can be done in any way the developer deems fit, and can be declared as any alias in the `package.json` file. Some encountered examples include bundling tools, such as webpack, that present the compiled package as a single JavaScript file, with a structure that CodeQL dataflow cannot follow at all. The functions exported from an entry point can be implemented in numerous ways and re-exported under aliases from other files, obfuscating the tracking of the original function via its different representations.

All aforementioned complications of data tracking between dependencies are but one example of many encountered during this study, underscoring the complexity of analysing the landscape in its entirety.

A study on the whole ecosystem using a similar approach would be beneficial in concluding cross-language use with certainty. However, this would require vast data handling for extracting all 72,000+ extensions and their potential use of millions of dependency versions.

6.8.1 Migration to other tools than CodeQL

As encountered in this study, the flow of information is complicated to track in the cross-language boundary using CodeQL, as highlighted by Zhu et al. [50], and the cross-package tracking. In retrospect, using Joern and Google Closure Compiler proved effective for Staicu et al. [47], and could have shown prominence in this study. However, during the making of this study, CHARON [45] has been developed, claiming to be the first to conduct cross-language analysis (unless glueing multiple single-language analyses together) using a polyglot approach. If functional as described by its authors, this tool would show prominence in tracking the intra-package data; however, for the tracking between packages, the same type of glueing would be required to find the actual reachability from an extension.

6.8.2 Dynamic Analysis

In addition to challenges in correctly analysing the structure of the extension execution, challenges with static vulnerability analysis in general are prevalent. As mentioned in Section 2.6, any bugs or vulnerabilities found by the CodeQL queries can be false positives, thus not propagating through the host system as structurally revealed.

To reliably find all potential security issues (bugs and vulnerabilities) within the native source code, testing the occurrences at runtime is necessary. Static analysis offers ways to reason about code without execution, but it cannot analyse specific runtime behaviours. This is particularly relevant when analysing native components whose behaviour might depend on runtime states, dynamically loaded resources, or system-level interactions. This static analysis was essential for constructing the call-graph, as this study builds the entire dependency structure of extensions. However, this was cut short due to the time constraints and the complexity of creating such call-graphs in interlanguage dependencies, which will further be discussed in Section 6.8. This was why CodeQL was chosen initially: to enable both call-graph creation and vulnerability analysis using a single tool. Adding dynamic analysis would require a different methodology, including controlled execution environments, input generation strategies, and instrumentation of both JavaScript and native components, adding significant complexity. Nonetheless, it does present an interesting direction for future work, especially by evaluating and validating statically found interlanguage dependencies with monitored execution traces. This aligns with Huang et al. [29], who showed that using dynamic analysis complements static analysis by expanding the range of detection and potentially identifying false negatives from the static analysis.

7

Conclusion

This thesis explored the extent and nature of cross-language dependencies in the VS Code extension ecosystem, focusing on native vulnerabilities and their security implications in the domain of VS Code. Our findings show the possibility of vulnerabilities and the implicit trust developers place in dependencies that they do not explicitly trust. Although few extensions had native code implemented directly (only 95 of 3,078), 361 additional extensions had dependencies, direct or transitive, up to a depth of 10, on `npm` packages containing native code. However, in the occurrences of native code directly in the extensions, only 2 contained compilable, non-testing code for Linux systems. We also find that among the top 3,078 extensions, amassing >95% of all total extension installs, this vector of attack is not directly made in a large-scale fashion by implementation in the extensions themselves. Instead, potential cross-language vulnerabilities occur in cross-language code imported via package dependencies. This demonstrates that cross-language integration occurs primarily through the `npm` ecosystem rather than explicit developer choices. Our conducted static analysis identified 1,136 potential vulnerabilities across 151 dependency versions, with memory-related issues (MEM-01 through MEM-07) comprising 95.1% of findings. Critically, 74.7% of potential vulnerabilities originate from transitive rather than direct dependencies, indicating that security risks are introduced mainly implicitly. As shown in the extension benchmarks of this study, there are possibilities to exploit extensions for low-level code execution. The analysis demonstrates that cross-language dependencies represent a significant component of the extension ecosystem, with the majority occurring through transitive dependency chains rather than deliberate usage by extension developers. We conclude that the gap in research on which this research was constructed demands further work, exploring broader patterns of the VS Code marketplace with a more robust and precise vulnerability analysis tooling.

As software systems increasingly rely on complex dependency chains spanning multiple programming languages, understanding and mitigating these cross-language security risks becomes critical for ecosystem security. The findings underscore the need for enhanced tooling, improved security practices, increased package vetting, and continued research into cross-language dependency security across software ecosystems beyond VS Code.

Bibliography

[1] Extension runtime security [Online]. Accessed: Mar. 28, 2025. Available: https://code.visualstudio.com/docs/configure/extensions/extension-runtime-security.

[2] How To Run Javascript In Your Browser [Online]. Accessed: Mar. 28, 2025. Available: https://www.lighthouselabs.ca/en/blog/how-to-run-javascript-in-your-browser.

[3] Introduction | Electron. Accessed: Feb. 19, 2025. Available: https://electronjs.org/docs/latest.

[4] microsoft/vscode-extension-samples [Online]. Accessed: May. 22, 2025. Available: https://github.com/microsoft/vscode-extension-samples/tree/main/helloworld-minimal-sample.

[5] Node.js vs Deno vs Bun: Comparing JavaScript Runtimes | Better Stack Community [Online]. Accessed: Apr. 23, 2025. Available: https://betterstack.com/community/guides/scaling-nodejs/nodejs-vs-deno-vs-bun.

[6] Technology | 2024 Stack Overflow Developer Survey [Online]. Accessed: Mar. 19, 2025. Available: https://survey.stackoverflow.co/2024/technology.

[7] What is an IDE? - Integrated Development Environment Explained - AWS [Online]. Accessed: Feb. 05, 2025. Available: https://aws.amazon.com/what-is/ide/.

[8] What Is Static Analysis | Perforce [Online]. Accessed: May. 22, 2025. Available: https://www.perforce.com/blog/sca/what-static-analysis.

[9] JavaScript Statistics: Latest Usage Insights and Trends 2025 [Online], June 2022. Accessed: Mar. 28, 2025. Available: https://radixweb.com/blog/top-javascript-usage-statistics.

[10] Choosing the Right Node.js Package Manager in 2024: A Comparative Guide [Online], February 2024. Accessed: Mar. 04, 2025. Available: http://nodesource.com/blog/nodejs-package-manager-comparative-

guide-2024.

[11] npm/node-semver, June 2025. Accessed: Jun. 5, 2025. Available: https://github.com/npm/node-semver.

[12] Amit Assaraf. 1/6 | How We Hacked Multi-Billion Dollar Companies in 30 Minutes Using a Fake VSCode Extension [Online], October 2024. Accessed: Apr. 05, 2025. Available: https://blog.extensiontotal.com/the-story-of-extensiontotal-how-we-hacked-the-vscode-marketplace-5c6e66a0e9d7.

[13] Sruthi Bandhakavi, Nandit Tiku, Wyatt Pittman, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vetting browser extensions for security vulnerabilities with VEX. *Communications of the ACM*, 54(9):91–99, September 2011.

[14] Fraser Brown. Short Paper: Superhacks: Exploring and Preventing Vulnerabilities in Browser Binding Code. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 103–109, Vienna Austria, October 2016. ACM.

[15] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, May 2017. ISSN: 2375-1207.

[16] Visual Studio Code. Additional components and tools [Online]. Accessed: Apr. 13, 2025. Available: https://code.visualstudio.com/docs/setup/additional-components.

[17] Visual Studio Code. Documentation for Visual Studio Code [Online]. Accessed: Apr. 13, 2025. Available: https://code.visualstudio.com/docs.

[18] VS Code. Extension Anatomy [Online]. Accessed: May. 04, 2025. Available: https://code.visualstudio.com/api/get-started/extension-anatomy.

[19] Cay S Cornell, Gary;Horstmann. *Core Java 2 Advanced features*, volume 2. Pearson, tenth edition edition, 2016.

[20] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 181–191, New York, NY, USA, May 2018. Association for Computing Machinery.

[21] Domenic Denicola. Node.js — Peer Dependencies [Online]. Accessed: May. 04, 2025. Available: https://nodejs.org/en/blog/npm/peer-dependencies.

[22] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexan-

dros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupe, and Yan Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society.

[23] Shehan Edirimannage, Charith Elvitigala, Asitha Don, Wathsara Daluwatta, Primal Wijesekera, and Ibrahim Khalil. Developers are victims too : A comprehensive analysis of the vs code extension ecosystem, 11 2024.

[24] GitHub. About codeql. Accessed: May. 31, 2025. Available: https://codeql.github.com/docs/codeql-overview/about-codeql.

[25] Github. About CodeQL queries — CodeQL [Online]. Accessed: Mar. 26, 2025. Available: https://codeql.github.com/docs/writing-codeql-queries/about-codeql-queries.

[26] GitHub. CodeQL [Online]. Accessed: Mar. 04, 2025. Available: https://codeql.github.com.

[27] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E. Eghan, and Bram Adams. On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (jni). *IEEE Transactions on Reliability*, 70(1):428–440, 2021.

[28] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association.

[29] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. Donapi: malicious npm packages detector using behavior sequence knowledge mapping. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[30] Zihao Jin, Shuo Chen, Yang Chen, Haixin Duan, Jianjun Chen, and Jianping Wu. A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society.

[31] JSConf. Ryan dahl: Node js [Online]. Accessed: Feb. 27, 2025. Available: https://code.visualstudio.com/api/get-started/your-first-extension.

[32] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. pages 641–654, 2014.

[33] Gurpreet Kaur. Top 34 JavaScript Stats You Must Know in 2024 [Online],

June 2024. Accessed: Mar. 28, 2025. Available: https://bigohtech.com/top-javascript-statistics.

[34] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William K. Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *CoRR*, abs/1811.00918, 2018.

[35] Elizabeth Lin, Igibek Koishybayev, Trevor Dunlap, William Enck, and Alexandros Kapravelos. UntrustIDE: Exploiting Weaknesses in VS Code Extensions. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society.

[36] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. Investigating Managed Language Runtime Performance: Why {JavaScript} and Python are 8x and 29x slower than C++, yet Java and Go can be Faster? pages 835–852, 2022.

[37] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-Language Attacks. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-259/.

[38] Node.js. Deprecated APIs | Node.js v24.0.0 Documentation [Online]. Accessed: May. 07, 2025. Available: https://nodejs.org/api/deprecations.html.

[39] Node.js. Modules: Packages | Node.js v23.11.0 Documentation [Online]. Accessed: May. 04, 2025. Available: https://nodejs.org/api/packages.html#package-entry-points.

[40] Node.js. Node-api | Node.js v24.0.1 Documentation [Online]. Accessed: May. 07, 2025. Available: https://nodejs.org/api/n-api.html.

[41] npm. package.json | npm Docs [Online]. Accessed: May. 04, 2025. Available: https://docs.npmjs.com/cli/v11/configuring-npm/package-json/.

[42] npm, Inc. npm-ls, 2021. Accessed: May. 31, 2025. Available: https://docs.npmjs.com/cli/v7/commands/npm-ls.

[43] Nikolaos Pantelaios, Nick Nikiforakis, and Alexandros Kapravelos. You've Changed: Detecting Malicious Browser Extensions through their Update Deltas. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–491, Virtual Event USA, October 2020. ACM.

[44] Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1195–1205, New York, NY, USA, October 2024. Association for Computing Machinery.

[45] Raoul Scholtes, Soheil Khodayari, Cristian-Alexandru Staicu, and Giancarlo Pellegrino. CHARON: Polyglot code analysis for detecting vulnerabilities in scripting languages native extensions, June 2022.

[46] AALA IT Solutions. Package Managers: A face-off (npm vs. pnpm vs. Yarn vs. Bun) [Online], February 2025. Accessed: Mar. 04, 2025. Available: https://medium.com/@AALA-IT-Solutions/package-managers-a-face-off-npm-vs-pnpm-vs-yarn-vs-bun-d3375683fbcb.

[47] Cristian-Alexandru Staicu, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages. pages 6133–6150, 2023.

[48] H.H. Thompson. Why security testing is hard. *IEEE Security & Privacy*, 1(4):83–86, July 2003.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017.

[50] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Xiaofei Xie, and Rui Chang. My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 486–503, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[51] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. pages 995–1010, 2019.

A

Appendix

```
Request:
    curl -L \
    -H "Accept: application/vnd.github+json" \
    -H "Authorization: Bearer <YOUR-TOKEN>" \
    -H "X-GitHub-Api-Version: 2022-11-28" \
    https://api.github.com/repos/OWNER/REPO/languages

Response schema:
{
  "title": "Language",
  "description": "Language",
  "type": "object",
  "additionalProperties": {
    "type": "integer"
  }
}

Response example:
{
  "C": 78769,
  "Python": 7769,
  "JavaScript": 5980
}
```

Listing A.1: `curl`-command for fetching the languages of a GitHub repository, and the corresponding Response.