UntrustIDE: Exploiting Weaknesses in VS Code Extensions

Elizabeth Lin, Igibek Koishybayev, Trevor Dunlap, William Enck, and Alexandros Kapravelos
North Carolina State University
{etlin, ikoishy, tdunlap, whenck, akaprav}@ncsu.edu

***Abstract*—With the rise in threats against the software supply chain, developer integrated development environments (IDEs) present an attractive target for attackers. For example, researchers have found extensions for Visual Studio Code (VS Code) that start web servers and can be exploited via JavaScript executing in a web browser on the developer's host. This paper seeks to systematically understand the landscape of vulnerabilities in VS Code's extension marketplace. We identify a set of four sources of untrusted input and three code targets that can be used for code injection and file integrity attacks and use them to design taint analysis rules in CodeQL. We then perform an ecosystem-level analysis of the VS Code extension marketplace, studying 25,402 extensions that contain code. Our results show that while vulnerabilities are not pervasive, they exist and impact millions of users. Specifically, we find 21 extensions with verified proof of concept exploits of code injection attacks impacting a total of over 6 million installations. Through this study, we demonstrate the need for greater attention to the security of IDE extensions.**

I. Introduction

Software developer workstations are a classic [12] and continuing [20] target for software supply chain attacks. Unlike typical enterprise hosts, developer hosts commonly require a range of tools and applications for developers to be effective at their jobs. One such application is the integrated development environment (IDE). According to the 2022 Stack Overflow developer survey [34] with more than 70 thousand responses, the Visual Studio Code (VS Code) IDE is used by more than 74% of developers. One factor contributing to VS Code's popularity is its vibrant marketplace [39] containing a wide-range of extensions for users to customize their environment. Extensions can be developed and published by anyone, leading to a rapid growth in the number of extensions. Over the nine month period of September 2022 to May 2023, the VS Code marketplace grew from 39 thousand to more than 47 thousand extensions.

VS Code is built on Electron, a framework that embeds Chromium and Node.js into a standalone application. Therefore, VS Code extensions are similar to Node.js applications. The extensions can import dependencies from the node package manager (npm), which contains over a million packages. Unlike web browser extensions, VS Code extensions are not sandboxed. Extensions have full access to view and modify the VS Code user interface. They also execute with the same privileges as the VS Code application and have the ability to execute shell commands, read and write user files, establish network connections, and create network servers that listen for incoming connections.

While prior work has extensively studied vulnerabilities in Node.js, npm, JavaScript, and Electron [7], [16], [19], [23], [29], [43], [44], vulnerabilities in VS Code extensions have been largely unstudied. Only a few blog posts [38], [11], [26] have discussed VS Code extension vulnerabilities, identifying vulnerabilities in only a handful of extensions.

VS Code extensions have a different threat model than traditional JavaScript and Node.js applications. Many of the existing rules in static application security testing (SAST) tools do not apply. For example, input from the user is inherently trusted. Cross-site scripting (XSS) and similar detection rules that perform taint analysis from user interface fields will only produce false positives. In contrast, VS Code extensions read files in potentially untrusted code repositories. Some VS Code extensions also create web servers that listen to network ports on localhost. If the user visits a malicious website in their browser, the website JavaScript can connect to the VS Code extension. From an attacker goal perspective, we primarily consider code injection attacks leading to arbitrary code execution, e.g., via a shell command, a JavaScript `eval()`, or writing to a file such as `.bashrc` that is executed. We also consider attacks that impact file integrity, e.g., writing arbitrary content to arbitrary files, which may or may not be in a code repository.

In this paper, we perform a systematic study of vulnerabilities of extensions in the VS Code marketplace. To perform our analysis, we implemented 12 custom SAST rules for CodeQL that are tailored to the VS Code extension threat model. Specifically, we identify four attack vectors (taint sources) and three attack targets (taint sinks). The sources and sinks are not always single method calls and require additional program context to identify. We used our custom SAST rules to perform an ecosystem-wide study of 43,436 VS Code extensions collected during January 2023. We note that our snapshot included 43,436 extensions, but only 25,402 included JavaScript. The remaining extensions only modified the VS Code user interface.

We make the following contributions in this paper.

- *We identify a threat model for VS Code extension vulnerabilities.* We encoded the threat model as a series of 12 CodeQL taint analysis rules, consisting of four taint sources and three taint sinks.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24073
www.ndss-symposium.org

- *We identified and verified code execution vulnerabilities in 21 extensions that amount to over 6 million installations*. Data read from VS Code workspace settings and files are the most common source of code injection vulnerabilities in VS Code extensions.
- *We show the impact of the Node.js ecosystem on VS Code extensions.* We discovered 13,655 VS Code extensions where each one has more than 100 npm transitive dependencies. Furthermore, 9,710 extensions depend on vulnerable npm packages with a *critical-level* advisory.

The remainder of this paper proceeds as follows. Section II provides background on VS Code extensions. Section IV describes our analysis rules. Section V characterizes our dataset. Section VI evaluates our results. Section VII discusses additional considerations. Section VIII describes related work. Section IX concludes.

Availability: Our CodeQL rules are publicly available at https://github.com/s3c2/UntrustIDE. More details are available in the appendix.

II. Background and Motivation

Visual Studio Code: The VS Code IDE is built using the Electron framework [33], which uses Chromium and Node.js to create desktop applications using JavaScript, HTML, and CSS. Much of VS Code's popularity can be attributed to its vibrant marketplace of extensions. Not all VS Code extensions contain code. For example, many theme and snippet extensions use values in the extension manifest file to customize VS Code. For example, the *Dracula Official*[1] extension does not include JavaScript code, but includes JSON files that specify the colors for its color theme.

However, many extensions include code that is executed as child processes of the main VS Code application process. VS Code extensions are written in JavaScript or TypeScript. They can use the built-in Node.js modules as well as import any npm package. VS Code also provides a `vscode.commands` interface that allows extensions to manipulate VS Code itself (e.g., selecting windows and adding comments to code). Similar to Node.js applications, VS Code extensions define a `package.json` manifest file [9]. We discuss two configuration settings relevant to our discussion. The *dependencies* configuration specifies the name and version of imported npm packages. The *activationEvents* configuration specifies an array of events that cause the extension to execute. Developers can define a range of different activation events, including `onLanguage`, `onCommand`, and `workspaceContains`, which can be parameterized for more fine-grained control. The manifest also includes triggers for VS Code startup (wildcard (`*`)), or some time after startup (`onStartupFinished`). When any activation event is triggered, VS Code calls the `activate()` function for the extension, which must demultiplex the event.

VS Code *workspaces* define the build environment for projects. Users can define per-workspace configuration of VS Code via a `settings.json` file placed in the root of the workspace directory. Adding a `settings.json` file to a project's Git repository is a common way to manage per-project configuration. VS Code extensions also leverage the `settings.json` file. Extensions can programmatically access the settings via the `workspace` object.

Finally, workspace trust is a functionality provided by VS Code that allows the user to decide whether to allow code to be executed [42]. A workspace can be in trusted mode or restricted mode. A prompt appears when the user opens a folder in VS Code, asking if the user trusts the authors of the folder. Two options are given. Clicking "Yes, I trust the authors" goes into trusted mode and allows all features of VS Code to run, including extensions, debugging, tasks, etc. Selecting "No, I don't trust the authors" makes VS Code go into restricted mode and limit extension functionality and disables debugging and tasks. Developers of VS Code extensions can specify whether they support workspace trust and how their extension behaves in restricted mode. If a developer does not specify, the extension does not support restricted mode by default and is disabled. Using restricted mode could prevent malicious actors from targeting extensions and performing code injection; however, entering restricted mode removes extension functionality and impacts a developer's normal workflow and defeats the purpose of installing extensions in VS Code in the first place.

```
function exploit(port) {
  const socket = new WebSocket(`ws://127.0.0.1:${
    port}`);
  socket.addEventListener('open', () => {
      socket.send(JSON.stringify({
          type: 'external_link',
          url: 'file://${path}/$Contents/MacOS/
    Calculator',}));
});}
```

Listing 1. Synk's proof-of-concept exploit for the LaTeX Workshop VS Code Extension [26]

Motivating Example: In 2021, Snyk [26] identified a vulnerability in the popular *LaTeX Workshop* extension for VS Code. The extension starts a local web server and opens a port whenever the user opens a *.tex* file. Users commonly have web browsers open at the same time as VS Code, which allows any open website to make a connection to the LaTeX Workshop extension. Listing 1 shows the proof-of-concept exploit developed by Snyk. As shown in the listing, by passing a JSON object to the socket, an attacker is able to start the calculator application.

III. Threat Model

The goal of the adversary is to steal secrets from or modify source code on a developer workstation as part of a software supply chain attack. For the purposes of this paper, we consider achieving arbitrary code execution within a VS Code extension as a sufficient precondition to execute such an attack. Modifying files can also achieve this goal, either by modifying a file that is later executed (e.g., `.bashrc`) or modifying the target source code directly.

Figure 1 depicts four entry points that adversaries can use for their attacks. First, the attacker has *control of the code*

[1] `https://marketplace.visualstudio.com/items?itemName=dracula-theme.theme-dracula`

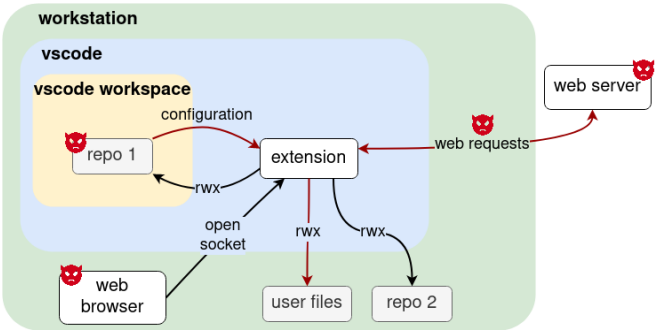

Fig. 1. VS Code extension threat model with assets and actors

repository for the active VS Code workspace (i.e., `repo 1` in Figure 1). From this position, the attacker can control the workspace settings or other files to inject malicious strings into the VS Code extension. Second, the attacker is executing JavaScript in the *web browser*. As seen in the motivating example in Section II, some VS Code extensions run web servers, which can be attacked from any process executing on the developer's workstation (i.e., the web browser in Figure 1). Third, the attacker is situated *on-path to the web server* of a network connection made by the VS Code extension. While ideally VS Code extensions should always use HTTPS, a VS Code extension developer may insecurely use HTTP. Finally, the attacker is *on the web server itself*. While we trust the web for URLs defined by the VS Code extension developer, an attack may influence the URL used for the web request to redirect the extension to a malicious web server.

An attacker can be present in multiple locations and construct a multi-stage attack by linking multiple modifiable values passed to the extension. We demonstrate an example of such an attack that follows the red arrows in Figure 1. An extension first reads configuration files from `repo 1`, which includes malformed values from the attacker. The configuration values are then used to determine a URL, causing the extension to connect to a malicious server. Depending on the use case of the extension, the web response may allow for further malicious actions (e.g., download of malicious `.bashrc`).

Assumptions: For our work, we make the following assumptions: (1) *Trusted Extensions:* VS Code extension developers do not intentionally develop malicious extensions.[2] (2) *Trusted User:* VS Code users are not intentionally attacking their own system. (3) *Disabled Restricted Mode:* The user explicitly disabled the restricted mode after cloning a potentially untrusted repository. (4) *Trusted Filesystem:* Files outside VS Code workspace (e.g., local user files) are also considered trusted.

IV. Analysis

The goal of this paper is to systematically identify code injection and file integrity vulnerabilities in VS Code extensions. Both types of vulnerabilities are classically detected using taint analysis. Taint sources are the locations in code where attackers can introduce malformed data. Taint sinks are the places in the code where the code execution or file write operations occur. A taint analysis algorithm determines if information can flow from a taint source to a taint sink.

[2]We encourage future work to investigate the VS Code marketplace for malicious extensions.

TABLE I. Taint sources and their common APIs

Source	Description	APIs / Function Calls
Workspace Settings	Data from vscode workspace settings flowing through extension application	vscode.workspace.getConfiguration
File Read	Extension reading file contents	fs.readFile() fs.readFileSync() ...
Network Response	Requesting data from the network and accepting responses	http.get(options) http.request(options) axios.get(options) request.get(options) ...
Web Server	Spawning a local web server that accepts incoming network requests	server.listen() express.get() ...

We built our VS Code analysis rules on top of CodeQL [4], an emerging static program analysis tool that transforms a program into a database and provides a specialized programming language for specifying queries of the database. Conceptually, CodeQL uses *classes* to define criteria that match a specific instruction or part of an instruction (e.g., function parameter). Classes can build upon one another to create complex queries that further refine the specific match criteria. CodeQL's taint analysis rules define the criteria to select the taint source and the taint sink. The taint source and sink queries can use a combination of classes and logic on the properties of those classes. CodeQL already has a large number of built-in classes such as `Http::RouteHandler`, `DomBasedXssQuery`, and `RemotePropertyInjectionQuery`, which are used to identify sources and sinks for classic JavaScript injection vulnerabilities in web applications. We leverage these built-in classes whenever possible. However, as motivated in Section I, existing CodeQL JavaScript vulnerability rules do not capture vulnerabilities in VS Code extensions.

The remainder of this section describes the taint sources and sinks used by our VS Code extension vulnerability rules. For each source and sink, we describe the rationale for including it. We then describe how we constructed the corresponding CodeQL query, providing short snippets in listings when appropriate. A summary of our taint sources and sinks is provided in Tables I and II.

A. Taint Sources

Taint sources are an entry point where an attacker can introduce malformed data into a system. In addition to using VS Code APIs to retrieve information, extensions can use Node.js modules and npm packages. We consider four taint sources: (1) workspace settings, (2) file read, (3) network response, and (4) web server input. Table I overviews the four taint sources and lists common APIs used by developers.

1) Workspace Settings: VS Code allows users to customize the editor, user interface, and other behavior in the editor via JSON files. While these settings are stored in files, VS Code extensions use a special API to access their configuration. Therefore, we treat the settings as a dedicated taint source.

```
class VSCodeWorkspaceConfig extends DataFlow::Node{
  VSCodeWorkspaceConfig(){
    exists(
      DataFlow::ModuleImportNode mod, DataFlow::Node
     node, AstNode ast, Token ast_parent_token |
      mod.getPath() = "vscode" and
      mod.flowsTo(node) and
      ast = node.getAstNode().getParent() and
      ast_parent_token = ast.getParent().getAToken()
      and ast_parent_token.toString().matches("%
    getConfiguration%") |
      this = node)
}}
```

Listing 2. Simplified CodeQL class for identifying workspace setting values

There are two scopes for the settings: user settings and workspace settings.

A workspace is the collection of folders opened in a VS Code window. A workspace can contain one or multiple folders. User settings apply to all VS Code windows, regardless of the folder or workspace the user opens. Workspace settings are specific to the workspace and override user settings. A user can modify the settings through the settings editor in VS Code or directly edit the `settings.json` file, which stores settings as key-value pairs. Depending on the OS platform, the JSON file for user settings is typically stored in the user's home directory, which we assume cannot be controlled by the adversary. In contrast, the JSON file for workspace settings is stored in the `.vscode` directory in the root workspace folder, which may be controlled by an adversary (see Section II). Therefore, our analysis only considers workspace settings as a taint source.

VS Code extensions can declare extension-specific settings via its `package.json` manifest file. The fields listed under `contributes.configuration` are visible and can be modified by the user. These settings provide users better customization of VS Code for installed extensions. An extension retrieves the values via the `workspace.getConfiguration` VS Code API. If the user downloads an untrusted workspace (e.g., clones a Git repository), an adversary can use the workspace settings JSON file to introduce malformed data.

As the CodeQL JavaScript library does not include predefined classes for VS Code APIs, we defined our own class called `VSCodeWorkspaceConfig`. First, we identify the `vscode` module that is imported into the application, then we track flows from the module. We use CodeQL's AST module to get syntactic structure of the flows, checking for calls to the `getConfiguration` API. Simplified code for this class is shown in listing 2.

2) Files: VS Code extensions commonly read files for reading JSON configuration, file templates, and user interface configuration. Extensions use Node.js's built-in `node:fs` module to interact with the file system. The module includes the `readFile` and `readFileSync` APIs. VS Code also provides file read APIs such as `vscode.workspace.fs.readFile`, which is similar to Node.js's `readFile`.

Similar to the attack via workspace settings, an adversary can distribute malformed data in other files in the repository. Anecdotally, we observed VS Code extensions that obtain file paths from the workspace settings and then read those files for additional configuration. Since it is nontrivial to statically track data flow through multiple files, we simply use the file read APIs as taint sources.

Automated Filtering: While the `node:fs` module allows extensions to access all files, we consider files within the extension directory as trusted. As per our threat model, extension developers are not malicious. We used the `FileSystemReadAccess` CodeQL class to identify data nodes that read data from files. To filter out trusted files, we combine the `FileSystemReadAccess` CodeQL class with another query that sets the file path of the file read as a sink and identifies sources flowing to the sink.

Filtering Limitations: Filtering the results from the combined queries sometimes required manual inspection of the taint sources. For example, the file path could be coded as `path.join(user_dir(),'settings.json')`, which combines the path returned from `user_dir()` with the string `'settings.json'`. The value from `user_dir()` also needs to be resolved. Unfortunately, CodeQL does not have good support for constant propagation to automatically resolve complex file path strings. Therefore, it is nontrivial to identify a comprehensive set of patterns to filter the file path.

The CodeQL VS Code extension provides a simple way to manually inspect all nodes on the data flow path from the taint source to the taint sink. Each node (e.g., a variable or function identified by CodeQL) is hyperlinked to its location in the source code. By scanning through the lines of code around the node and combining the information from multiple nodes, it is typically easy to determine if the file path for the file read is within the workspace. That is, nodes for file reads within the workspace commonly have some hard-coded string or a variable that would represent the workspace path. Therefore, on average, we were able to determine the file path by looking at context in source code files of 3 to 4 nodes that are on the data flow path. Inspecting each node and the surrounding source code typically took less than a minute. This manual inspection allows us to quickly acquire context for each node and determine whether the file is in a trusted directory. Section VI details the extent to which manual filtering was required for our dataset.

3) Network responses: VS Code extensions commonly make network requests to hosts on the Internet. If the extension uses an insecure protocol (e.g., HTTP), an on-path adversary can modify the server's response to introduce malformed data. Even if the extension uses a secure protocol (e.g., HTTPS), the adversary may be able to influence the network destination, e.g., via an untrusted workspace setting. Our analysis rules consider both cases.

Node.js provides multiple built-in modules for making network requests and receiving responses, e.g., `node:http` and `node:net`. VS Code extensions can also use web focused packages from npm, e.g., `axios` and `request-promise`. Each API specifies URLs in different ways. For example, the built-in `http.request` API is passed an object specifying hostname, method, and additional options. The API also has an optional *callback* parameter, which listens for the response from the network request and performs further actions.

Automated Filtering: We restrict our definition of network

```
class UntrustedURL extends TaintTracking::
    Configuration{
  override predicate isSource(DataFlow::Node source)
    {
    exists(FileSystemReadAccess src | source = src.
    getADataNode().getALocalSource()) or
    exists(ClientRequest r | source = r.
    getAResponseDataNode()) or
    exists(Http::RouteHandler rh | source = rh.
    getARequestNode()) or
    exists(VSCodeWorkspaceConfig config, DataFlow::
    SourceNode src |
      src.getFile() = config.getFile() and src.
    getStartLine() = config.getStartLine() |
      source = src)
}}
class HttpURL extends TaintTracking::Configuration{
  override predicate isSource(DataFlow::Node source)
    {
    exists(StringLiteral str| str.getStringValue().
    matches("%http://%") |source = str.flow()) or
    exists( StringOps::Concatenation str | str.
    getAnOperand().getStringValue().matches("%http
    ://%") | source = str)
}}
```

Listing 3. Simplified CodeQL filter for the network response taint source

response taint sources to URLs from untrusted sources and all HTTP URLs. CodeQL's built-in `Http::ResponseNode` class can be used to identify network responses; however, additional filtering is required to determine whether the response is untrusted. We define a new class called `UntrustedURL`, which determines if the URL comes from (a) a workspace setting, (b) a file read, (c) a network response, or (d) an input from a web server interface. For HTTP URLs, we define a new class called `HttpURL`, which determines if the string begins with "http://". Simplified code for these classes is shown in Listing 3.

4) Local web servers: Recall from our motivating example in Section II that VS Code extensions sometimes spawn a web server and listen for incoming connections. This functionality is provided by Node.js's built-in `node:http` module, as well as third-party modules such as `express`. If the adversary is able to execute JavaScript in a web browser running on the same host as VS Code, they can send malformed data to any listening VS Code extensions, as demonstrated by the motivating example.

We identify local web server taint sources using CodeQL's built-in `Http::RouteHandler` class. This class identifies the callback functions that handle requests sent to a web server API endpoint. For example, with the `express` framework, "app" is often used to handle HTTP methods with `app.get()`. The `app.get('route',callback)` matches for the specified route from the incoming request and performs further actions in the callback function. The `Http::RouteHandler` class identifies these functions that handle web server endpoints.

B. Sinks

Taint analysis determines if malformed data from an untrusted taint source can flow to a security-sensitive taint sink. We consider three taint sinks: (1) execution of shell commands, (2) evaluation of JavaScript code (i.e., `eval()`), and (3) writes

TABLE II. TAINT SINKS AND THEIR COMMON APIS

Sink	Description	APIs / Function Calls
Shell Command	Built-in Node.js modules and third-party npm packages allow for executing shell commands	child_process.exec() child_process.spawn() shell.exec() ...
Evaluate String as Code	Strings passed in will be evaluated as JavasScript statements or expressions	eval()
File Write	Writing content to files	fs.writeFile(filepath, content) fs.appendFile(filepath, content) ...

```
from
Configuration cfg, DataFlow::PathNode source,
    DataFlow::PathNode sink, StringOps::
    Concatenation concat_string
where
cfg.hasFlowPath(source, sink) and concat_string =
    sink.getNode() and
( concat_string.getFirstLeaf().toString() = "\"(\""
    or concat_string.getFirstLeaf().toString() =
    "'('" )
and
( concat_string.getLastLeaf().toString() = "\")\""
    or concat_string.getLastLeaf().toString() = "')
    '" )
```

Listing 4. Simplified CodeQL filter for eval() taint sink

to files. Table II overviews the four taint sinks and lists common APIs used by developers.

1) Shell Commands: Node.js's built-in `child_process` module allows an application to spawn subprocesses. The module provides a collection of different functions. We discuss the two most commonly used. `child_process.exec()` takes in a command string and an optional object with options such as the directory of the process or the shell to execute the command with. It spawns a shell and passes the command string directly to the shell to execute. `child_process.spawn()` takes in a command string, a list of string arguments, and an optional options object. It runs a new process with the command string, and passes in the list of arguments.

To perform a shell injection, the adversary must control strings passed as the command argument. While VS Code extensions often hard code parts of the command string, they frequently use variables for other parts. Therefore, the adversary can use symbols such as `&`, `;`, `|` to append additional commands. As CodeQL's existing `SystemCommandExecution` class for JavaScript sufficiently captures shell commands as a taint sink, we used it directly.

2) Evaluate string as code: The `eval()` function evaluates strings passed in as JavaScript code and returns the completed value of the JavaScript execution. Data flows from untrusted sources to this sink can embed arbitrary JavaScript to achieve code injection. However, if brackets are added to the beginning and end of the string passed to `eval()`, the string is evaluated as an expression rather than as statements. This use of brackets is often used to evaluate JSON objects. Note that while the adversary can abuse `eval()` to perform shell injection and write to files, we do not attempt to differentiate these attacks with our tool.

```
class FlowToWritePath extends TaintTracking::
    Configuration {
  override predicate isSource(DataFlow::Node source)
     {
    exists(VSCodeWorkspaceConfig config, DataFlow::
    SourceNode src | src.getFile() = config.getFile
    () and src.getStartLine() = config.getStartLine
    () | source = src) }
  override predicate isSink(DataFlow::Node sink) {
    exists(FileSystemWriteAccess write | sink =
    write.getAPathArgument()) }}
class FlowToWriteContent extends TaintTracking::
    Configuration {
  override predicate isSource(DataFlow::Node source)
     {
    exists(VSCodeWorkspaceConfig config, DataFlow::
    SourceNode src | src.getFile() = config.getFile
    () and src.getStartLine() = config.getStartLine
    () |  source = src) }
  override predicate isSink(DataFlow::Node sink) {
    exists(FileSystemWriteAccess write | sink =
    write.getADataNode()) }}
where
writepath.hasFlowPath(source_path, sink_path) and
writecontent.hasFlowPath(source_content,
    sink_content) and
filewrite_func.getADataNode() = sink_content.getNode
    () and
filewrite_func.getAPathArgument() = sink_path.
    getNode()
```

Listing 5. Simplified example of combining two data flows to filter file write flows. This example shows workspace settings as the source.

Automated Filtering: The CodeQL library includes the `DirectEval` class to help with identifying calls of `eval()`. However, adding brackets to the beginning and end of the string is a common way of sanitizing strings passed to `eval()`. Therefore, we added filters that identified the brackets and excluded those from the results, as shown in Listing 4.

3) File Write: The Node.js `node:fs` module provides APIs such as `writeFile` and `appendFile` to make changes to files. Two arguments are passed to these APIs: (1) a file path and (2) the content to write to the file path. We assume that for this sink to be exploitable, both file path *and* the content need to be controlled by the adversary. Subtle vulnerabilities might occur where the adversary has control of one but not the other; however, identifying such cases as vulnerable requires understanding the semantics of the VS Code extension logic. Therefore, we limit our identification of file write taint sinks as instances where there exist flows from untrusted sources to both arguments.

Automated Filtering: We used CodeQL's existing `FileSystemWriteAccess` class to identify file writes. To capture our specific threat model criteria, we combined queries that identify sources that flow to the file path and the content argument of file writes. A simplified example of the combined filtering is shown in Listing 5.

Filtering Limitations: Similar to the limitations filtering file paths for the file read taint source, we required manual inspection to filter results for the file write sink. Another challenge occurs when a VS Code extension writes to a file that it is also reading. Without context, it is unclear if it is reading from and writing to the same file, as there are multiple ways to encode the file path into a string.

Our process of manual inspection for file writes follows a similar process as manual inspection for file reads. Our additional queries identify all possible untrusted sources with a data flow to the file path and file content argument for a file write. We navigated through the nodes on the path with hyperlinks from the CodeQL VS Code extension. Combining context from the source code files of the nodes provides sufficient information to determine whether the file path or file content can be injected through untrusted sources. This process typically required inspection of 3 to 4 nodes. Inspection of each node and its surrounding source code typically took less than a minute.

C. Implementation

Our analysis pipeline consists of four stages: a package downloader, a database builder, multiple query runners, and query result parsers. We implemented it using the *BullMQ* framework [40], a Node.js library built on top of redis queue [30] that makes building microservice architectures easier. The scraper gathers extension metadata and download links for each extension package. The links were then added into the package downloader queue, where a worker downloads the extension source code from the VS Code marketplace and unpackages it. Once the code is successfully downloaded and unpackaged, the worker adds the job to a queue in our second stage of the pipeline, the database builder, where another worker runs CodeQL commands to build databases for the downloaded source code.

After CodeQL databases are successfully built, the query runner executes. Using the four sources and three sinks, we constructed twelve queries with corresponding filters. We used separate queues and workers so the queries ran in parallel. Once a query has completed, it is added to a queue for the query result parser. The worker for the parser processes the query results and transforms it into the format saved in mongoDB. Finally, we applied filters on our query results and verified the results for vulnerabilities.

V. VS Code Extension Dataset

In this section, we discuss the data we collected and characteristics within the dataset. The dataset comprises VS Code extension metadata and source code. As mentioned in Section II, VS Code extensions are largely similar to Node.js applications, thus the code is similar. As the extensions are similar to Node.js applications, we were able to collect the actual source code for the extensions.

A. Dataset Collection

We collected extension metadata and source code from the Microsoft Visual Studio Marketplace [39] in January 2023. Our dataset consists of the 43,436 extensions on the marketplace at the time of our study. We implemented a scraper to scrape the marketplace for extension metadata. Fields such as *extensionName, publisherName, lastUpdated, installs, GitHub repository* were included in the metadata. To gain further insight into the data and its dependencies, we also collected content and metadata of the extensions' GitHub repository and npm advisory data from *Open Source Insights* [27].

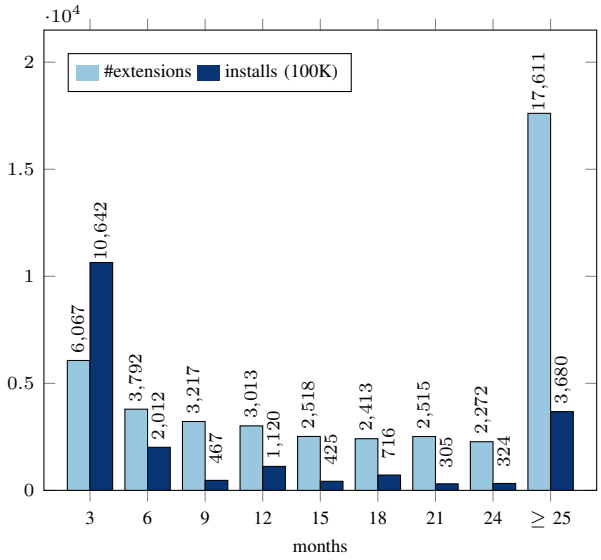

Fig. 2. Time since last update (relative to time of data collection, Jan 2023)

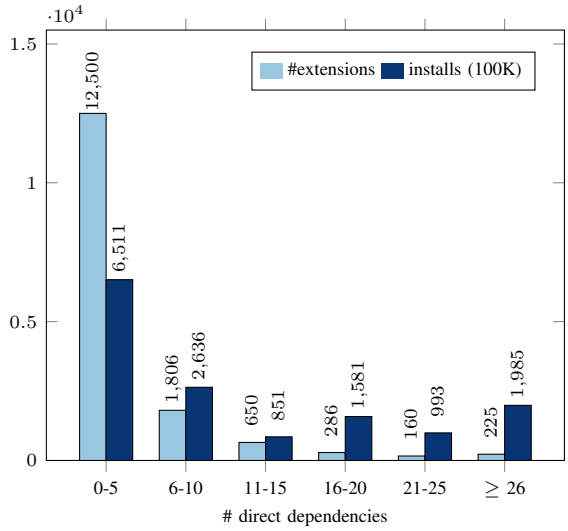

Fig. 3. Direct dependency count in a VS Code extension

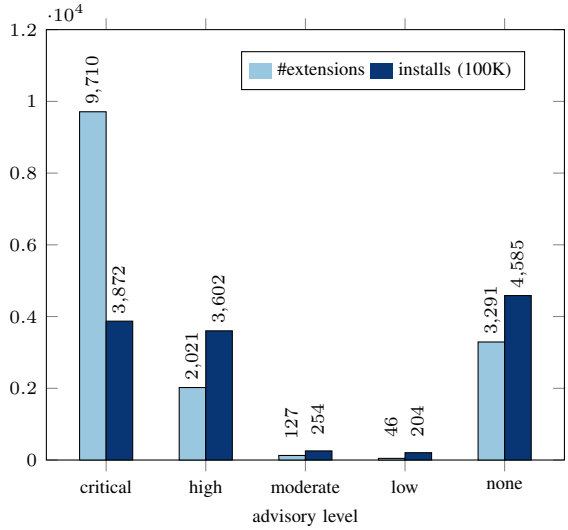

Fig. 4. Highest advisory level in VS Code extension dependencies

Our scraper also collected extension source code from the marketplace. The source code of an extension typically includes the manifest file `package.json`, the `node_modules` directory with dependency data, and the JavaScript or TypeScript code in a `src` directory. The manifest file, similarly to Node.js applications, contains general application information and dependency data. It also includes additional fields of data specific to VS Code extensions, such as *activationEvents*, *commands*, or *configuration*.

Out of the 43,436 extensions we gathered, 25,402 extensions had JavaScript or TypeScript code, the remaining 18,034 did not include code. As mentioned in Section II, VS Code extensions without code include JSON files that specify behavior of the extensions, common in extensions for themes and snippets.

B. Dataset Characterization

To develop a deeper understanding of the data, we compared various metadata metrics.

Recent Updates: We compared the field *lastUpdated* with *installations* to gain an understanding of the state of extension developers maintaining the application. As shown in Figure 2, 40% of the extensions have not been updated in more than two years. As Zahan et al. [43] proposed, unmaintained packages are considered a weak link in software. If we look at the number of installations, we observe that over 54% of the installations are extensions that have been updated in three months. 18% of the installations however, are of extensions that have not been updated in two years.

Number of Direct Dependencies: The number of dependencies in an application correlates to the attack surface of an application. The more dependencies a VS Code extension depends on, the more it relies on third-party code, which increases the chances of vulnerabilities being introduced. We compare the number of direct dependencies with installs in Figure 3, and observe that 80% of the extensions have less than five direct dependencies. From the figure, we observe that extensions with more installs tend to include more dependencies, 37% of the installations are extensions with more than ten direct dependencies. Note that the numbers in the graph only include *direct* dependencies, dependencies could be importing other dependencies, thus the number would be larger for *transitive* dependencies. Transitive dependency data required a *lock* file, such as `package-lock.json` or `yarn.lock`, that specifies the version of the package, which not all VS Code extension source code included. We were able to collect transitive dependency data for 15,185 of the extensions. Of those, 13,655 had more than 100 transitive dependencies.

Advisories in Dependencies: To gain a more comprehensive understanding of dependencies and its attack surface in VS Code extensions, we compared npm package advisories against extension dependencies. Figure 4 shows the most severe advisory level in an extension's transitive dependencies. Of the 15,185 extensions where we could collect exact transitive

dependency version data, 9710 (63.9%) imported a package with a *critical-level* advisory. The large number of VS Code extensions that import a *critical-level* advisory shows the impact a supply chain attack through the npm ecosystem has on VS Code extensions.

C. Access to system resources through Node.js

We present an overview of the number of VS Code extensions with access to files, network, shell commands, and local web servers. While access to these resources through the Node.js environment do not directly pose a threat, it provides context to better understand results of data flows from our taint analysis. Figure 5 shows the number of VS Code extensions where we identified the extension accessing files, network resources, shell commands, and spawning local web servers. CodeQL queries were constructed to identify these instances. For a web server to be vulnerable, some action has to be done on the incoming network request. It might be the case that a VS Code extension spawns a web server and opens a socket, but there is no clear API endpoint that an adversary can target. Therefore, we not only identified applications that spawn a web server, but further identified an existing function that handles the incoming request for us to classify it as spawning a web server.

From the 322 extensions where we identified a web server, we installed and activated the extensions to check whether sockets on the local system were opened by the VS Code extension. We were able to verify 65 VS Code extensions that opened sockets *immediately* after extension activation. Of those where we did not detect an open socket, we were not able to trigger the specific flow of events that caused an extension to open sockets therefore we could not verify. We also identified 5,112 extensions that access files, 4,074 extensions that access the network, and 1,902 extensions that call shell command APIs. Those resources were accessed through Node.js modules, which shows VS Code extensions' reliance on the Node.js ecosystem.

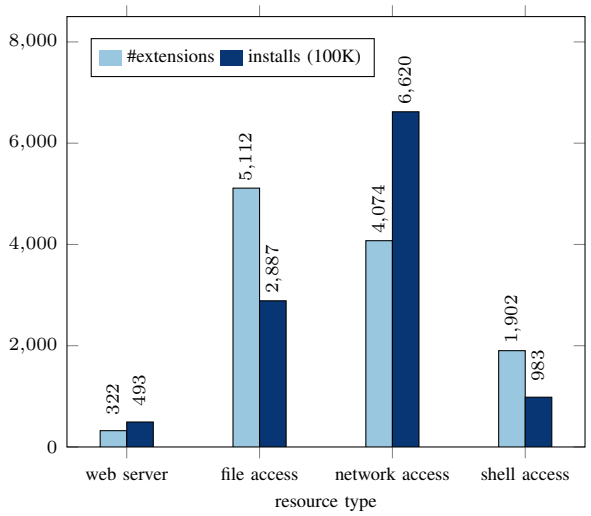

Fig. 5. Overview of resources accessed by VS Code extensions identified by CodeQL

```
// taint source
get gitPaths() {
  const configValue = vscode.workspace.
    getConfiguration('git').get('path', null);
  ...
}
// taint sink
function getGitExecutable(path) {
  return new Promise((resolve, reject) => {
    resolveSpawnOutput(cp.spawn(path, ['--version'])
    ).then((values) => {
    ...
}
```

Listing 6. Simplified code of taint source and taint sink in git-graph VS Code extension

VI. Vulnerability Results

In this section, we discuss the vulnerabilities identified by our analysis. The section is organized by taint source, i.e., the attack vector used to exploit the vulnerability. For each taint source, we provide a table that details the reduction from the number of possible extensions that could have contained the vulnerability (i.e., the code includes both the taint source and taint sink) down to data flows between the source and sink. The tables also detail the reduction provided by both automated and manual filtering. When feasible, we created proof-of-concept (PoC) exploits to demonstrate impact. As developing PoCs is time and labor intensive, we focused on extensions with higher installations and when code was not obfuscated.

A. VS Code workspace settings

Workspace setting values in VS Code allow for malformed data to enter VS Code extensions. An adversary could distribute repositories including malformed data, targeting specific VS Code extension vulnerabilities. It is not uncommon for open source repositories to include VS Code settings in the `.vscode` subdirectory. Suppose a VS Code user has the targeted VS Code extension installed, downloads the repository with malformed data, and opens the repository in VS Code. The malformed payload could enter the VS Code extension application through workspace setting values and lead to a remote code injection attack. Table III shows the number of flows we detected flowing from our first source: workspace settings to our three sinks, (1) shell command, (2) `eval()`, (3) file write, which we will discuss separately in this section.

1) Sink 1 - Shell Command: We identified 2,213 extensions where both a workspace setting and a shell command existed. Our CodeQL query tracking data flows from workspace settings to shell command identified 389 extensions, which is around 18% of those with source-sink combinations. Sorting by the number of installations, we manually inspected the top 27 with more than 40,000 installations. Out of the 27, we verified 7 extensions where the identified vulnerable data flows allowed for code execution, the total number of installations amounting to more than 5.7 million. Of the remaining 20 extensions, 5 were clearly not exploitable, 4 had obfuscated code, another 4 extensions were deprecated or no longer on the marketplace, and 7 we were unable to verify for proof of concepts for an exploit.

Verified Exploit: For example, we identified a flow from a workspace configuration value to a shell command in the *git-*

TABLE III. NUMBER OF IDENTIFIED FLOWS FROM THE WORKSPACE SETTINGS TAINT SOURCE

Sink	Number of extensions with calls to both the source and the sink	Flows	Filters		Filtered Flows	Investigated	PoC	Not Exploitable		Unable to Confirm			
			eval() without enclosing brackets	untrusted source to filepath and content				hard-coded write filepath	other	deprecated	obfuscated	activation error	other
shell	2,213	389	0	0	389	27	7	0	5	4	4	0	7
eval	192	12	12	0	12	12	6	0	4	0	1	0	1
file write	1,847	79	0	24	24	24	0	13	0	1	1	6	3

```
"git": { "path": "a-shell-script.sh" }
```

Listing 7. Workspace setting value to exploit git-graph extension vulnerability

```
function showAvailableCommands(context, manageFile)
    {
  vscode.window.showInformationMessage(
    LOAD_COMMANDS_MSG);
  const pythonPath = vscode.workspace.
    getConfiguration(PYTHON).get(
    CONFIGURATION_PYTHON_PATH);
  const command = `${pythonPath} ${manageFile} help
    --commands`;
  cp.exec(command, (err, stdout) => {
  ...
```

Listing 8. Simplified code of taint source and taint sink of in django-commands VS Code extension

graph[3] VS Code extension (5.1 million installations). This extension shows a git graph for a repository and allows the user to perform actions on the graph. As shown in Listing 6, the extension gets the value for the field `git.path` from the workspace setting (line 3), the value goes through a couple function calls and is finally passed into `child_process.spawn` as the `path` argument to be executed as a shell command (line 8). To exploit this, we set up a workspace in VS Code and modified the `git.path` value in the workspace configuration file to the path of a shell script, as in Listing 7. With the extension installed, we were able to execute the shell script upon opening the workspace in VS Code.

Another extension we identified is *Django Commands*[4] (49,000 downloads). The extension allows developers to run commands in django projects in VS Code. As shown in Listing 8, we identified a flow from a workspace configuration value (line 3) to a shell command (line 5). An adversary could create an open source django repository including the `.vscode` directory that contains malformed workspace setting values to target extension users. Suppose a user installs the *Django Commands* extension and downloads an open source repository of a django project template containing VS Code workspace setting configuration to quickstart a project. Opening the repository with malformed payload in VS Code could lead to a remote code injection attack. We verified this by crafting the workspace setting value as shown in Listing 9 and were able to execute the malformed payload as a shell command.

[3] `https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph`

[4] `https://marketplace.visualstudio.com/items?itemName=MaxChamps.django-commands`

```
{ "python.pythonPath": "a-shell-script.sh;" }
```

Listing 9. Workspace setting value to exploit scss-lint extension vulnerability

```
let scssLintPath = hasConfigFileDir && foundFile ?
    path.join(configFileDir, '.scss-lint.yml') : '';
const configCmd = scssLintPath ? `-c "${scssLintPath
    }" ` : '';
// taint sink
const cmd = `${echoCmd}scss-lint ${pathCmd}${
    configCmd}${fileCmd}`;
exec(cmd, CONFIG_OBJ, (err, stdout) => { ...
```

Listing 10. Simplified code of taint source and taint sink of unexploitable flow

Non-exploitable flow: We present an example to discuss how manual inspection was required to determine that the flows identified by CodeQL were not exploitable. The simplified code is shown in Listing 10. Here, a flow from the workspace setting value to a shell command was detected by our tool. The extension retrieves a workspace setting value which is passed to the `configFileDir` variable (line 1), then passed to conditional operators to determine the values of strings passed into the `scssLintPath` variable in the shell command. Upon manual inspection, we find that the `scssLintPath` variable value is resolved from either another variable or from the combination of our `configFileDir` variable with a hard-coded string. While there exists a data flow from our taint source workspace settings to our taint sink shell command, we determined that it is not able to achieve a code injection attack with only a malformed value from the workspace setting.

2) Sink 2 - Eval(): There were 192 extensions where both a workspace setting and a `eval()` existed. Our CodeQL tool identified 12 VS Code extensions with flows from the source to the sink, all of which did not include brackets enclosing the string passed to the function. We manually inspected them and found one with obfuscated code. For the remaining 11 extensions, we verified vulnerabilities for 6 extensions with proof of concepts for code injection, where the total number of installations add up to more than 80,000. 4 extensions were not exploitable and one we were not able to confirm.

Verified Exploit: Our case study extension for this vulnerability is the *scss-lint*[5] extension, a linter for scss in VS Code with more than 62,000 installations. As shown in Listing 11, the extension reads for the workspace setting value `scssLint.statusBarText` (line 3), which is passed to a `eval()`

[5] `https://marketplace.visualstudio.com/items?itemName=adamwalzer.scss-lint`

```
// taint source
const updateConfig = () => {
  const newConfig = vscode.workspace.
    getConfiguration('scssLint');
  statusBarText = newConfig.statusBarText;
  ...
}
// taint sink
this._statusBarItem.text = eval(statusBarText);
```

Listing 11. Simplified code of taint source and taint sink in scss-lint VS Code extension

```
"scssLint.statusBarText": "exec('ls > outputFile.out
    ');"
```

Listing 12. Workspace setting content to exploit scss-lint extension vulnerability

function (line 8). We exploited this by crafting a workspace setting value with a JavaScript string for the specified field as shown in Listing 12. The JavaScript string was passed to the `eval()`, evaluated as a statement, and we were able to achieve code injection.

3) Sink 3 - File Write: For our third sink, we were not able to verify proof of concepts for code injection attacks. There were 1,847 extensions where both a workspace setting and a file write existed. Our CodeQL query identified 79 VS Code extensions with data flows from workspace settings to file write. Of those, 24 extensions had a flow from an untrusted source to both the write file path and the write content. After manual investigation, we found one extension with obfuscated code, one that was deprecated, 13 where the write file path had hard-coded filenames or file extensions, 6 where we could not correctly activate the extension to test exploits, and 3 where we could not confirm proof of concepts.

Non-exploitable flow: To demonstrate how static analysis could lack context in determining exploitability, we discuss one of the flows we identified but categorized as not exploitable. The simplified code for this example is shown in Listing 13. Our tool identified a data flow from a workspace setting (line 2) to a file write (line 8). We inspected the source code and tested the extension and observed that the extension uses the workspace setting value as the heading value in the outputted markdown file, thus we determined this was not exploitable for code injection.

B. Files

As VS Code extensions are able to read file data by leveraging Node.js modules, files opened in a VS Code workspace could be a source of malformed data. Similar to the scenario discussed in the previous section for the workspace setting taint source (Section VI-A), an adversary could distribute repositories with malformed data. Opening the repositories in VS Code would allow the malformed data to enter extension applications. What sets files apart from workspace settings as a taint source is that files could be of any filename, while workspace settings are stored in a specific file. Here we discuss the number of data flows we detected from our second source: file read to our three sinks (1) shell command, (2) `eval()`, (3) file write, as shown in Table IV.

```
function dateHeader() {
  const configDateFormat = vscode.workspace.
    getConfiguration().get("dailyNotes.dateFormat");
  if (configDateFormat) {
    return "## " + moment(today).format(
    configDateFormat) + "\r\n\r\n\r\n";
  }
... }
async function prependFile(filePath, content) {
    await fsp.writeFile(filePath, content);
... }
```

Listing 13. Simplified code of non-exploitable flow from workspace settings to file write

1) Sink 1 - Shell Command: There were 1,718 extensions where both the file read and shell command existed. Our CodeQL query tracking data flows from a file read to shell command identified 75 extensions. We manually inspected 26 extensions with more than 5,000 installations. Out of the 26, we were able to come up with proof of concepts for code injection for 4 of the extensions, adding up to more than 121,000 installations. Of the remaining, 9 were reading from files in the extension's directory (considered trusted), 2 were not exploitable, 2 had obfuscated code, 4 we were not able to activate correctly, and 5 we were not able to verify.

Verified Exploit: One extension where we found a vulnerability is *CMake Test Explorer*[6], with more than 76,000 installations. As shown in Listing 14, the source of the flow is in `getCtestPath()`, where the extension is reading from a `cacheFilePath` file in the workspace (line 4). The value of the filepath is resolved by joining the `cwd` and `CMAKE_CACHE_FILE` variable. As discussed in Section IV-A2, using our combined CodeQL queries, we determined that the resolved filepath is within the workspace and therefore untrusted by manual inspection. The content of the file flows to our sink in `loadCmakeTests()`, where it is passed to a `spawn()` (line 14) that executes the content as a shell command. We exploited this flow by creating a `CMakeCache.txt` file in the workspace, and setting the content of the file to a shell script path. Then by setting certain workspace setting values to trigger the vulnerability flow, we were able to execute the shell script.

2) Sink 2 - Eval(): There were 397 extensions with the file read and `eval()` combination, our CodeQL tool identified 55 extensions where there existed a data flow between this source and sink. Our filters detected 21 extensions which had brackets enclosing the string passed to `eval()`, thus they were excluded. We inspected the remaining 34 extensions, and verified vulnerabilities for 4 extensions with proof of concepts for code injection, with the sum of installations totaling to more than 11,000. Of the remaining 30 extensions, we found that 3 were obfuscated, 18 were reading from files within the extension or required the user to choose the file, and 9 we were not able to confirm with exploits.

Verified Exploit: For this flow, we present the extension *autodocblocker*[7], with more than 10,000 installations. As

[6] `https://marketplace.visualstudio.com/items?itemName=fredericbonnet.cmake-test-adapter`

[7] `https://marketplace.visualstudio.com/items?itemName=maddog986.autodocblocker`

TABLE IV. NUMBER OF IDENTIFIED FLOWS FROM THE FILE READ TAINT SOURCE

Sink	Number of extensions with calls to both the source and the sink	Flows	Filters		Filtered Flows	Investigated	PoC	Not Exploitable			Unable to Confirm			
			eval() without enclosing brackets	untrusted source to filepath and content				hard-coded write filepath	read from file in extension	other	deprecated	obfuscated	activation error	other
shell	1,718	75	0	0	75	26	4	0	9	2	0	2	4	5
eval	397	55	34	0	34	34	4	0	18	0	0	3	0	9
file write	2,847	1,296	0	150	150	21	0	6	4	6	2	1	1	1

```
// taint source
function getCtestPath(cwd) {
  const cacheFilePath = path.join(cwd,
    CMAKE_CACHE_FILE);
  const match = fs.readFileSync(cacheFilePath).
    toString().match(CTEST_RE);
...
  return match[1];
}
// taint sink
function loadCmakeTests(ctestPath, cwd,
  buildConfig, extraArgs = '') {
  return new Promise((resolve, reject) => {
    try {
    ...
      const ctestProcess = child_process.spawn(
    ctestPath, [...], { cwd });
}}}
```

Listing 14. Simplified code of taint source and taint sink in CMake Test Explorer VS Code extension

```
for (var folder of vscode.workspace.workspaceFolders
    ) {
  if (fs.existsSync(folder.uri.fsPath + "\\.
    autodocblocker.js")) {
    try {
      eval(fs.readFileSync(folder.uri.fsPath + '\\.
    autodocblocker.js', 'utf8'));
    }
  ...
```

Listing 15. Simplified code of taint source and taint sink in autodocblocker VS Code extension

shown in Listing 15, the extension reads content from a `.autodocblocker.js` file in the workspace (line 4), then passes the file content to `eval()` to evaluate as JavaScript statements. To exploit this, we created a `.autodocblocker.js` file in the workspace with payload as shown in Listing 16. Our payload includes the `exec()` function from the Node.js built-in module `child_process`, as it is the most direct way to achieve code injection. Other JavaScript code could also be embedded for other attacks.

3) Sink 3 - File Write: 2,847 extensions had a file read and a file write combination. Our CodeQL query tracking data flows from a file read to file write identified 1,296 extensions. We applied filters to identify those where there existed a flow from an untrusted source to both the file path and the content argument. This reduced the number to 150 extensions. Of those, we inspected the 21 extensions with more than 5,000 installations and found one that was obfuscated, 2 were deprecated, one was obfuscated, 6 had the filename hard-coded, 4 read from files within the extension, one where we were not able to activate the extension, and one where we could not confirm the vulnerability with proof of concept for code injection.

```
const cp = require('node:child_process');
cp.exec('dir > "/path"', (err, stdout, stderr) => {
  if(err){ console.log(err); }
});
```

Listing 16. File content to exploit autodocblocker extension vulnerability

C. Network Response

As shown in Section V-C, VS Code extensions can make outgoing network requests and receive incoming network responses. This allows for untrusted data to enter VS Code extension applications through network responses. The number of combinations we found for this source and the various sinks was drastically less than compared to the two previous sources. Table V shows the number of flows we detected flowing from our network responses to our three taint sinks.

1) Sink 1 - Shell Command: Although we identified a good amount of vulnerable flows from the two previous sources to shell command, our CodeQL tool did not identify any extension with data flows from a network response to a shell command. Only 174 extensions were found to have both a network response and a shell command exist in the source code.

2) Sink 2 - Eval(): We identified 122 extensions with this source-sink combination. For data flows from a network response to an `eval()` as a sink, our CodeQL tool identified 19 extensions. 15 of those had brackets enclosing the string passed to `eval()`. We excluded those and further filtered the results to restrict the URLs to insecure URLs and narrowed it down to one extension. Due to the complicated logic in the source code, we were not able to verify the exploitability of the vulnerability with a proof of concept for code injection.

3) Sink 3 - File Write: 259 extensions were found to have a call to a network response and a file write. We identified 191 extensions with a flow from network response to a file write. After applying filters, we were left with 25 extensions where (1) the URL of the network response is insecure and (2) there exists a flow from untrusted sources to both the file path and content arguments in the file write function. After investigating the 25 extensions, we found that one was clearly not exploitable, 17 had the write file path hard-coded or required user input for the path, one was obfuscated, 2 extensions we could not activate, and another 4 where we

TABLE V. NUMBER OF IDENTIFIED FLOWS FROM THE NETWORK RESPONSE TAINT SOURCE

Sink	Number of extensions with calls to both the source and the sink	Flows	Filters			Filtered Flows	Investigated	PoC	Not Exploitable		Unable to Confirm		
			eval() without enclosing brackets	untrusted source to filepath and content	insecure URL				hard-coded write filepath	other	obfuscated	activation error	other
shell	174	0	0	0	0	0	0	0	0	0	0	0	0
eval	122	19	15	1	0	1	1	0	0	0	0	0	1
file write	259	191	0	42	25	25	25	0	13	5	1	2	4

```
1 let extension = ".god";
2 // action on network response
3 return contentReq.then((response) => {
4   let outFileName = getModulePath(outline.entity.
      ownerName) + "\\" + outline.entity.ownerName +
      extension;
5   outFileName = path.normalize(outFileName);
6   fs.writeFile(outFileName.replace(/\\/g, '/'),
      response["content"]);
```

Listing 17. Simplified code of non-exploitable flow from network response to file write

```
1 // taint source
2 this.app = express();
3 this._localWebService = new localWebService_1.
      LocalWebService(extensionPath);
4 ...
5 this._localWebService.addPostJobDetailHandler("/api/
      editScript", (req, res) => this.
      apiEditScriptRequestHandler(req, res));
6 // taint sink
7 let jobCacheFolder = path.join(vscode.workspace.
      rootPath, '.jobCache');
8 ...
9 fs.writeFileSync(scriptPath, jobDetail.properties.
      script, { encoding: 'utf8' });
```

Listing 18. Simplified code of taint sources and taint sink in Azure Data Lake Tools extension

could not confirm the vulnerabilities with proof of concepts for code execution.

Non-exploitable flow: We present an example where we determined the flow was not exploitable after manual inspection. The simplified code for the flow is shown in Listing 17. This extension receives a network response and writes the content to a `outFileName` (line 6). This data flow seems exploitable as it writes data from the network to a file. However, after manual inspection, we observe that the `extension` variable (line 4) specified the file extension of `outFileName` to `.god` (line 1). This prevents a malicious actor from setting the file path of the file written to a sensitive file such as `.bashrc`. Therefore, we determine this flow as not exploitable.

D. Web Server

VS Code extensions have the capability to spawn web servers through Node.js modules. Web servers open ports on the user's system and allow untrusted data to enter through API endpoints declared by the web server. Table VI shows the number of flows we detected flowing from web server API endpoints to our three taint sinks. The number of source-sink combinations for this category is also drastically less than compared to the first two sources workspace setting and file read.

1) Sink 1 - Shell Command: We identified 151 extensions where a web server and a shell command existed. Our CodeQL query tracking data flows from a web server API to a shell command identified 3 extensions. We inspected all 3 extensions, one of which was obfuscated, one we were not able to activate and another one we were not able to confirm vulnerabilities with proof of concepts for code injection.

2) Sink 2 - Eval(): Although data flows to `eval()` yielded results for taint sources we discussed previously, our CodeQL query did not identify any extension where there existed a flow from a web server API endpoint to `eval()`. Only 64 extensions had both a web server and a `eval()` exist in the source code.

3) Sink 3 - File Write: We identified 6 extensions, out of 146 with the source-sink combination, where data flows from a web server API endpoint to a file write were identified with our CodeQL tool. 3 of them had a flow from an untrusted source to both the write file path and write content. We inspected them and found one had the file path hard-coded, another one we were not able to activate correctly, and one where we verified it had a partial file integrity attack.

Verified Exploit: The extension where we identified a partial file integrity attack is the *Azure Data Lake Tools*[8] extension, with more than 279,000 installations. As shown in Listing 18, it spawns a web server on the user's system using the *express* framework and declares web server APIs (line 5), from which incoming data flows to a file write. Data incoming from the network to the web server would be written to a subdirectory within the workspace (line 9). Upon manual inspection, we were able to exploit the *editScript* API by sending payload shown in Listing 19 that writes file content passed in the request body to the file name specified in the payload. An adversary could send requests to the local web server with malformed payload through a website the user visits and perform a partial file integrity attack.

VII. DISCUSSION

While we identified four sources and three sinks from our threat model, our results show that certain combinations of taint sources and taint sinks are more likely to expose a VS Code extension user to attacks. Data from untrusted sources

[8] `https://marketplace.visualstudio.com/items?itemName=usqlextpublisher.usql-vscode-ext`

TABLE VI. NUMBER OF IDENTIFIED FLOWS FROM THE WEB SERVER TAINT SOURCE

Sink	Number of extensions with calls to both the source and the sink	Flows	Filters	Filtered Flows	Investigated	PoC	Not Exploitable		Unable to Confirm		
			untrusted source to filepath and content				hard-coded write filepath	other	obfuscated	activation error	other
shell	151	3	0	3	3	0	0	0	1	1	1
eval	64	0	0	0	0	0	0	0	0	0	0
file write	146	6	3	3	3	1*	1	0	0	1	0

* partial file integrity attack.

```
{ "detail":{
      "name": "${filename}",
      "properties":{
          "script": "${filecontent}" }}}
```

Listing 19. Request body sent to extension API for partial file integrity attack

such as workspace settings and files in the workspace were the most common in our identified vulnerability flows and verified code execution vulnerabilities. Shell command and `eval()` taint sinks consist of the large majority of our verified code execution vulnerabilities. While these sources and sinks allow a VS Code extension to provide rich features to the IDE, we argue that more security measures could be taken to prevent code execution attacks. Sanitizers could be implemented by VS Code extension developers to filter out malformed data from untrusted sources, string checking could also be put in place for strings passed to shell commands or `eval()`. While the VS Code marketplace implements a check on extension source code when it is uploaded to the marketplace, it is unclear how the checks are implemented and the effectiveness of it. VS Code extensions could also be obfuscated, adding complexity to code reviews. The Chrome extension web store, for example, does not allow obfuscated code [36].

We encountered some interesting findings while going through the taint analysis results. While these are not necessarily exploitable code injection vulnerabilities in VS Code extensions, we hope it provides insight on practices that should or should not be adopted when developing VS Code extensions.

Heroku URLs: Through our CodeQL query that identifies data flows from network responses, we identified 5 extensions that made outgoing requests to Heroku URLs and performed actions on the responses. Heroku [13], is a platform as a service that allows developers to build applications. It provides a subdomain under "herokuapp.com" where developers can host their applications, which is registered upon the creation of the app. No two applications can be registered under the same subdomain; however, subdomains of deleted applications can be used in future apps. Four of the five extensions we identified with Heroku URLs made requests to deleted Heroku applications. As deleted Heroku applications release Heroku subdomains, an adversary could take advantage of these and register new apps under the same subdomain and distribute malformed payload to extension users. For example, one of the extensions we identified, *EasyAPI*[9], makes network requests to "https://flask-apillist.herokuapp.com". Visiting the URL shows that no Heroku application exists there, allowing anyone to register a new application under "flask-apillist.herokuapp.com".

Similar Code: We observed multiple cases of highly similar code in separate VS Code extensions. We identified a flow from a network response to a file write in the extension, *vscode-element-helper*[10]. After going through similar flows from other extensions, we identified nine other VS Code extensions with similar code. We ran the files from the ten extensions through the *Moss* system [31] for code similarity detection. Moss selected pairs from the ten extensions, comparing each pair of directories and returned the number of lines matched between the compared files and the percentage of matched lines within each directory. There were a total of ten extensions, therefore 45 comparisons. Out of the 45, 28 of the comparisons returned a percentage of at least 90% of matched lines in at least one of the directories. All comparisons had at least one directory where 40% of the lines were matched. While we were not able to verify whether the detected flows are exploitable vulnerabilities, the similarity of code in different extensions sheds light on how code clones [17] could allow vulnerabilities to propagate through the VS Code marketplace ecosystem.

Limitations: CodeQL queries code databases by searching for specific patterns to identify certain expressions in code. There may be edge cases or unconventional ways of coding where our CodeQL query tool will not be able to detect our defined data flows. Furthermore, our tool focused on the data flows within the VS Code extension, excluding dependency source code, as our study is on VS Code extensions rather than npm packages. However, there could exist vulnerabilities in the dependencies that affect the users of VS Code. Additionally, static analysis could include false positives, we attempted to eliminate them through the filters we applied and verifying exploits. Verifying proof-of-concepts for the vulnerabilities required extensive manual investigation, hence we sampled the extensions identified by our tool to verify PoCs for exploits.

Responsible Disclosure: We have notified all developers of the extensions where we verified PoC exploits. Six developers responded confirming vulnerabilities, with three developers fixing the vulnerability and releasing new versions for the extension. We have also notified the GitHub Security Team and Visual Studio Code team at Microsoft of our results.

[9] https://marketplace.visualstudio.com/items?itemName=webStarter.easyapi

[10] https://marketplace.visualstudio.com/items?itemName=ElemeFE.vscode-element-helper

VIII. Related Work

NPM dependencies and Node.js: Much work has been done on dependencies in package managers and the npm ecosystem overall [3], [6], [18], [21], [25], [41]. Cox et al. [6] proposed metrics to measure the dependency freshness in software and found that software with outdated dependencies are more likely to encounter security issues. Prior work has shown that the large number of maintainers and contributors provides a large attack surface [43], [44]. Further, the number of dependents and dependencies for a package increases the effect a malicious package can cause. Studies have shown that while packages have few direct dependencies, they have a much higher number for transitive dependencies [8]. Popular packages have a large influence over the ecosystem, some having more than a hundred thousand packages as dependents. It is estimated that up to 40% of the packages are reliant on packages with known vulnerabilities [44]. Ohm et al. [24] has shown that a malicious package is available for 209 days on average before being reported. Decan et al. [7] find that it takes more than two years to find 50% of the vulnerabilities. They find that more than 50% of packages are affected by vulnerabilities in its dependencies.

With the large amount of dependencies in the Node.js ecosystem, Abdalkareem et al. [1] analyzed the use of trivial packages in npm. Their results show that 16.8% of npm packages are trivial packages, and 10.9% of node applications depend on them. The leftpad incident shows how large an effect a trivial package with eleven lines of code has on the ecosystem [5]. As more attention is directed toward vulnerabilities in npm, various systems have been proposed to mitigate the vulnerabilities from npm packages. Ferreira et al. [10] proposed a permission system to limit and contain permissions used by packages. Nielsen et al. [23] presented Nodest, while Staicu et al. [35] proposed Synode, both static analysis tools to detect vulnerabilities for injection attacks. Koishybayev and Kapravelos [19] presented a static analysis tool, Mininode, to remove unused code in Node.js applications.

Vulnerabilities in open source software: Being a free registry, npm faces the same threats open-source software faces. There have been efforts in prior work to address these threats. Hoepman and Jacobs [14] argue that open source is more secure than closed source as it allows more developers to review code and contribute to software. While others argue that is not the case, since not all developers are experienced with vulnerabilities and may not have the intention of finding them [22], [28], [32], [37]. The onion model proposed by Aberdour [2] shows the characteristics for a sustainable software development community. Jarczyk et al. [15] proposed two metrics for measuring the quality of open source projects. Their study also found that the number of repositories a developer has correlates negatively with the number of bugs fixed.

IX. Conclusion

Developers commonly install extensions for added functionality. However, vulnerabilities in IDE extensions are an attractive target for adversaries who are increasingly targeting the software supply chain. In this paper, we studied the security of extensions in the VS Code marketplace. We defined a threat model for extensions and a collection of CodeQL taint analysis rules to detect vulnerabilities specific to VS Code extensions. We then performed an ecosystem-wide vulnerability study, showing that while vulnerabilities are not pervasive, they are significant and impact millions of users. We hope that our results will raise awareness of vulnerabilities in extensions and motivate greater measures to restrict access given to extensions in VS Code.

Acknowledgment

This work is supported in part by NSF grants CNS-2207008 and CNS-2247686, and by the Office of Naval Research (ONR) under grant N00014-21-1-2159. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 385–395.

[2] M. Aberdour, "Achieving quality in open-source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, 2007.

[3] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[4] "Codeql." [Online]. Available: https://codeql.github.com/

[5] K. Collins, "How one programmer broke the internet by deleting a tiny piece of code," https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code. [Online]. Available: https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/

[6] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 109–118.

[7] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[8] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[9] "Extension Anatomy." [Online]. Available: https://code.visualstudio.com/api/get-started/extension-anatomy/#extension-manifest

[10] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1334–1346.

[11] I. Goldman and Y. Kadkoda, "Can you trust your vscode extensions?" Mar 2023. [Online]. Available: https://blog.aquasec.com/can-you-trust-your-vscode-extensions

[12] D. Goodin, "Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store," Ars Technica, Sep. 2015. [Online]. Available: https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/

[13] "Heroku." [Online]. Available: https://www.heroku.com/platform

[14] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *Communications of the ACM*, vol. 50, no. 1, pp. 79–83, 2007.

[15] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki, "Github projects. quality analysis of open-source software," in *International Conference on Social Informatics*. Springer, 2014, pp. 80–94.

[16] Z. Jin, S. Chen, Y. Chen, H. Duan, J. Chen, and J. Wu, "A security study about electron applications and a programming methodology to tame dom functionalities."

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE transactions on software engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[18] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[19] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 121–134.

[20] Krebs On Security, "3CX Breach Was a Double Supply Chain Compromise," Apr. 2023. [Online]. Available: https://krebsonsecurity.com/2023/04/3cx-breach-was-a-double-supply-chain-compromise/

[21] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[22] E. Levy, "Wide open source; is open source really more secure than closed," *Elias Levy says there's a little security in obscurity. SecurityFocus*, 2000.

[23] B. B. Nielsen, B. Hassanshahi, and F. Gauthier, "Nodest: feedback-driven static analysis of node.js applications," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 455–465.

[24] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2020, pp. 23–43.

[25] A. Ojamaa and K. Düüna, "Assessing the security of node.js platform," in *2012 International Conference for Internet Technology and Secured Transactions*. IEEE, 2012, pp. 348–355.

[26] R. Onitza-Klugman and K. Efimov, "Deep dive into visual studio code extension security vulnerabilities," Nov 2021. [Online]. Available: https://snyk.io/blog/visual-studio-code-extension-security-vulnerabilities-deep-dive/

[27] "Open source insights." [Online]. Available: https://deps.dev/

[28] C. Payne, "On the security of open source software," *Information systems journal*, vol. 12, no. 1, pp. 61–78, 2002.

[29] K. Peguero and X. Cheng, "Electrolint and security of electron applications," *High-Confidence Computing*, vol. 1, no. 2, p. 100032, 2021.

[30] "Redis." [Online]. Available: https://redis.io/

[31] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[32] G. Schryen, "Is open source security a myth?" *Communications of the ACM*, vol. 54, no. 5, pp. 130–140, 2011.

[33] "Setup Visual Studio Code's Network Connection." [Online]. Available: https://code.visualstudio.com/docs/setup/network

[34] "Stack overflow developer survey 2022." [Online]. Available: https://survey.stackoverflow.co/2022/

[35] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js." in *NDSS*, 2018.

[36] "Trustworthy chrome extensions, by default," Oct 2018. [Online]. Available: https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html

[37] J. Viega, "The myth of open source security," 2001.

[38] "Visual studio code - remote code execution in restricted mode (cve-2021-43908)," Jun 2022. [Online]. Available: https://blog.electrovolt.io/posts/vscode-rce/

[39] "Visual Studio Marketplace." [Online]. Available: https://marketplace.visualstudio.com/vscode

[40] "What is BullMQ - BullMQ." [Online]. Available: https://docs.bullmq.io/

[41] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 351–361.

[42] "Workspace Trust Extension Guide." [Online]. Available: https://code.visualstudio.com/api/extension-guides/workspace-trust

[43] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 331–340.

[44] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

APPENDIX

A. Description & Requirements

The dataset for our study consisted of more than 25K packages. We constructed CodeQL queries to run on these packages to conduct data flow analysis. It is not feasible to include all packages as they take up a lot of storage. Therefore, our goal is to publish the queries we used and show its executability. We will provide some packages and the experiments to run on them to acquire a small amount of our results.

1) How to access: https://doi.org/10.5281/zenodo.10146469

2) Hardware dependencies: None.

3) Software dependencies: None.

4) Benchmarks: We provided five sample packages to perform experiments on. The folders in *sample-data* are CodeQL databases built from the VS Code extension source code. These databases are what we will execute the queries on.

B. Artifact Installation & Configuration

Have the VS Code IDE and the VS Code CodeQL extension installed. Additional information on installing VS Code and the VS Code CodeQL extension can be found in the *Requirements* section in the *readme.md* in the GitHub repository or in VS Code and CodeQL documentation. Download the repository at the GitHub link. Opening the repository in the VS Code IDE will set up the required CodeQL environment and allow you to run the queries in your VS Code workspace.

C. Experiment Workflow

1) Prepare the databases for the experiments: Sample CodeQL databases are provided in the repository in the directory *sample-data*. If you wish to build other CodeQL databases to run experiments on, use the CodeQL CLI and run the `codeql database create` command.

2) Running the experiments (queries):

1) Open the GitHub repository (section A1) in VS Code.
2) Load and select the CodeQL database in the *sample-data* directory into the workspace using the CodeQL VS Code extension. Figure 6 shows the interface to load databases in VS Code.

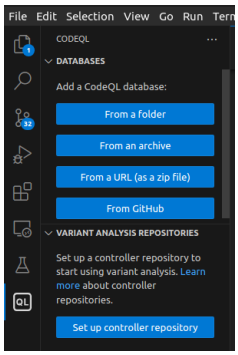

Fig. 6. Interface in VS Code CodeQL extension to load database.

Fig. 7. Selecting database in CodeQL extension.

3) Select database, as shown in figure 7, a checkmark indicates database is selected.
4) Right-click on the query you wish to execute and select `CodeQL: Run Query on Selected Database` to execute the query, as shown in figure 8. Results of the query will display in the VS Code workspace.

D. Major Claims

Our study shows that we found a number of data flows from our defined sources and sinks, as shown in table VII. To reproduce these numbers, queries would have to be run on more than 25K VS Code extension databases. Thus we provide smaller scale experiments in Section E.

E. Evaluation

The directory *sample-data* in the repository contains pre-built CodeQL databases that can be used for evaluating the CodeQL queries. Our goal is to demonstrate that our queries are executable and can identify data flows within the packages. As the entire pipeline of running all queries on all 25K

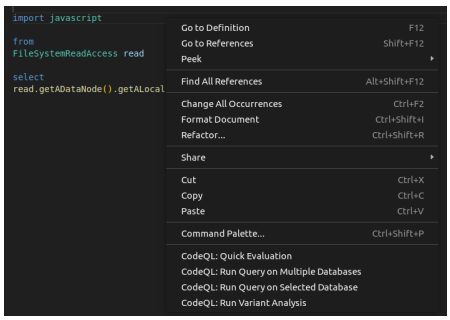

Fig. 8. Menu displayed when right-clicking on query file. Includes option to run query on CodeQL databases.

TABLE VII. NUMBER OF DATA FLOWS FOUND USING CODEQL QUERIES.

Source	Sink	Data Flows
workspace setting	shell	389
	eval	12
	file write	24
file read	shell	75
	eval	34
	file write	150
network response	shell	0
	eval	1
	file write	25
web server	shell	3
	eval	0
	file write	3

packages requires a lot of storage for the packages and setting up an automation framework, we do not expect the entirety of our results to be replicable. We show that these queries can be used to evaluate VS Code extensions and showcase a small part of the results. These experiments can also be run on other databases but would yield different results.

1) Experiment (E1): [5 human-minutes]: This experiment's goal is to show our queries can find our defined sources and sinks in the packages. We will run our queries that identify these sources and sinks on one of the sample databases provided.

[How to] Prepare the database, then execute the query on the database to acquire results.

[Preparation] Load and select the *sample-data/fredericbonnet.cmake-test-adapter-0.16.3* database into the VS Code Workspace using the CodeQL extension as detailed in Section C2.

[Execution] A right-click on the folder with the query files will show a menu including the option to run the query, as shown in figure 8. Run all the queries in the subdirectory *queries-source-and-sink* on the database by selecting `CodeQL: Run Queries in Selected Files` from the menu. The order of query execution does not matter in this experiment.

[Results] The results from the query *workspace-setting-api.ql* will display in the VS Code workspace and should have identified 4 instances of this source in the database. The *shell.ql* query should have identified 2 instances of this source. The *file-read.ql* query should have identified 1 file read source.

2) Experiment (E2): [5 human-minutes]: This experiment's goal is to find a data flow from web server API to a file write. We will run our query for this specific data flow on one of the sample databases provided.

[How to] Prepare database and execute the query on it to acquire results.

[Preparation] As detailed in Section C2, load and select the *sample-data/usqlextpublisher.usql-vscode-ext-0.2.15* database.

[Execution] A menu including the option to run the query will be available when right-clicking on the query file *queries/dataflow/webServer-to-fileWrite.ql*. Run the query on the database by selecting `CodeQL: Run Queries in Selected Files` from the menu.

[Results] The results from the query will display in the VS Code workspace and should have identified 62 instances of this dataflow in the database.

3) Experiment (E3): [5 human-minutes]: This experiment's goal is to find a data flow from file read to a shell command. We will run our query for this specific data flow on one of the sample databases provided.

[How to] Prepare the database, then execute the query on the database to acquire results.

[Preparation] Load and select the *sample-data/fredericbonnet.cmake-test-adapter-0.16.3* database into the VS Code Workspace using the CodeQL extension, as detailed in Section C2.

[Execution] Run the query *queries/dataflow/fileRead-to-shell.ql* on the database by right-clicking on the query file and selecting `CodeQL: Run Queries in Selected Files` from the menu.

[Results] The results from the query will display in the VS Code workspace and should have identified 12 instances of this dataflow in the database.

4) Experiment (E4): [5 human-minutes]: This experiment's goal is to find a data flow from network response to a file write. We will run our query for this specific data flow on one of the sample databases provided.

[How to] Prepare the database, then execute the query on the database to acquire results.

[Preparation] Load and select the *sample-data/iann0036.live-share-for-aws-cloud9-0.11.4* database into the VS Code Workspace using the CodeQL extension, as detailed in Section C2.

[Execution] Run the query *queries/dataflow/network-to-fileWrite.ql* on the database by right-clicking on the query file and selecting `CodeQL: Run Queries in Selected Files` from the menu.

[Results] The results from the query will display in the VS Code workspace and should have identified 2 instances of this dataflow in the database.