Noname manuscript No.
(will be inserted by the editor)

Come for Syntax, Stay for Speed, Write Secure Code: An Empirical Study of Security Weaknesses in Julia Programs

Yue Zhang · Justin Murphy · Akond Rahman

the date of receipt and acceptance should be inserted later

Abstract ***Context***: Practitioners prefer to achieve performance without sacrificing productivity when developing scientific software. The Julia programming language is designed to develop performant computer programs without sacrificing productivity by providing a syntax that is scripting in nature. According to the Julia programming language website, the common projects are data science, machine learning, scientific domains, and parallel computing. While Julia has yielded benefits with respect to productivity, programs written in Julia can include security weaknesses, which can hamper the security of Julia-based scientific software. A systematic derivation of security weaknesses can facilitate secure development of Julia programs—an area that remains under-explored.

Objective: *The goal of this paper is to help practitioners securely develop Julia programs by conducting an empirical study of security weaknesses found in Julia programs.*

Methodology: We apply qualitative analysis on 4,592 Julia programs used in 126 open-source Julia projects to identify security weakness categories. Next, we construct a static analysis tool called Julia Static Analysis Tool (`JSAT`) that automatically identifies security weaknesses in Julia programs. We apply `JSAT` to

This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2310179, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175.

Yue Zhang
Auburn University, AL, USA
E-mail: yzz0229@auburn.edu

Justin Murphy
Cookeville, TN, USA
E-mail: justindmurphy33@gmail.com

Akond Rahman
Auburn University, AL, USA
E-mail: akond.rahman.buet@gmail.com

automatically identify security weaknesses in 558 open-source Julia projects consisting of 25,008 Julia programs.

Results: We identify 7 security weakness categories, which include the usage of hard-coded password and unsafe invocation. From our empirical study we identify 23,839 security weaknesses. On average, we observe 24.9% Julia source code files to include at least one of the 7 security weakness categories.

Conclusion: Based on our research findings, we recommend rigorous inspection efforts during code reviews. We also recommend further development and application of security static analysis tools so that security weaknesses in Julia programs can be detected before execution.

Keywords empirical study · insecure coding · Julia · security weakness · secure software development

1 Introduction

Scientific software developers prefer scripting languages, such as Python and R due to ease in iterative and exploratory development (Bezanson et al., 2018b). However, to increase program execution speed, scientific software developed in scripting languages needs to be migrated to languages, such as C and Fortran as these languages increase program execution speed (Bezanson et al., 2018b). While such migration usually results in improved program execution speed, it comes with development and maintenance overhead (Bezanson et al., 2018b). The programming language Julia is designed and introduced so that developers do not need to migrate from one language to another, in order to improve program execution speed. Julia is designed to provide programming syntax similar to that of scripting languages, with similar program execution speed of compiled languages with low-level memory access (Jul, 2019; Bezanson et al., 2018b). Julia's appeal for developers is colloquially referred to as *"come for the syntax, stay for the speed"* because Julia provides the ability to develop software in a scripted manner without sacrificing program execution speed (Jul, 2019).

Since its inception in 2012, Julia has experienced growing popularity in recent years (Computing, 2022). According to a survey of Stack Overflow users in 2020, Julia is considered as one of the "*top 10 most loved programming languages*" by practitioners (Julia, 2020). We observe Julia being used in research and product development. For example, Julia was used in Celeste (Julia, 2017; jul, 2017), a software used in astronomy research. Celeste was used to load 178 terabytes of astronomical image data to produce a catalog of 188 million astronomical objects in 14.6 minutes, yielding a performance improvement by a factor of 1,000, compared to prior implementation (Julia, 2017).

Despite reported benefits, Julia programs may include security weaknesses that can potentially lead to serious consequences. Security weaknesses are recurring coding patterns that can make software vulnerable to security attacks. Let us consider Figure 1 in this regard. Figure 1 provides a code snippet from a Julia program found in the open source software (OSS) Julia project `jevo` [1] cloned

[1] https://github.com/tmptrash/jevo

```
1function serialize(s::SerializationState, x::Symbol)
2    tag = sertag(x)
3    if tag > 0
4        return write_as_tag(s.io, tag)
5    end
6    pname = unsafe_convert(Ptr{UInt8}, x)
7    ln = Int(ccall(:strlen, Csize_t, (Cstring,), pname))
8    if ln <= 255
9        writetag(s.io, SYMBOL_TAG)
10       write(s.io, UInt8(ln))
11   else
12       writetag(s.io, LONGSYMBOL_TAG)
13       write(s.io, Int32(ln))
14   end
15   unsafe_write(s.io, pname, ln)
```

Fig. 1: Example of a security weakness (unsafe invocation) in a Julia project

from GitHub [2]. Considering Figure 1, a security weakness can be observed at line 6 of the code snippet. The `unsafe_convert()` function on line 6 is used to convert a Julia object to a pointer (Julia, 2021c). `unsafe_convert()` function is later used by `pname`, which in turn is used by `ln` in line 6. Another security weakness is observed on line 15 where the `unsafe_write()` function is used.

Figure 1 demonstrates an example of a security weakness in Julia source code files, which provides malicious users the opportunity to perform a unsafe invocation attack (Julia, 2021c; Boxler, 2018). According to the Common Weakness Enumeration (CWE) (MITRE, 2021j), *"when software allows a user's input to contain code syntax, it might be possible for an attacker to craft the code in such a way that it will alter the intended control flow of the software. Such an alteration could lead to arbitrary code execution"* (MITRE, 2021i).

Over the last two decades, experts in the domain of scientific software development have advocated for integration of quality assurance activities, such as code review and testing for software (Heroux et al., 2007; Kelly et al., 2008, 2011; Morris, 2008). Such advocacy was echoed recently by Heymann et al. (2023), who observed that practitioners who develop scientific software lack necessary knowledge and tools to secure software that are used for scientific experiments. Milewicz et al. (2022) emphasized on early integration of secure development practices for scientific software mentioning "*Not securing software right from the development stage is like putting a deadbolt on a cardboard door*". These guidelines from experts is applicable for Julia programs as well, as these programs are frequently used in scientific software development (Rahman et al., 2023a; Farhana et al., 2019; Rahman et al., 2023a). A systematic analysis of security weaknesses for Julia programs will lay the groundwork for secure development of Julia programs by detecting and mitigating security weaknesses in Julia programs similar to that in Figure 1. Such analysis, which remains under-explored (Rahman et al., 2023a), can help practitioners to (i) perform security code reviews for Julia programs, and (ii) conduct static analysis to identify security weaknesses in an automated way for Julia programs.

[2] https://github.com/

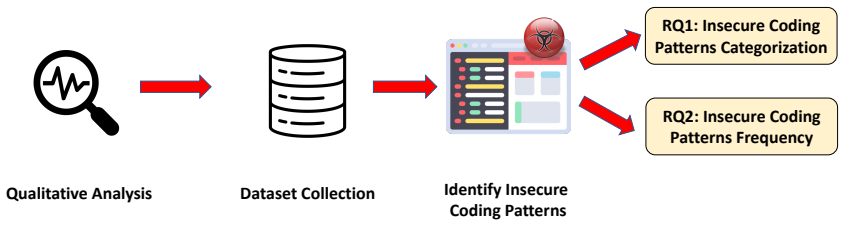

Fig. 2: An overview of our methodology.

Existence of security weaknesses in Julia source code files similar to that of Figure 1 can be detrimental to the security of Julia-based software projects. Secure coding is important for Julia-based projects as recently in 2021, a Julia-related vulnerability impacted 56,703 IP addresses (cvedetails, 2021). The vulnerability was rated to be 'critical' (cvedetails, 2021). These evidence motivate our empirical study of investigating security weaknesses in Julia projects.

The goal of this paper is to help practitioners securely develop Julia programs by conducting an empirical study of security weaknesses found in Julia programs.

We answer the following research questions:

- ***RQ-1 (Categorization)**: What categories of security weaknesses appear for Julia programs?*
- ***RQ-2 (Frequency)**: How frequently do the identified security weaknesses appear for Julia programs?*

We derive security weakness categories for Julia programs by applying a qualitative analysis technique called open coding (Saldaña, 2015) with 4,592 Julia programs obtained from 126 OSS Julia project repositories. Next, we construct a security static analysis tool called Julia Static Analysis Tool (`JSAT`) that automatically identifies security weaknesses in Julia programs. We apply `JSAT` to identify security weaknesses in 25,008 Julia programs from 558 open-source Julia programs. Dataset and source code used for our paper is publicly available online (Rahman et al., 2023c).

Contributions: We list our contributions as follows:

- A derived list of security weakness categories in Julia source code files; and
- An evaluation of security weakness frequency in Julia source code files.

We organize rest of the paper as follows: We provide the methodology and results related to RQ-1 in Section 2. We provide the methodology and results related to RQ-2 in Section 3. We discuss our findings, limitations, and related work respectively in Section 4, 5, and 6. Finally, we conclude the paper in Section 7. An overview of our methodology is presented in Figure 2.

2 Security Weakness Categories in Julia Programs

Our paper showcases an empirical study with a focus on identifying security weaknesses in Julia programs. In this section, we answer *RQ-1: What categories of security weaknesses* appear for Julia programs? We provide necessary background, methodology and results respectively, in Sections 2.1, 2.2 and 2.3.

2.1 Background

We provide necessary background on Julia source code files and CWE in the following subsections.

2.1.1 Julia Source Code Files

Julia is an emerging programming language with over 25 million downloads and over 5,000 Julia packages registered by the Julia community, including various mathematical libraries, data manipulation tools, and packages for general-purpose computing, for community use (Julia, 2021b). Julia is recognized to solve the "two language problem" (Bezanson et al., 2018a, 2017), referring to the circumstance where practitioners have to switch to a programming language that is more difficult to use in order to achieve better performance. For example, because of its scripting nature, writing programs in Python can be relatively straightforward for practitioners. However, this may come with the sacrifice of program execution time as Python programs may not be as fast as C programs. Rapid program execution is a desirable characteristic when employing resource-intensive, computational programs on large datasets, as done by the Celeste project (Julia, 2017). Python provides libraries to process large datasets, however, when it comes to performance compared to C, Python has its limitations. Julia was designed to provide both friendly syntax as well as speed. According to Perkel (2019), *"Julia circumvents that two-language problem because it runs similar to C, but reads similar to Python. And it includes built-in features to accelerate computationally intensive problems, such as distributed computing, that otherwise require multiple languages"*.

Julia programs take advantage of Just-In-Time (JIT) compilation, which is the process of compiling lines of code sequentially as they are seen, instead of compiling all lines beforehand. Using JIT compilation, Julia is perceived to be useful in developing computationally efficient programs. Julia programs are also compiled into an intermediate representation of byte-code, allowing for portability between different computer architectures. Julia programs can also call low-level functions from the C run-time.

Let us consider the annotated example of a Julia program in Figure 3. Similar to general purpose programming languages (GPLs), Julia has dedicated code elements, such as `include` on line 6 and `println` on line 8, which are used to specify dependencies and redirect program output to the console. A collection of Julia programs is referred to as a package. Functions in Julia are defined using the `function` keyword, as seen on line 10. Julia allows the return of one or multiple values without explicitly specifying the `return` keyword, as long as the values to

```
1# Pre-compilation enabled
2__precompile__()
3# Create Example module
4module Example
5    # Include a package dependency
6    include("example.jl")
7    # Hello world program
8    println("Hello World")
9    # Function to add two values
10   function add_and_sub(a, b)
11       sum = a+b
12       dif = a-b
13       sum, dif
14   end
15
16   ans1, ans2 = add_and_sub(11, 7)
17   # result: 18, 4
18end
```

Fig. 3: An annotated example of a Julia program

be returned are on the last line in the function body. For example, in Figure 3, the function `add_and_sub` performs two mathematical operations, addition and subtraction, and returns the results of those two operations by using the statement `'sum, dif'` as the last line in the function body on line 16.

2.1.2 CWE

CWE is a community-driven database for software security weaknesses and vulnerabilities (MITRE, 2021j). The database is owned by the MITRE Corporation, with support from US-CERT and the National Cybersecurity Division of the United States Department of Homeland Security (MITRE, 2021j). The intention behind creation of the database is to aid the software community in understanding security weaknesses in software, creating automated tools so that security weaknesses in software can be automatically identified and addressed, and creating a common baseline standard for security weakness identification, mitigation, and prevention efforts (MITRE, 2021j).

2.2 Methodology for RQ-1

We answer RQ-1 by applying a qualitative analysis approach as summarized in Figure 4. The qualitative analysis technique that we apply is called open coding (Saldaña, 2015), which we use with open source Julia programs. Indeed, application of qualitative analysis techniques, such as open coding (Saldaña, 2015) is applicable for any programming language, but here the difference is that we allocated a rater who is an expert in Julia and secure software development to derive security weakness categories. Through this exercise, we focus on deriving security weakness categories that are applicable for Julia programs. Because of this particular activity, a practitioners who uses Julia programs will not have to learn about CWEs themselves and identify security weaknesses.

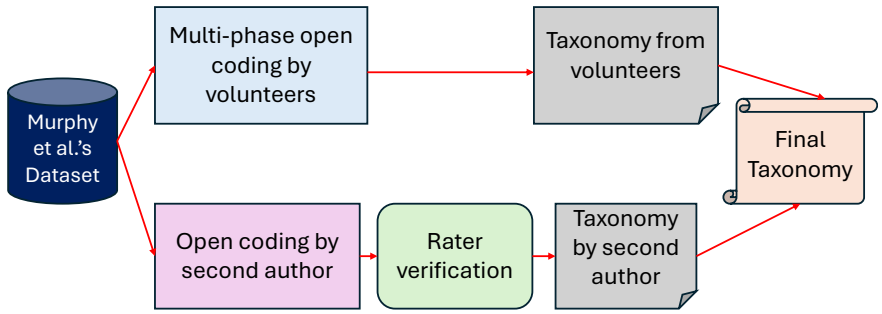

Fig. 4: An overview of the qualitative analysis process used to answer RQ-1.

Security weaknesses are recurring coding patterns that can make a software vulnerable to security attacks. Security weaknesses may not always lead to a security breach, but manual inspection is still a necessary undertaking (Rahman et al., 2019c, 2021a).

2.2.1 Qualitative Analysis

Instructions for Raters Performing Open Coding We conduct the following:

1. we confirm that the rater has familiarity with security weaknesses;
2. we provide the CWE database (2021j) to the rater so that they know what security weakness categories can occur for software;
3. we provide a tutorial [3] for the rater so that they can be familiar with the common code constructs of Julia;
4. upon being familiar with CWE entries and Julia code constructs, for each source code file the rater performs the following:
   (a) inspect each line of code in the source code file;
   (b) isolate any coding pattern that could be of indicative a security weakness based on rater knowledge; and
   (c) map a CWE for the identified pattern, if there is no mapping then discard the pattern. We use CWE entries here for validation purposes as a rater's knowledge can bias the security weakness derivation procedure. According to the GitHub Advisory website, only one CVE ('CVE-2021-4048') [4] is reported, which may not be adequate for deriving the common security weaknesses. That is why we asked each rater to familiarize with CWE entries. While deriving the security weakness categories, it is necessary to validate the open coding process. To substantiate the derived categories, we map each identified category to at least one CWE entry (MITRE, 2021j). For

[3] https://syl1.gitbook.io/julia-language-a-concise-tutorial
[4] https://github.com/advisories/GHSA-wgf2-cvhg-c384

example, the derived security weaknesses category shown in Figure 5, weak encryption, maps to "CWE-327: Use of a Broken or Risky Cryptographic Algorithm" (MITRE, 2021d). By mapping between security weakness category and at least one corresponding CWE entry, we are able to validate the open coding process.

5. groupify all identified coding patterns using open coding by
   (a) First, identifying initial codes;
   (b) Second, identifying initial categories; and
   (c) Third, deriving categories from initial categories.

To keep track of the security weaknesses, raters maintain a spreadsheet. A snapshot of the spreadsheet is available in Table 1.

Table 1: A Snapshot of the Spreadsheet Used by Raters While Performing Open Coding

Coding Pattern	**CWE ID**	**File Name**	**Initial Code**	**Initial Category**
`getindex(a::UnsafeMatrixView, i::Int) = unsafe_load(a.ptr, i)`	CWE-94	`NumericExtensions.jl/src/unsafe_views.jl`	`unsafe_load`	Unsafe Invocation
`ptr = Base.unsafe_convert(PtrT, x)`	CWE-94	`MPI.jl/src/datatypes.jl`	`unsafe_convert`	Unsafe Invocation
`ptr = Base.unsafe_convert(PtrT, x)`	CWE-94	`MPI.jl/src/datatypes.jl`	`unsafe_convert`	Unsafe Invocation
`unsafe_store!(convert(Ptr$(T.types[i]), out)+$jloffset, read_ref(file, ref))`	CWE-94	`JLD.jl/src/jld_types.jl`	`unsafe_store!`	Unsafe Invocation
`filehash = sha1(f)`	CWE-321	`OpenGene.jl/src/Reference/Genome/downloader.jl`	`sha1(f)`	Weak Encryption
`push!(args, a), _args) # FIXME: append! doesn't support sets`	CWE-546	`Latte.jl/src/net.jl`	`# FIXME: append! doesn't support sets`	Suspicious Comments

Demonstration of Open Coding We determine categories of security weaknesses in Julia programs by conducting a qualitative analysis technique called open coding (Saldaña, 2015). In open coding, a rater observes and synthesizes patterns within structured or unstructured text (Saldaña, 2015). We apply open coding to recognize patterns that are indicative of a security weakness in Julia programs contained in OSS Julia programs downloaded from GitHub. By applying open coding, we assume to be able to determine which categories of security weaknesses appear for Julia programs, as well as obtain context on how the identified security weaknesses can be detected in an automated way.

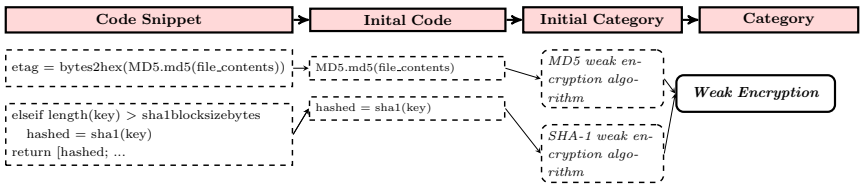

Fig. 5: Example to demonstrate the process of determining security weakness categories in Julia source code files.

Figure 5 provides an example of how the open coding process is performed for Julia programs. First, we extract Julia code snippets and separate out the raw text. Here, the raw text is unaltered text obtained for each source code file. We extract source code file content from each source code file. For example, we separate the raw text of `filehash = sha1(f)` and `hashed = sha1(key)` from the two code snippets shown in Figure 5. Next, we determine an initial category for each of the raw texts. The two initial categories based on the raw text are `'SHA-1 weak encryption algorithm'` and `'SHA-1 weak encryption algorithm'`, derived from the use `sha1` from the Julia standard library (Jul, 2022). The methods associated with `SHA-1` used in the code snippets of Figure 5 is a weak encryption algorithm (OWASP, 2021b). Finally, as both initial categories `'SHA-1 weak encryption algorithm'` and `'SHA-1 weak encryption algorithm'` is related to weak encryption algorithms, we merge the two initial categories into one final category called 'weak encryption'.

The last author further checked the rating of the second author who performed open coding. As part of this inspection process, the last author inspected if the definition of the mapped CWE is in fact applicable for the derived category and the code snippet that is used to derive the category. Upon completion of this inspection process, the last author found three categories identified by the second author that do not directly map to CWE-provided definitions. These three categories are: hard-coded user name, default port, and unrestricted IP address.

2.2.2 Dataset Collection

We apply open coding on all 4,592 Julia programs contained in the 126 OSS repositories provided by Murphy et al. (2020). We select Murphy et al.'s (2020) dataset because it is curated and systematically filtered based on prior research (Munaiah et al., 2017), and can be used for research studies. The open coding process took 665 hours to complete.

2.2.3 Rater Verification

Our open coding process is subject to rater bias as these categories are derived by one rater. We mitigate this limitation by allocating another rater, who is not an author of the paper. The rater is a software engineer working for a defense contractor, with three years of professional software experience. We allocate randomly-selected 500 Julia source code files to the additional rater. By selecting these 500

randomly-selected Julia source code files, we obtain a 95% confident level for our set of 4,592 Julia source code files.

The rater is asked to inspect each of the 500 randomly-selected Julia source code files, and map them to one or multiple identified security weakness categories. The rater was provided a document with examples, definitions for each security weakness category. We record a Cohen's Kappa (Cohen, 1960) of 0.79, indicating 'substantial' agreement according to Landis and Koch (1977). In the case of disagreements, the first author acted as the resolver. The first author's decision is final in this regard. The agreement level 'substantial' is based on Landis and Koch (1977) interpretation of Cohen's Kappa. According to this interpretation, the rater have agreed on majority on the items that were asked to label, but there were disagreements as well.

2.2.4 Additional Open Coding

The reason of using one rater is because of the lack of rater availability, i.e., finding a volunteer who has academic experience in securing coding and willing to spend time voluntarily to derive security weakness categories by inspecting each of the 4,592 Julia source code files. While we have mitigated this limitation by using another rater with 500 randomly-selected Julia source code files, this activity is limiting as the additional rater only inspected 500 of the total set. As such, we allocate two additional raters who are graduate students in the department to perform an additional round of open coding. Both students are enrolled in M.Sc. in Cybersecurity program, and voluntarily agreed to participate in the open coding process. Both students have taken courses that teaches secure coding, such as 'Secure Software Process' and 'Computer and Network Security'. We apply open coding in two phases as multi-phase open coding (Hickey and Kipping, 1996; Sweeney et al., 2013) facilitates rater reliability and achieves rater consensus. The two phases are synchronized open coding and independent open coding, which are discussed as follows:

1. *Synchronized open coding*: As part of this open coding process, both rater conduct the open coding process in collaboration where they discuss with each other while deriving security weakness categories. As part of the open coding process, each rater individually inspect each of the 2,296 Julia source code file to determine is a security weakness is present and if the determined security weakness category is not reported in the taxonomy. After the open coding process is complete, we compute a Cohen's Kappa (1960) of 0.92 indicating 'almost perfect' agreement (Landis and Koch, 1977). The disagreements are resolved by the last authors of the paper who has 8 years of experience in secure coding. The last author's categorization is final for the files that the raters are disagreed upon. Upon completion of the synchronized open coding phase we do not identify any security weakness categories that have not been identified by the second author.
2. *Independent open coding*: As part of this open coding process the two raters do not coordinate or discuss during the open coding process. Each rater inspects the remaining 2,296 Julia source code files that are not used in the synchronized open coding phase. Upon completion of the open coding process, we observe a

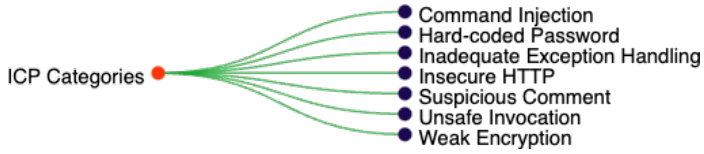

Fig. 6: Identified security weakness Categories in Julia Programs.

Cohen's Kappa (1960) of 0.68 indicating 'substantial' agreement (Landis and Koch, 1977). The disagreements are resolved by the last authors of the paper similar to that of the synchronized open coding process. The last author's categorization is final for the files that the raters are disagreed upon. Upon completion of the independent open coding phase we do not identify any security weakness categories that have not been identified by the second author.

2.3 Answer to RQ-1: Security Weakness Categories in Julia Source Code Files

In this section, we answer *RQ-1: What categories of security weaknesses appear for Julia programs?* As shown in Figure 6, we identify 7 security weakness categories for Julia programs through the open coding (Saldaña, 2015) process described in Section 2.2.1. We provide definitions, descriptions, examples, and mapped CWE entries for each of the 7 security weakness categories as follows.

I. **Command Injection**: This category is the recurring pattern of using certain methods that facilitate the execution of arbitrary commands in Julia programs. Command injection allows attackers to execute dangerous commands directly on the operating system (MITRE, 2021g). An attacker can use a command injection attack to acquire privileges, allowing the attacker to gain unauthorized access to execute a privileged program (MITRE, 2021g). For Julia programs, the `run(cmd)` method can be used to execute arbitrary Julia commands and gain control of the operating system (Wallace, 2016). Julia-specific coding patterns that facilitate execution of arbitrary commands are: `run(cmd), eval(parse()), and remote_caller()`. The category corresponds to "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')" (MITRE, 2021g).

Example: Listing 1 provides an example of `command injection` in a Julia source code file. Here, in line #2, `eval(parse())` is the security weakness, which is used to assign a value to `keytype`. `keytype` is later used by `h2` in line# 3.

```
1 for ikey = 1 : length(key)
2     keytype = eval(parse("typeof( Seismic.InitSeisHeader().$(
      ↪  string(key[ikey])))"))
3     h2 = reinterpret(keytype,vec(h1[ikey,:]))
4     a = minimum(h2)
5     b = maximum(h2)
```

Listing 1: An example of `command injection` in a Julia source code file.

II. **Hard-Coded Password**: This category is the recurring pattern of using hard-coded passwords. The use of a hard-coded password can increase the possibility that encrypted data is uncovered by a malicious actor (MITRE, 2021h). Hard-coded passwords allow an attacker to bypass any authentication configured by the software administrator (MITRE, 2021h). The category corresponds to "CWE-798: Use of Hard-coded Credentials" (MITRE, 2021h).

Example: Listing 2 provides an example of a `hard-coded password` in a Julia source code file. In line #5, the hard-coded password is `mysql_pswd = ''root''` that is used to setup a database connection string.

```
1 #
2 # Create MySQL DB connection
3 host = "127.0.0.1";
4 mysql_usr = "root";
5 mysql_pswd = "root";
6 dbname = "pubmed_obesity_2010_2012";
7
8 const mysql_conn = DBUtils.init_mysql_database(host, mysql_usr, mysql_pswd,
  ↪  dbname) # hide
9 PubMed.create_tables!(mysql_conn) # hide
```

Listing 2: An example of `hard-coded password` in a Julia source code file.

III. **Inadequate Exception Handling**: This category is the recurring pattern of using generic exception handlers or using no exception handlers at all by leveraging `println()`. The use of generic exceptions promotes complex error handling code that is more likely to contain security vulnerabilities (MITRE, 2021e). According to CWE, not using an exception handler allows a malicious user to trigger unexpected conditions "*thus violating the programmer's assumptions, possibly introducing instability, incorrect behavior, or a vulnerability*". The category corresponds to "CWE-754: Improper Check for Unusual or Exceptional Conditions" (MITRE, 2023).

Example: Listing 3 provides an example of `inadequate exception handling` in a Julia source code file. Here, in line #2, we observe a specific exception not being thrown, which is an example of inadequate exception handling.

```
1 p = prime_decomposition(OK, 2)[1][1]
2 @test_throws ErrorException  Hecke.locally_free_basis(I, p)
```

Listing 3: An example of `inadequate exception handling` in a Julia source code file.

IV. **Insecure HTTP**: This category is the recurring pattern of using HTTP without Transport Layer Security (TLS) or HTTPS without Secure Sockets Layer (SSL) protection. If sensitive data is transmitted without TLS protection with HTTP, the data is transmitted in clear-text and could be sniffed by a malicious actor (MITRE, 2021b). Coding patterns found in Julia programs include the use

of the HTTP protocol without TLS, or SSL is set to false when using HTTPS protocol. The category corresponds to "CWE-319: Cleartext Transmission of Sensitive Information" (MITRE, 2021b).

Example: Listing 4 provides an example of `insecure HTTP` in a Julia source code file. The example of insecure HTTP in line #6 is used to obtain and extract JSON data.

```
1 push!(v.data, VegaData(name = "geodata",
2     url = "http://trifacta.github.io/vega/data/ us-10m.json",
3     format = VegaFormat(_type = "topojson", feature = "states"),
4     transform = [VegaTransform(Dict{Any, Any}("type" => "geopath", "projection"
      ↪  => "albersUsa")),
5                  VegaTransform(Dict{Any, Any}("type" => "lookup", "on" => table,
                   ↪  "onKey" => "x", "keys" => ["id"], "as" => ["table2"])),
6                  VegaTransform(Dict{Any, Any}("type" => "filter", "test"
                   ↪  =>"datum.layout_path!=null && datum.table2!=null"))]
```

Listing 4: An example of `insecure HTTP` in a Julia source code file.

V. **Suspicious Comments**: This category is the recurring pattern of including information in comments relating to malfunctioning or missing features of a Julia project. Suspicious comments indicate the referenced code could possibly be exploited by malicious users (MITRE, 2021f). Coding patterns found in Julia programs include putting keywords, such as `'FIXME'` in the comments of the source-code. Including keywords, such as `'TODO'`, in comments is a common practice in software engineering, but can lead to negative consequences, such as introducing bugs in software source code (Tan et al., 2007; Storey et al., 2008). The category corresponds to "CWE-546: Suspicious Comment" (MITRE, 2021f).

The motivation of including this particular category is because of its recurrence observed during our open coding process and an entry in the CWE database (MITRE, 2021j) According to CWE, "*Suspicious comments could be an indication that there are problems in the source code that may need to be fixed and is an indication of poor quality. This could lead to further bugs and the introduction of weaknesses*" (MITRE, 2021f). This indicates that suspicious comments can propagate and trigger more weaknesses in software source code. The derivation of this category is also consistent with other research studies (Rahman et al., 2021a; Rahman and Williams, 2021) related to security weaknesses.

Example: Listing 5 provides an example of a suspicious comment. The code snippet in line #1 shows that the implementation for `get_main_mode` needs to be improved so that when the module is changed `REPL` content can be reprinted.

VI. **Unsafe Invocation**: This category is the recurring pattern of using methods that do not validate user input in Julia programs. Unsafe invocation occurs when the code allows a user's input to not be validated, and therefore it could be possible for an attacker to provide malicious code that will alter the intended function of the program (MITRE, 2021i). Unsafe injection is different from command injection command injection does not involve the attacker's code being executed by the application, but rather the attacker extends the default functionality of

```
# FIXME: Find a way to reprint what's currently entered in the REPL after
  ↪  changing
#        the module (or delete it in the buffer).

using Logging: with_logger
using .Progress: JunoProgressLogger

function get_main_mode()
  mode = Base.active_repl.interface.modes[1]
  mode isa LineEdit.Prompt || error("no julia repl mode found")
  mode
end
```

Listing 5: An example of `suspicious comment` in a Julia source code file.

the application to execute system commands (OWASP, 2021a). Unsafe invocation can occur in Julia programs through the use of the **'unsafe'** prefix for particular functions (Julia, 2021c). Julia-specific coding patterns that does not provide validation for user input are: **'Base.unsafe_load'**, **'Base.unsafe_convert'**, **'Base.unsafe_store!'**, **'Base.unsafe_copy!'**, **'Base.unsafe_pointer_to_objref'**, and **'Base.unsafe_wrap'**. The category corresponds to "CWE-94: Improper Control of Generation of Code ('Unsafe Invocation')" (MITRE, 2021i).

The four functions that are related to unsafe invocation can be used to interface with C functions (Jul, 2022). These functions are used when developers want to interface the Julia program with a C program. The 'unsafe' prefix has different implications for different 'unsafe' functions. In particular,

- For `unsafe_load` the 'unsafe' prefix indicates that the functions itself does not perform any validation on the pointer p that is provided as an input parameter to the function. It is crucial to ensure that the pointer is valid and points to the expected memory location so that program crashes or incorrect results can be avoided (JLHUB, 2024).
- For `unsafe_convert`, the 'unsafe' prefix indicates that the result of this function is no longer accessible to the program, which may cause undefined behavior, including segmentation faults, at any later time (JLHUB, 2024).
- For `unsafe_store!` the 'unsafe' prefix indicates that no validation is performed on the pointer p that is passed as an input parameter to the function. Incorrect usage of this function may cause a segmentation fault, similar to that of C programs (JLHUB, 2024).
- For `unsafe_copy!` the 'unsafe' prefix indicates that no validation is performed to ensure that the parameter N is within the limits of the array. Incorrect usage may cause segmentation fault because of out-of-bounds errors (JLHUB, 2024).
- For `unsafe_pointer_to_objref` the 'unsafe' prefix indicates that if not handled adequately, undefined behavior will result when the pointer is passed to the function. The pointer that is passed to the function refers to a valid heap-allocated Julia object (JLHUB, 2024).

– For `unsafe_wrap` the 'unsafe' prefix indicates that the function will crash if the provided pointer is not a valid memory address to data of the requested length. The function wraps an `Array` object around the data at the address given by the pointer without making a copy (julia, 2024).

Example: Listing 6 provides an example of `Unsafe invocation` in a Julia source code file. The program is used to calculate the dimension of an array `a`. In line# 5, `unsafe_load` is used to load a value from the i address starting at $a.ptr$.

```
1 size(a::UnsafeCubeView) = (a.dim1, a.dim2, a.dim3)
2 size(a::UnsafeCubeView, d::Int) = d == 1 ? a.dim1 : d == 2 ? a.dim2 : d == 3 ?
  ↪  a.dim3 : 1
3 length(a::UnsafeCubeView) = a.len
4
5 getindex(a::UnsafeCubeView, i::Int) = unsafe_load(a.ptr, i)
```

Listing 6: An example of `unsafe invocation` in a Julia source code file.

VII. **Weak Encryption**: This category is the recurring pattern of using cryptographic algorithms that are weak or have been proven to be broken (OWASP, 2021b) to encrypt sensitive information. The use of a broken or risky cryptographic algorithm may result in the exposure of confidential information (MITRE, 2021a,c,d). This category includes security weaknesses that use the `md5()` function, and the `sha-1()` function. The category corresponds to "CWE-311: Missing Encryption of Sensitive Data", "CWE-321: Use of Hard-coded Cryptographic Key", and "CWE-327: Use of a Broken or Risky Cryptographic Algorithm" (MITRE, 2021a,c,d).

Example: Listing 7 provides an example of `weak encryption` in a Julia source code file. We observe in line #4, `sha1` to be used to hash the contents of a file.

```
1 function check_sha1(file, hash)
2     info("checking SHA1...")
3     f = open(file)
4     filehash = sha1(f)
5     # work around for SHA.sha1() incompatibility
6     if isa(filehash, Array{UInt8,1})
7         filehash = bytes2hex(filehash)
8     end
9     if lowercase(filehash) == lowercase(hash)
10        info("SHA1 OK")
11        return true
12    else
13        warn("wrong hash, expect $hash, but got $filehash")
14        return false
15    end
16    return false
17 end
```

Listing 7: An example of `weak encryption` in a Julia source code file.

We report the exploit likelihood for each identified security weakness category in Table 2. We observe four categories to have a 'high' likelihood of being exploited: command injection, hard-coded password, insecure HTTP, and weak encryption. Results reported in Table 2 further showcase that our derived security weakness categories have relevance as four of the seven security weakness categories are highly likely to be exploited.

Table 2: Exploit Likelihood of Identified Security Weakness Categories

Category	**CWE ID**	**Exploit Likelihood**
Command Injection	CWE-78	'High' (MITRE, 2021g)
Hard-Coded Password	CWE-798	'High' (MITRE, 2021h)
Inadequate Exception Handling	CWE-754	'Medium' (MITRE, 2023)
Insecure HTTP	CWE-319	'High' (MITRE, 2021b)
Suspicious Comments	CWE-546	Not Available
Unsafe Invocation	CWE-94	'Medium'
Weak Encryption	CWE-327	'High' (MITRE, 2021d)
Weak Encryption	CWE-321	'High' (MITRE, 2021c)
Weak Encryption	CWE-311	'High' (MITRE, 2021a)

We have reported which of our derived security weakness categories appear for other programming languages. The details are available in Table 3.

Table 3: Comparison of Security Weakness Categories Across Languages

Category	**Julia [This Paper]**	**Ansible (Saavedra and Ferreira, 2023)**	**Chef (Rahman et al., 2021a)**	**Kubernetes (Rahman et al., 2023b)**	**Puppet (Rahman et al., 2019b)**	**Python (Ruohonen et al., 2021)**
Command Injection	Y	N	N	N	N	Y
Hard-Coded Password	Y	Y	Y	Y	Y	Y
Inadequate Exception Handling	Y	N	N	N	N	Y
Insecure HTTP	Y	Y	Y	Y	Y	N
Suspicious Comments	Y	Y	Y	N	Y	N
Unsafe Invocation	Y	N	N	N	N	N
Weak Encryption	Y	Y	Y	N	Y	Y

In Table 3, we observe the categories of security weaknesses vary from one programming language to another. The category that is common across all languages is hard-coded password. Four of the seven categories identified for Julia is observant for Ansible, Chef, Puppet, and Python. In the case of Ansible and Kubernetes, the programs are written in YAML format, whereas Chef and Puppet have a Ruby-like syntax. As all Python, Ruby, and YAML are more mature programming languages that Julia, the likelihood of finding static analysis tools are higher than that of Julia. However, we observe that not all identified security weakness categories for Julia appear solely in one programming language. Because of this observation and due to syntactic and semantic differences, we have no option to use an existing tool for these languages. Therefore, we needed to develop a security static analysis tool called `JSAT`.

Answer to RQ-1: We identify 7 categories of security weaknesses in Julia programs: command injection, hard-coded password, inadequate exception handling, insecure HTTP, suspicious comments, unsafe invocation, and weak encryption.

3 Frequency of Security Weaknesses in Open Source Julia Source Code Files

In this section, we answer *RQ-2: How frequently do the identified security weaknesses appear for Julia programs?* We answer RQ-2 by investigating the structure of Julia programs so that we can automatically identify security weaknesses in Julia programs. Use of parse trees for security weakness detection is commonplace but as evident from our discussion in Section 3.1, there exists no tools for Julia that can detect security weaknesses. Our construction of `JSAT` addresses that gap. Construction of the tool `JSAT` required usage of Julia-specific parsers and gaining an understanding of Julia-specific code constructs.

3.1 Methodology for RQ-2

`JSAT` is a static analysis tool that will automatically identify the 7 security weakness categories identified and described in Section 2.3 for Julia programs. As input, a practitioner will provide the file path where the Julia repositories that are to be evaluated reside, and `JSAT` will output to a CSV file the count of each detected security weakness for each Julia program contained in the repositories.

Two factors motivate the construction of `JSAT`:

1. We observe that there does not exist a static analysis tool dedicated to the Julia programming language (OWASP, 2021c; NIST, 2021b). We also have have explored for static analysis tools for Julia programs and found two tools namely, 'StaticLint.jl' (julia-vscode, 2023) and 'Jet.jl' (aviatesk, 2023) We have downloaded and explored the code of these two tools. From our investigation, we find that none of the seven categories of security weaknesses are detected by the two tools. A comparison of between JSAT and the two tools is provided in Table 4. 'StaticLint.jl' identifies code smells, such as unused binding, unused variables, and default keyword mismatch. 'JET.jl' identifies type-related defects in Julia programs, such as type instability and erroneous type usage.
2. There are fundamental differences between Julia and other GPLs with respect to syntax and semantics (Bezanson et al., 2017, 2018a; Zappa Nardelli et al., 2018), and therefore there is a need to construct a tool that will detect security weaknesses in Julia programs.

For certain programming languages, such as Python there exists linters that also detects security weaknesses. For example, Pylint [5] can be used to detect instances of inadequate exception handling in Python programs. Our assumption is that

[5] https://pylint.readthedocs.io/en/

Table 4: Comparison of JSAT, 'StaticLint.jl', and 'JET.jl' With Respect to Detecting Identified Security Weakness Categories

Category	JSAT	StaticLint.jl	JET.jl
Command Injection	Yes	No	No
Hard-Coded Password	Yes	No	No
Inadequate Exception Handling	Yes	No	No
Insecure HTTP	Yes	No	No
Suspicious Comments	Yes	No	No
Unsafe Invocation	Yes	No	No
Weak Encryption	Yes	No	No

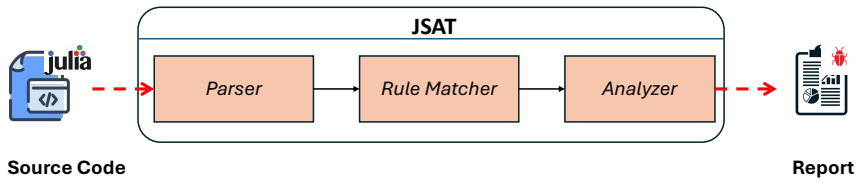

Fig. 7: Components of JSAT.

by exploring two popular linters for Julia, namely StaticLint.jl and JET.jl we too would be able to to identify a set of security weaknesses derived as part of our taxonomy. However, from Table 4 we observe that none of these tools is capable of detecting the identified security weakness categories.

We by no means are advocating against the usage of StaticLint.jl and JET.jl. Both are popular static analysis tools, where StaticLint.jl can be used to extract source code metrics from a Julia source code file, and JET.jl can be used to detect type-related defects, such as type instability (Rahman et al., 2023a). In short, StaticLint.jl and JET.jl includes multiple rules that are useful for Julia-based software development but do not detect our identified set of security weakness categories.

Figure 7 shows the components of JSAT. As input, JSAT takes one or multiple Julia source code files as input. Upon receiving the input, JSAT's 'Parser' component applies lexical analysis to determine is the provided program is syntactically correct. If it is, then JSAT extracts tokens from the code and generates an abstract syntax tree. Next, the 'Rule Matcher' component matches a set of rules written in first-order logic to detect certain patterns in the code that are indicative of security weaknesses. Finally, the 'Analyzer' component applies def-use chain analysis to determine if the detected pattern is actually being used by other portions of the program. If yes, then the reported pattern is reported tot he Julia user as a valid security weakness in forms of a report. None of the three components use any deep learning techniques.

3.1.1 JSAT's Security Weakness Detection Process

JSAT detects security weaknesses in three steps:

Table 5: Example rules for detecting security weaknesses in Julia programs

Security Weakness Category	**Rule**
Command Injection	*isRunCommand(x) OR isEval(Parse(x)*
Hard-Coded Password	*(isKey(x) OR isJLVar(x)) AND len(x.value)>0 AND (isPassword(x))*
Inadequate Exception Handling	*isJLCatchBlock(x) AND isGenExcept(x.value) OR len(x.value) == 0*
Insecure HTTP	*isKey(x) AND isHTTP(x.value)*
Suspicious Comments	*isJLComment(x) AND hasSuspWord(x)*
Unsafe Invocation	*hasUnsafePrefix(x) AND hasPtrUsedLater(x)*
Weak Encryption	*isJLEncodeMethod(x) OR isJLDecodeMethod(x) AND (isMD5(x) OR isSHA-1(x)*)

Table 6: String patterns used for functions in rules

Function	**String Pattern**
hasUnsafePrefix()	'unsafe_convert' 'unsafe_load', 'unsafe_store!', 'unsafe_copy!', 'unsafe_wrap', 'unsafe_pointer_to_objref'
hasPtrUsedLater	'ptr'
isRunCommand()	'run'
isEval()	'eval'
isJLCatchBlock	'catch'
isJLEncodeMethod	'encode'
isJLDecodeMethod	'decode'
isParse()	'parse'
isPassword()	'password', 'pswd', 'psswd'
isGenExcept()	'throw, 'error', 'ErrorException'
isHTTP()	'http'
hasSuspWord()	'hack', 'todo', 'to-do', 'fixme', 'bug'
isMD5()	'md5'
isSHA1()	'sha1'

Step#1-Parsing: `JSAT` uses a parser to parse the Julia programs into tokens that are used to determine the presence of security weaknesses. For parsing, `JSAT` uses a Julia package, `CSTParser.jl` [3]. Prior to parsing, the file type for each file in the repository is checked to make sure that only Julia files are parsed, and if the file exists (not empty). The `CSTParser` package uses another Julia package, `Tokenize.jl` [4], which aims to extend the built-in parser by providing additional meta information along with the resultant abstract syntax tree (AST).

Step#2-Rule Execution: `JSAT` will execute a set of rules for each security weakness category to identify security weaknesses. To generate rules for each security weakness category, we abstract coding patterns associated with each category. The rules for detecting security weaknesses in Julia programs are shown in Table 5. `JSAT` uses pattern matching to execute the rules in Table 5, similar to other static analysis tools (Rahman et al., 2019c, 2021b, 2023b; Mohammad Mehedi and Rahman, 2022). The patterns needed to execute the rules are listed in Table 6. For

[3] https://github.com/julia-vscode/CSTParser.jl
[4] https://github.com/JuliaLang/Tokenize.jl/

```
1ptrA = Base.unsafe_convert(Ptr{$elt},A)
2ptrB = Base.unsafe_convert(Ptr{$elt}, B)
3ptrC = Base.unsafe_convert(Ptr{$elt}, C)
4
5strA = size(A, 3) == 1 ? 0 : Base.stride(A, 3)
6strB = size(B, 3) == 1 ? 0 : Base.stride(B, 3)
7strC= Base.stride(C, 3)
8
9n_threads = min(Threads.nthreads(), 1 + max(length(A),
    ↪ length(B))  8000)
10# In some tests,size(20,20,20)is worth splitting
    ↪ between two threads,
11# as is size (32,32,8).
12
13if n_threads > 1
14
15    old_threads = get_num_threads()
16    set_num_threads(1)
17    Threads.@sync for ks in
        ↪ Iterators.partition(1:size(C, 3),
        ↪ cld(size(C, 3), n_threads))
18        Threads.@spawn for k in ks
19
20            ptrAk = ptrA + (k-1) * strA * sizeof($elt)
21            ptrBk = ptrB + (k-1) * strB * sizeof($elt)
22            ptrCk = ptrC +(k-1) * strC * sizeof($elt)
```

Fig. 8: An example to demonstrate our def-use chain analysis approach.

example, to execute the rule for insecure HTTP, `JSAT` uses isHTTP() from Table 6.

Step#3-Def-use chain analysis: We use def-use chain analysis (Aho et al., 1986) to identify if a security weakness is actually being used in other portions of a Julia program. Def-use chain analysis is one form of information flow analysis that is used to track data flows from an information source to a target.

A code snippet that matches the rules to detect security weaknesses may not be used anywhere in a Julia program. Therefore, `JSAT` should not generate security weakness alerts where a rule is matched but not used anywhere in the program. In order to provide `JSAT` this capability, we apply def-use chain analysis (Aho et al., 1986). `JSAT` applies def-use chain analysis for the following security weakness categories: command injection, insecure HTTP, hard-coded password, unsafe invocation, and weak encryption, as these were the security weaknesses that were observed to be associated with assignment operations during the open coding (Saldaña, 2015) process discussed in Section 2.2.1.

Figure 8 provides a code snippet from a Julia project found in our dataset, the OSS Julia project `'NNlib.jl'`[6]. We use Figure 8 to demonstrate our application of def-use chain analysis. Considering Figure 8, the variables `ptrA`, `ptrB`, and `ptrC` use the security weakness `unsafe_convert()`. The `unsafe_convert()` function is a security weakness for the unsafe invocation category, and if any or all of the inputs for `A`, `B`, or `C` contain malicious data, a unsafe invocation attack could be

[6] https://github.com/FluxML/NNlib.jl

performed. Looking further down the code snippet in Figure 8, we observe `ptrA`, `ptrB`, and `ptrC` to be respectively, used by `ptrAk`, `ptrBk`, and `ptrCk` in lines 20, 21, and 22.

Our def-chain analysis technique tracks the flow of data by constructing data dependence graphs (DDGs). In each DDG, an edge exists between a sink and a source node if the variable in the source node is used by the sink node. The source node is the code snippet that matches any of the rules listed in Table 5. The sink node can be any code snippet for which the assignment operation is applied.

```
1 P <-input by user
2 ptrA = Base.unsafe_convert(Ptr{$elt},A)<- A can contain malicious data
3
4 ptrAk = 5
5 if P:
6 ptrAk = ptrA
7 end
8 if !P:
9 run(construct_command(ptrAk))
10 end
11
12 << Do something with ptrAk >>
```

Listing 8: An example Julia program to demonstrate `JSAT`'s ability to detect security weaknesses.

In the context of Listing 8, the security weakness will be detected as a valid security weakness, which in fact is a false positive. As we track the flow of data, `JSAT` is susceptible to generate false positives for programs that use path-sensitive data flow analysis.

3.1.2 Evaluation of `JSAT`

We conduct two activities to evaluate `JSAT`: (i) evaluation with a sampled dataset and (ii) practitioner verification. We describe these activities as follows:

Evaluation activity-1: Evaluation with a sampled dataset: Security static analysis tools are subject to empirical evaluation (Rahman et al., 2019c, 2021b). We use a sampled dataset to evaluate `JSAT`'s accuracy. We apply closed coding (Saldaña, 2015) to identify which of the randomly selected files include a security weakness category. Closed coding differs from the open coding process explained in Section 2.2.1 because with the closed coding process we are looking for predetermined patterns for each of the 7 security weakness categories.

The rater who performed closed coding did not participate in the open coding process. The rater also is not any of the authors of the paper. We used this rater to mitigate any bias that can stem from while using any of the authors or the voluntary rater for closed coding process. While performing closed coding the rater followed the definitions of each security weakness category. The rater did not report any new security weakness categories, i.e., during the closed coding process

Table 7: Attributes of the Sampled Dataset

Category	Data
Data Source	Repositories provided by Murphy et al. (2020)
Total Repositories	57
Total Commits	33,053
Timespan	01/2014-10/2019
Total Julia Source Code Files	100
Total Size (Lines of Code)	193,738

Table 8: Evaluation of `JSAT` with sampled dataset for evaluation

Security Weakness Category	Occurrences	Precision	Recall
Command Injection	6	1.00	1.00
Hard-Coded Password	1	1.00	1.00
Inadequate Exception Handling	82	0.96	1.00
Insecure HTTP	4	1.00	1.00
Suspicious Comment	48	1.00	1.00
Unsafe Invocation	3	1.00	1.00
Weak Encryption	3	1.00	1.00
No security weakness	54	1.00	0.98
Average		0.99	0.99

the rater did not mention any new categories that we have not identified from our open coding analysis.

The sampled dataset was obtained by selecting a random sample of 100 Julia source code files from the dataset provided by Murphy et al. (2020). There is no overlap between the oracle dataset and sanity dataset as well as the oracle dataset and the dataset used for practitioner verification. Attributes of the sampled data is provided in Table 7.

The closed coding process took 26 hours to conduct and upon completion, we apply `JSAT` on the 100 Julia programs. We assess `JSAT`'s accuracy using two metrics: precision and recall. Precision refers to the fraction of correctly identified security weaknesses among the total identified security weakness, as determined by `JSAT`. Recall refers to the fraction of correctly identified security weaknesses that have been retrieved by `JSAT`.

We identify 142 security weaknesses among the 100 Julia programs through the closed coding process (Saldaña, 2015). The '`No security weakness`' row indicates that of the 100 Julia programs, 54 files do not include any security weakness. The rest of the 46 files include at least one security weakness. A complete breakdown of `JSAT`'s precision and recall values is provided in Table 8. For example, in the sampled dataset, we identify 82 occurrences of the inadequate exception handling. The precision and recall of `JSAT` for 82 occurrences of inadequate exception handling is respectively, 0.96 and 1.0. `JSAT` generates 3 false positives and zero false negatives for inadequate exception handling. For the sampled dataset, the average precision and recall of `JSAT` is 0.99.

Evaluation activity-2: Practitioner verification: We recruit a practitioner who is working in industry and has experience in Julia-based software development for three years. From the set of security weaknesses detected by `JSAT`, we select

Table 9: Evaluation of JSAT by practitioner

Security Weakness Category	TPs	Precision	Recall
Hard-Coded Password	1	0.50	1.00
Inadequate Exception Handling	9	0.74	1.00
Insecure HTTP	3	0.60	1.00
Suspicious Comment	54	0.67	1.00
Unsafe Invocation	130	0.77	1.00
Weak Encryption	1	1.00	0.67
Average		0.71	0.94

a random sample with 95% confidence interval. As reported in Section 13, the detected security weakness count is 23,839, for which a random sample with 95% confidence yields 379 security weaknesses. We provide these security weaknesses for labeling, where the practitioners apply closed coding to map each detected security weakness to any of the 7 categories. While applying closed coding, the practitioner is asked to use the provided guidebook that includes the definitions of the 7 security weakness categories, and their Julia-related software development experience. We provide no time restriction for the practitioner. For this activity, we provide a 25 USD Amazon gift card to the practitioner for their efforts.

The practitioner performs labeling in 50.5 hours. With the dataset labeled by the practitioner, we observe the average precision and recall to be respectively, 0.71 and 0.94. Compared to the evaluation with the sampled dataset, as summarized in Table 8, we observe the precision and recall to decrease. The 'TPs' column shows the true positive instances determined by the practitioner. According to Table 9, the 198 security weaknesses detected by JSAT are in fact present in the corresponding Julia programs and also relevant to the practitioner. The practitioner found one instance of weak encryption usage that was not identified by JSAT, which resulted in a decrease in recall, compared to that of the other categories. Results presented in Table 9 also show that majority of the detected security weaknesses are relevant, as of the 379 security weaknesses instances detected by JSAT, the practitioner found 198 of them as true positives.

3.1.3 Comparison with SEMGREP

We further compare the performance of JSAT to that with a state-of-the-art tool called SEMGREP, which provide support for scanning Julia source code files. For comparison, we use a randomly-selected sample of 500 Julia source code files that is not used in the Evaluation Dataset or during practitioner verification. We selected a set of 500 Julia source code files as it is a sample with 95% confidence interval, 5% margin of error, and 50% population proportion from the set of 4,592 source code files. For the collected sample the first author inspects the source code files as well as the alerts generated by both tools to calculate precision and recall. We do not use the Evaluation Dataset of the dataset used by practitioner verification as the rater in this regard.

Table 10 provides the findings. On average, JSAT's precision and recall is respectively, 3.7% and 35.9% higher than that of SEMGREP. Compared to that of JSAT, the recall of SEMGREP is higher for unsafe invocation. Compared to that of JSAT, the precision of SEMGREP is higher for the following categories: command injection,

Table 10: Comparison of JSAT and SEMGREP with respect to detecting security weaknesses

Category		JSAT		SEMGREP	
	Count	**Precision**	**Recall**	**Precision**	**Recall**
Command Injection	31	0.93	0.84	1.00	0.64
Hard-Coded Password	14	0.87	0.50	0.00	0.00
Inadequate Exception Handling	248	0.88	0.97	0.95	0.70
Insecure HTTP	8	0.80	1.00	0.87	0.87
Suspicious Comment	154	0.83	0.96	0.97	0.86
Unsafe Invocation	45	1.00	0.86	1.00	0.93
Weak Encryption	8	0.53	1.00	0.80	0.50
Combined	508	0.83	0.87	0.80	0.64

inadequate exception handling, insecure HTTP, suspicious comment, and weak encryption. In the case of SEMGREP, the precision and recall for detecting hard-coded passwords is 0.0.

Findings related to Evaluation of JSAT: We observe an average precision and recall of respectively, 0.99 and 0.99 for the sampled dataset. The average precision and recall drops to respectively, 0.71 and 0.94 when obtained feedback from an industry practitioner. The practitioner found 198 of the provided 379 security weaknesses to be true positives. These results show that JSAT is capable of detecting security weaknesses that are relevant to practitioners.

3.1.4 Dataset Collection

We answer RQ-2 by mining OSS repositories from GitHub. We select repositories from GitHub to assess the prevalence of the identified security weaknesses and increase generalizability of our findings, as organizations tend to host their popular OSS projects on GitHub (Rahman et al., 2019c). That being said, OSS repositories can be susceptible to quality concerns since users often host repositories on GitHub for personal purposes that do not adequately reflect professional software development (Munaiah et al., 2017). As advocated by prior research (Munaiah et al., 2017), OSS repositories need to be curated. We apply the following criteria to curate the collected repositories:

- *Criterion-1:* At least 1% of the files in the repository must be Julia programs to collect repositories that contain sufficient amount of Julia programs for analysis (Murphy et al., 2020).
- *Criterion-2:* The repository is not a copy of another repository.
- *Criterion-3:* The repository has at least five contributors. Prior research (Humbatova et al., 2020) has also used the threshold of at least five contributors, with the assumption that it helps to filter out repositories used for personal purposes.
- *Criterion-4:*. The repository must have at least two commits per month. Munaiah et al. (2017) used the threshold of at least two commits per month to determine which repositories have enough development activity.

Table 11: Filtering of OSS Julia Repositories

	GitHub
Initial Repository Count	11,981
Criterion-1 (1% Julia files)	11,981
Criterion-2 (Not a copy)	11,968
Criterion-3 (Contributors >=5)	950
Criterion-4 (Commits/Month >=2)	558
Final Count	558

Table 12: Attributes of Dataset

Category	**Data**
Total Repositories	558
Total Commits	378,239
Total Developers	14,295
Time Span	01/2014 - 01/2023
Total Julia Source Code Files	25,008
Total Size (Lines of Code)	4,668,055
Total Count of Issue Reports	185,290
Total Count of Stars	117,673

As shown in Table 11, we answer RQ-2 using 25,008 programs collected from 558 OSS Julia project repositories downloaded from GitHub. We clone the master branches of the 558 repositories. We provide attributes of the mined 558 repositories in Table 12. In all we collect 25,008 Julia source code files.

3.1.5 Sanity Check

Our evaluation of `JSAT` described in Section 3.1.1 is limited to the sampled dataset for evaluation. We account for this limitation by creating another dataset called the 'sanity dataset' that does not include any of the 126 OSS repositories used in forming the sampled dataset or the open coding process described in Section 3.1. With respect to accuracy, `JSAT` may have high accuracy on the sampled dataset, but not on the complete dataset. We perform a sanity check using 430 Julia programs randomly-selected from 10 OSS Julia projects downloaded from GitHub that are part of the final dataset described in Section 3.1.4. We manually inspect each Julia program applying closed coding (Saldaña, 2015) to provide a mapping between each file and a security weakness category. Next, we run `JSAT` on the sanity dataset. Finally, we report the precision and recall of `JSAT` for the 430 randomly-selected files.

We locate 188 security weaknesses through the closed coding process. The sanity dataset respectively, includes 18, 58, 78, 31, and 3 instances of unsafe invocation, command injection, inadequate exception handling, suspicious comment, and weak encryption. The precision for unsafe invocation, command injection, inadequate exception handling, suspicious comment, and weak encryption is respectively, 0.94, 0.98, 0.98, 1.00, and 1.00. The recall for unsafe invocation, command injection, inadequate exception handling, suspicious comment, and weak encryption is respectively, 1.00, 1.00, 1.00, 1.00, and 1.00. The average precision and recall for the security weakness categories regarding the sanity dataset and sampled dataset is

>= 0.98, which gives us the confidence of detecting all existing security weaknesses in our datasets while generating false positives.

3.1.6 Quantifying Security Weakness Frequency

We answer RQ-2 by reporting:

- Count of security weaknesses;
- Weakness per repo using Equation 1;
- Weakness proportion using Equation 2; and
- Weakness density with Equation 3.

The three metrics characterize the frequency of security smells differently. The smell density metric is more granular, and focuses on the content of a Julia source code file as measured by how many smells occur for every 1,000 LOC. The weakness proportion metric is less granular and focuses on the existence of at least one of the seven security weakness categories. Finally, weakness per repo showcases on average how many repositories with Julia source code files include security weaknesses.

$$\text{Weakness per Repo } (i) = \frac{\text{\# of repositories with >= 1 security weaknesses}}{\text{Total Julia repos in the dataset}} * 100\% \tag{1}$$

$$\text{Weakness Proportion } (i) = \frac{\text{\# of Julia files with >= 1 security weaknesses}}{\text{Total Julia files in the dataset}} * 100\% \tag{2}$$

$$\text{Weakness Density } (i) = \frac{\text{Total \# of security weaknesses}}{\text{Total lines of code of Julia files / 1,000}} \tag{3}$$

3.2 Answer to RQ-2

In this section, we answer *RQ-2: How frequently do the identified security weaknesses appear for Julia programs?* Altogether, `JSAT` identifies 23,839 security weaknesses among the 25,008 Julia programs contained in the 558 OSS Julia programs making up our dataset. The most frequent category is inadequate exception handling. A complete breakdown of the findings is provided in Table 13.

Of the total 23,839 security weakness 8.18%, 5.00%, 0.11%, 54.7%, 2.13%, 29.24%, and 0.64% are respectively, instances of unsafe invocation, command injection, hard-coded password, inadequate exception handling, insecure HTTP, suspicious comment, and weak encryption. For the 13,050 instances of inadequate exception handling, 5.3%, 24.9%, and 69.8% respectively, are exposing stack traces, catching generic exceptions, and throwing generic exceptions.

Table 13: Answer to RQ-2: Frequency of Security Weaknesses In Julia Source Code Files

Category	Occurrences	Weakness Density	Weakness Proportion	Weakness per Repo
Command Injection	1,193	0.3	1.51%	31.0%
Hard-Coded Password	27	0.01	0.06%	1.25%
Inadequate Exception Handling	13,050	3.25	13.78%	86.0%
Insecure HTTP	508	0.13	0.59%	12.93%
Suspicious Comment	6,956	1.73	13.33%	78.31%
Unsafe Invocation	1,951	0.49	2.07%	23.11%
Weak Encryption	154	0.04	0.30%	6.81%
Combined	23,839	5.93	24.9%	95.34%

`JSAT` is run on the 25,008 Julia programs contained in the 558 OSS Julia repositories. We identify 25,016 Julia source code files and 1,978 repositories for Julia source code files. A complete breakdown of findings related to RQ-2 is presented in Table 13 with Occurrences, Weakness Density, Weakness proportion, and Weakness per Repo.

Occurrences: The occurrences of 7 categories are presented in the 'Occurrences' column of Table 13. The 'Combined' row presents the total security weaknesses. We identify 23,839 security weakness mapping to 7 categories in 25,008 Julia source code files. We observe inadequate exception handling is the most prevalent security weakness category in Julia source code files, with 13,050 occurrences, followed by suspicious comment with a 6,956 occurrences.

Weakness Density: In the 'Weakness Density' column of Table 13 we report weakness density. The 'Combined' row presents the weakness density for total Julia source code files when all 7 security weakness categories are considered. The most frequent category is inadequate exception handling, for which we observe a density of 3.25 per KLOC. Considering all 7 categories, the density is 5.93 per KLOC.

Weakness Proportion: In the 'Weakness Proportion' column of Table 13 we report weakness proportion. The 'Combined' row presents the proportion of Julia source code files in which at least one of the 7 categories appear. We observe 13.78% and 13.33% of Julia source code files include inadequate exception handling and suspicious comment. The proportion of hard-coded password, insecure HTTP, and weak encryption appear in Julia source code files is less than 1%. Altogether, 24.9% of Julia source code files include at least one security weakness.

Weakness per Repo: In the 'Weakness per Repo' column of Table 13 we report weakness per repo. The 'Combined' row presents the proportion of repositories with Julia source code files include at least one security weakness. We observe the proportion of inadequate exception handling and suspicious comment occur in repositories for Julia source code files exceeds 30.0%. As shown in the 'Combined' row, 95.34% of 558 repositories with Julia source code files include at least one security weakness.

We further compute how many detected security weaknesses were with and without def-use chain analysis. In Table 14, the 'Reduction (%)' column reports the percentage of security weaknesses that we reduced by applying def-use chain anal-

Table 14: Reduction in Detected Security Weaknesses with Def-use Chain Analysis

Category	**With Def-use**	**Without Def-use**	**Reduction (%)**
Command Injection	1,193	1,314	9.2
Hard-Coded Password	27	284	90.5
Insecure HTTP	508	511	0.6
Unsafe Invocation	1,951	2,292	14.8
Weak Encryption	154	154	0.0
Total	3,833	4,555	15.8

ysis. We applied def-use chain analysis for five categories. Use of def-use chain analysis resulted in a total reduction of 15.8% security weaknesses as detected by `JSAT`. These results demonstrate that `JSAT`'s use of def-use chain analysis can help in better detection of certain security weakness categories in Julia programs.

Answer to RQ-2: Altogether, we identify 23,839 security weaknesses in 25,008 Julia source code files. On average, we observe 24.9% of Julia source code files to include at least one security weakness.

4 Discussion

We discuss the implications for both practitioners and researchers who work with Julia programs in the following sections.

4.1 Implications for Practitioners

We discuss the implications of our empirical study for practitioners in the following subsections:

4.1.1 Mitigation-related Recommendations

We advise practitioners to apply the following measures to mitigate detected security weaknesses:

– Command Injection: Practitioners can apply static analysis tools, such as `JSAT` to detect security weaknesses related to command injection. When performing input validation, consider all potentially relevant properties, including length, type of input, range of acceptable values, missing inputs, extra inputs, and syntax. Only accept inputs that strictly conform to specifications and reject any input that does not strictly conform to specifications (MITRE, 2021g).

– Hard-Coded Password: Practitioners can apply static analysis tools, such as `JSAT` and Snyk [7], to detect hard-coded passwords. We also recommend practitioners to use secret management tools, such as Hashicorp Vault [8] to store and retrieve passwords.

[7] https://snyk.io/

[8] https://www.vaultproject.io/

– Inadequate Exception Handling:. We advocate for adequate exception handling by not exposing stack traces and throwing/catching specific exceptions. We recommend practitioners to avoid using broad or generic error handling.

– Insecure HTTP:. We advocate for better tool support so that programmers do not abandon the process of setting up HTTP with SSL/TLS. Practitioners can periodically check existence of secure HTTP endpoints, as a URL that uses HTTP now can later be changed to HTTP with SSL/TLS.

– Suspicious Comments: Remove any comments that suggest the presence of bugs or incomplete functionality before deploying a project (MITRE, 2021f).

– Unsafe Invocation: Practitioners can apply static analysis tools, such as `JSAT`, to detect security weaknesses relating to unsafe invocation attacks. When performing input validation, consider all potentially relevant properties, including length, type of input, range of acceptable values, missing inputs, extra inputs, and syntax (MITRE, 2021i). Allowlists and denylists can be useful for detecting potential attacks. An 'allowlist' is a list of assets, such as hosts, email addresses, or applications that are authorized to be present or active on a system according to a well defined baseline, and a 'denylist' is a list of assets that are known to be associated with malicious activity (Sedgewick et al., 2015; NIST, 2021a).

– Weak Encryption: Practitioners can use cryptography algorithms recommended by the National Institute of Standards and Technology (Baker, 2020). CWE (MITRE, 2021c,d) recommends that sensitive data *"should be encrypted with keys that are at least 128 bits in length for adequate security"*.

4.1.2 Prioritization-related Recommendations

Results reported in Table 2 have implications related to prioritization. Four of the seven identified security weakness categories have a high likelihood of being exploited and therefore, those security weakness categories can be prioritized first for mitigation. More concretely, `JSAT` can be further extended to not only report the location of the security weakness, but also report that the detected security weakness has a 'high', 'medium', or 'low' likelihood of being exploited. Furthermore, when applying code review, practitioners can prioritize for security weakness categories using the results from Table 2. Because of our derived taxonomy, practitioners do not need to hire a security expert to manually inspect Julia programs.

4.1.3 Propagation-related Implications

We observe from Table 13 that security weaknesses exist in OSS Julia programs hosted on GitHub. Our findings demonstrate the prevalence of security weaknesses in Julia programs. The prevalence of security weaknesses in OSS Julia programs could potentially trigger the propagation of security weaknesses in Julia project development, as unaware Julia practitioners may consider security weakness usage as an acceptable practice. Based on our findings and the afore-mentioned discussion, we advocate for detection and mitigation of security weaknesses in Julia programs. Our security static analysis tool `JSAT` can be helpful to locate where the 7 security weakness categories appear in Julia programs. Practitioners can use data presented in Tables 13 as a security weakness benchmark for Julia programs.

4.1.4 Usage of `JSAT` in Julia-based Software Development

`JSAT` can be used in two ways as discussed as follows:

- `JSAT` can be used to directly to scan any Julia program upon completion of writing code. The scan can be conducted manually by executing the tool or through a 'git hook' [9].
- `JSAT` can be used as a plugin in the IDE by creating a wrapper around the two three components namely, 'parser', 'rule matcher', and 'analyzer'. The detection of security weaknesses is dependent on `JSAT`'s rule matching ability and data flow analysis. `JSAT` relies on a specific set of patterns that prohibits generation of a lot of alarms. Therefore, when applied in practice `JSAT` is expected to not generate a lot of alarms.

4.2 Future Work

We provide the following directions as future work:

- Empirical Studies: This paper provides the ground-work to conduct empirical studies that will investigate the reasons that attribute to security weaknesses in Julia programs. Future research can investigate to what extent these reasons are applicable for Julia programs. Researchers can survey practitioners developing Julia programs to assess practitioner perspective of the identified 7 security weakness categories. More sophisticated information flow analysis can be developed and performed based on the work done for this paper. Furthermore, researchers can investigate if security weaknesses mentioned in code snippets from online forums, like Stack Overflow or the devoted Julia Discourse forum (Julia, 2021a), propagate into OSS Julia project development.
- A Generalizable Framework: Julia is still an emerging programming language, and more language features are expected to be introduced as time progresses. To meet the needs of this evolving programming language, researchers and toolsmiths can build on top of `JSAT` to propose a framework that can detect all categories of security weaknesses that can occur in Julia programs. One approach of constructing a framework of this nature is to rely on CWE entries, and perform a holistic analysis of Julia programs that are publicly available on GitHub.
- Repairs of security weaknesses: Currently, `JSAT` does not provide any recommendations on how to repair detected security weaknesses. As future work, researchers can explore if template-based techniques (Gazzola et al., 2019) or large language models, such as ChatGPT (OpenAI, 2022) are capable of repairing security weaknesses in Julia programs.

5 Threats to Validity

We discuss the limitations of this paper as following:

[9] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

– Conclusion Validity: Our derivation of security weakness categories and the rules to detect each category are limited to the files in the dataset that we used in Section 2.2.2. We mitigate these limitations by applying qualitative analysis with 4,592 Julia programs used in 126 open-source Julia programs (Murphy et al., 2020) to identify security weakness categories. The derived security weakness categories are susceptible to rater bias for both the open coding and closed coding processes (Saldaña, 2015). We mitigate this limitation by performing rater verification. We acknowledge that the derived rules may not capture all possible variants of a security weakness category as these rules are derived from open coding. As described in Section 2.2.2, we use 4,592 Julia programs mined from 126 repositories. Our rules are limited to this set of Julia programs.

  We acknowledge that `JSAT` may generate false positives when applied to other datasets. We mitigate this limitation by evaluating `JSAT` using both a sampled dataset and a sanity dataset, reported in Section 3.1, Section 3.1.2, and Section 13, and Section 3.1.5 respectively. We further mitigate the possibility of false positives by employing def-use chain analysis as a part of `JSAT`'s functionality to reduce the number of false positives for the security weakness categories: command injection, hard-coded password, unsafe invocation, and weak encryption, reported in Section 3.1.1. As we track the flow of data, `JSAT` is susceptible to generate false positives for programs that use path-sensitive data flow analysis even after applying def-use chain analysis.

  The datasets used for evaluating `JSAT` are limited to the repositories that we have mined and may not be reflective of other data sources that are hosted on proprietary and GitLab code hosting platforms. Furthermore, we have used one practitioner to seek practitioner feedback. As such the reported detection accuracy for `JSAT` is subject to vary when used by other practitioners.

  `JSAT` applies data flow analysis to mitigate false positives but does not apply control flow analysis. As a result, if a security weakness is mitigated by control flow, e.g., within an if-else block, then `JSAT` will report false positive instances. This limitation can impact the reported detection accuracy for `JSAT`.

– External Validity The datasets used for this paper are all constructed by mining OSS repositories from GitHub. Our findings may not generalize for proprietary datasets. Our findings may also not generalize for other OSS repositories that are not included in our dataset.

– Internal Validity: While constructing the sampled dataset for evaluation and sanity check datasets for evaluating `JSAT`, we acknowledge that we may have unconscious expectations surrounding the outcomes that could potentially impact the closed coding process (Saldaña, 2015). These implicit expectations is applicable for both: the initial and additional rounds of qualitative analysis.

  We have applied a multi-phase and multi-rater approach for deriving our taxonomy of security weaknesses. While this approach mitigates rater bias, we acknowledge that this approach can be a source of rater inconsistency.

6 Related Work

We discuss relevant related work in this section.

6.1 Prior Research Related to Julia

Our paper is related to prior research that has conducted research related to Julia-related software development. Rahman et al. (2023a) derived a defect taxonomy for Julia programs. Innes et al. (2019) constructed and evaluated 'Zygote', a tool to solve differential equations. Churavy (Churavy, 2019) constructed and evaluated a debugging tool called 'Cthulhu' that uses static and dynamic analysis. Comparison of Julia program performance to that of other programming languages has also been investigated. Gibson (2017) argued that Julia has multiple benefits over general purpose programming languages with respect to graphic rendering capabilities, user experience, and program execution time. Tomasi et al. (2018) documented Julia programs to have comparable performance compared to C++ programs. They (Tomasi and Giordano, 2018) also advocated for Julia's use in astrophysics research as Julia provides JIT compilation, produces optimized machine code with LLVM, and provides support for missing data values. Januszek et al. (2018) compared the performance of five programming languages with $\mathcal{O}(n^3)$ algorithms and observed superior computational efficiency for Julia programs compared to that of Wolfram, R, Python, and C# programs. For parallel programming, Gmys et al. (2020) found Julia to be better than Python with respect to performance, and better than C programs with respect to productivity. Sells (2020) used an open-source, industry-standard, missile and rocket simulation software called 'Mini-Rocket' to assess and benchmark productivity metrics of Julia against Python, Java, and C++. Their (Sells, 2020) results showed that Julia required far less lines of code than the other three languages and was second best only to Python in terms of 'ease-of-coding' productivity. Axillus (2020) compared the execution times of Julia and Python to perform machine learning tasks and reported that Julia was 1.25∼1.5 times faster than Python during the GPU-accelerated DNN experiments and also outperformed Python in 5 out of 8 k-nearest neighbor experiments. Dogaru et al. (2015) observed that Julia's base JIT implementation was 157.5 times faster than raw Python code and 3.09 times faster than JIT-assisted Python code.

6.2 Prior Research Related to Security Weaknesses

Our paper is also related to prior research that has investigated security weaknesses for other languages, such as C++, Python, and languages used for configuration scripts. We briefly describe relevant prior research as follows:

– **Security Weaknesses in C++ Programs:** In separate publications, Verdi et al. (2022) and Zhang et al. (2022) conducted empirical studies of security weaknesses in C++ programs that are available on Stack Overflow. Here they identified security weaknesses that map to CWE entries by using a tool called CPPCheck. Zhang et al. (2022) reported the most frequently occurring security weakness to be CWE-908, i.e., the use of an uninitialized resource. Unlike Verdi

et al. 2022, they identified security weaknesses that map to CWE entries by using a tool called SourcererCC. Verdi et al. (2022) reported the most frequently occurring category to be CWE-1006, i.e., bad coding practices. In another paper, Alnaeli et al. (2016) used to a tool called UnsafeDetector to investigate security weaknesses in 15 software systems developed in C and C++, and observed 'strcmp' to be used most frequently.

- **Security Weaknesses in Configuration Scripts:** Configuration scripts are used to automatically manage computing infrastructure with dedicated programming languages, such as Ansible, Chef, and Puppet. In separate publications, Rahman et al. derived taxonomies for Ansible (Rahman et al., 2021a), Chef (Rahman et al., 2021a), and Puppet (Rahman et al., 2019b). Across all these studies (Rahman et al., 2021a, 2019b), Rahman et al. found hard-coded secrets to be the most frequently occurring category (Rahman and Williams, 2021). In all the of the afore-mentioned publications security weakness categories are derived through open coding. Rahman et al.'s security weakness detection tools were improved by Ferreira et al. (2023) and Opendebeeck et al. (2022) for Ansible as well as by Reis et al. (2023) for Puppet.
- **Security Weaknesses in Container Orchestration:** In recent work, Rahman et al. (2023b) derived a taxonomy for configuration files used for container orchestration. In particular, Rahman et al. (2023b) built on existing work (Shamim et al., 2020) to derive security misconfigurations for Kubernetes manifests. Here, the authors detected security misconfigurations by deriving a taxonomy of security misconfigurations.
- **Security Weaknesses in Python Programs:** Rahman et al. (2019d) mined 5,822 Python Gists and found command injection to be the most frequently occurring security weakness category. Jukka et al. (2021) mined PyPI packages and reported inadequate exception handling to be the most frequently occurring category. Bhuiyan et al. (2022) studied Python-based machine learning projects, and found use of potentially dangerous functions to be the most frequently occurring category. Rahman et al. (2019a) mined 44,966 Stack Overflow answers and found code injection to be the most frequently occurring category. All of these publications use a security static analysis tool called 'Bandit' to identify security weaknesses that map to CWE entries.

6.3 Prior Research Related to Deep Learning for Classification Tasks

Zheng et al. (2021) used deep convolutional network to identify unauthorized broadcasting in radio communications. In separate publications, Zheng et al. used deep learning to automatically classify automated modulation (2022; 2023) and constellation images (2024) used in wireless networks. Chakraborty et al. (2022), Fu et al. (2022), and Hin et al. (2022) respectively, used deep learning, stacking transformers, and graph neural networks to detect vulnerabilities in source code.

Summary: The above-mentioned discussion showcases the prevalence of empirical research in security weakness detection for wide-range of general-purpose programming languages as well as domain-specific languages. None of these research addresses the secure development aspects of Julia-based software. Furthermore,

the research studies that have investigated Julia focus on performance and usability but do not address the need of deriving a taxonomy for security weaknesses and developing a tool to detect them. In our paper, we have addressed this research gap by:

1. Using qualitative analysis to derive a taxonomy of security weaknesses;
2. Using def-use chain analysis and rule matching to detect security weaknesses in Julia source code files; and
3. Conducting an empirical study of security weaknesses in OSS Julia source code files.

7 Conclusion

Julia is an emerging programming language used both in scientific computing and product development. Julia is designed to have similar syntax to a scripting language and similar program execution speed with low-level memory access to a compiled language, such as C. Despite reported benefits, Julia programs can include security weaknesses. Security weaknesses found in Julia programs can provide malicious users means to conduct attacks that can lead to serious consequences. We conduct an empirical study applying open coding to identify 7 security weakness categories: command injection, hard-coded password, inadequate exception handling, insecure HTTP, suspicious comments, unsafe invocation, and weak encryption. We construct and evaluate a static analysis tool called `JSAT` to automatically identify security weaknesses in Julia programs. Using `JSAT`, we analyze 558 OSS Julia project repositories, consisting of 25,008 Julia programs, and identify 23,839 security weaknesses.

Based on our empirical study, we advocate (i) practitioners to apply rigorous inspection efforts as part of their Julia program development process, and (ii) researchers to focus further on development and application of security static analysis tools, such as `JSAT`, so that security weaknesses in Julia programs can be detected before they are being deployed for production. We hope this paper will advance the domain of security research for Julia programs as it continues to emerge as a popular programming language.

Future Directions for Exploration Our paper provides groundwork for future directions to explore in the following areas:

- Enhanced analysis techniques: In our paper, we have applied data flow analysis within a Julia files. Researchers can further explore if other artifacts, such as Julia binaries can be used for applying data flow analysis for identifying more weaknesses efficiently. Development of such techniques can also lead to new research in other domains, such as Julia-related performance analysis and Julia-related defect detection.
- Inclusion of more weakness categories: Our taxonomy includes 7 weaknesses, which may not be comprehensive. We invite researchers to apply our methodology on more datasets to identify more weakness categories not reported in our paper.

- Replication studies for further substantiation: Similar to prior research in IaC, we invite researchers to conduct replication of our research in exact as well as different settings. These replications can challenge or substantiate our findings in this novel domain of Julia-related research.
- Industrial experience reports: While `JSAT` shows promise with respect to detection of security weaknesses, the tool needs to be deployed in an industrial setting where a team of practitioners will use the tools and provide their experience in using the tool.
- Systematic comparison: As time progresses, we forecast that there will be more tools that can identify security weaknesses in Julia programs. Once the tool landscape is more mature, researchers can conduct systematic evaluation of these Julia-related tools, similar to other domains, such as smart contracts.

Data Availability Statement: Dataset and source code used in our paper is publicly available online (Rahman et al., 2023c).

Acknowledgements We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2310179, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175. We thank Farzana Ahamed Bhuiyan for her help in creating the sampled dataset. We also thank Anceito Rivera for his help in conducting rater verification. This work has benefited from Dagstuhl Seminar 23181 "Empirical Evaluation of Secure Development Processes."

Declarations - Funding and/or Conflicts of interests/Competing interests

Funding and/or Conflicts of interests/Competing interests We, the authors have no conflict of interest. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175.

References

Julia joins petaflop club. `https://www.hpcwire.com/off-the-wire/julia-joins-petaflop-club/`, 2017.

Julia: come for the syntax, stay for the speed. `https://www.nature.com/articles/d41586-019-02310-3`, 2019.

The Julia language. `https://docs.julialang.org/en/v1/`, 2022.

A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

S. M. Alnaeli, M. Sarnowski, M. S. Aman, K. Yelamarthi, A. Abdelgawad, and H. Jiang. On the evolution of mobile computing software systems and c/c++ vulnerable code: Empirical investigation. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 1–7. IEEE, 2016.

aviatesk. aviatesk/jet.jl, 2023. URL https://juliapackages.com/p/jet.

V. Axillus. Comparing Julia and Python: An investigation of the performance on image processing with deep neural networks and classification, 2020. URL https://www.diva-portal.org/smash/record.jsf?pid=diva2\%3A1389123\&dswid=5389.

E. Baker. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. 03 2020. doi: https://doi.org/10.6028/NIST.SP.800-175Br1.

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. URL https://epubs.siam.org/doi/10.1137/141000671.

J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky. Julia: Dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2:1–23, 2018a. URL https://dl.acm.org/doi/abs/10.1145/3276490.

J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky. Julia: Dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018b. doi: 10.1145/3276490. URL https://doi.org/10.1145/3276490.

F. A. Bhuiyan, S. Prowell, H. Shahriar, F. Wu, and A. Rahman. Shifting left for machine learning: An empirical study of security weaknesses in supervised learning-based projects. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 798–808, 2022. doi: 10.1109/COMPSAC54236.2022.00130.

D. Boxler. Static taint analysis tools to detect information flows. 2018.

S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2022. doi: 10.1109/TSE.2021.3087402.

V. V. R. Churavy. *Transparent distributed programming in Julia*. PhD thesis, Massachusetts Institute of Technology, 2019. URL https://dspace.mit.edu/handle/1721.1/122755.

J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi: 10.1177/001316446002000104. URL http://dx.doi.org/10.1177/001316446002000104.

J. Computing. Julia computing celebrates 10 years with retrospective, 2022. URL https://www.hpcwire.com/off-the-wire/julia-computing-celebrates-10-years-with-retrospective/.

cvedetails. Vulnerability details : Cve-2021-4048. https://www.cvedetails.com/cve/CVE-2021-4048/, 2021. [Online; accessed 19-June-2024].

I. Dogaru and R. Dogaru. Using Python and Julia for efficient implementation of natural computing and complexity related algorithms. In *2015 20th International Conference on Control Systems and Computer Science*, pages 599–

604. IEEE, 2015. URL https://ieeexplore.ieee.org/abstract/document/7168488.

E. Farhana, N. Imtiaz, and A. Rahman. Synthesizing program execution time discrepancies in julia used for scientific software. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 496–500, 2019.

M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019. doi: 10.1109/TSE.2017.2755013.

J. Gibson. The julia programming language: the future of scientific computing. *APS*, pages L39–011, 2017. URL https://ui.adsabs.harvard.edu/abs/2017APS..DFDL39011G/abstract.

J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, and D. Tuyttens. A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation*, page 100720, 2020. URL https://www.sciencedirect.com/science/article/abs/pii/S22106.

M. A. Heroux, J. M. Willenbring, and M. N. Phenow. Improving the development process for cse software. In *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, pages 11–17, 2007. doi: 10.1109/PDP.2007.51.

E. Heymann, B. P. Miller, A. Adams, K. Avila, M. Krenz, J. R. Lee, and S. Peisert. Guide to securing scientific software, June 2023. URL https://doi.org/10.5281/zenodo.8137009. This document is a product of Trusted CI. Trusted CI is supported by the National Science Foundation under Grant #1920430. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

G. Hickey and C. Kipping. A multi-stage approach to the coding of data from open-ended questions. *Nurse researcher*, 4(1):81–91, 1996.

D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1110–1121, 2020. doi: 10.1145/3377811.3380395.

M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *CoRR, abs/1907.07587*, 2019. URL

https://deepai.org/publication/a-differentiable-programming-system-to-bridge-machine-learning-and-scientific-computing.

T. Januszek and M. Pleszczyński. Comparative analysis of the efficiency of Julia language against the other classic programming languages. *Silesian Journal of Pure and Applied Mathematics*, 8, 2018. URL https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-c4339453-4519-4b92-a673-307638a50cb1.

JLHUB. Julia manual - function list and reference. https://www.jlhub.com/julia/manual/en/, 2024. [Online; accessed 19-July-2024].

Julia. Parallel supercomputing for astronomy, 2017. URL https://juliacomputing.com/case-studies/celeste.html.

Julia. Programming languages: Developers reveal what they love and loathe, and what pays best, 2020. URL https://www.zdnet.com/article/programming-languages-developers-reveal-what-they-love-and-loathe-and-what-pays-best/.

Julia. Discourse, 2021a. URL https://discourse.julialang.org/.

Julia. The julia programming language, 2021b. URL https://docs.julialang.org/en/v1/base/c/\#Base.unsafe_convert.

Julia, 2021c. URL https://docs.julialang.org/en/v1/base/c/\#Base.unsafe_convert.

julia. Julia Documentation, 2024. URL https://web.mit.edu/julia_v0.6.2/julia/share/doc/julia/html/en/index.html.

julia-vscode. julia-vscode/staticlint.jl, 2023. URL https://github.com/julia-vscode/StaticLint.jl/tree/master.

D. Kelly, R. Sanders, et al. Assessing the quality of scientific software. In *First International Workshop on Software Engineering for Computational Science and Engineering*. Citeseer, 2008.

D. Kelly, S. Smith, and N. Meng. Software engineering for scientists. *Computing in Science & Engineering*, 13(05):7–11, 2011.

J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977. ISSN 0006341X, 15410420. URL http://www.jstor.org/stable/2529310.

R. Milewicz, J. Carver, S. Grayson, and T. Atkison. A secure future for open-source computational science and engineering. *Computing in Science and Engineering*, 24(4):65–69, 2022. doi: 10.1109/MCSE.2022.3221877.

MITRE. Cwe-311: Missing encryption of sensitive data, 2021a. URL https://cwe.mitre.org/data/definitions/311.html.

MITRE. Cwe-319: Cleartext transmission of sensitive information, 2021b. URL https://cwe.mitre.org/data/definitions/319.html.

MITRE. Cwe-321: Use of hard-coded cryptographic key, 2021c. URL https://cwe.mitre.org/data/definitions/321.html.

MITRE. Cwe-327: Use of a broken or risky cryptographic algorithm, 2021d. URL https://cwe.mitre.org/data/definitions/327.html.

MITRE. Cwe-396: Declaration of catch for generic exception, 2021e. URL https://cwe.mitre.org/data/definitions/396.html.

MITRE. Cwe-546: Suspicious comment, 2021f. URL https://cwe.mitre.org/data/definitions/546.html.

MITRE. Cwe-78: Improper neutralization of special elements used in an os command ('os command injection'), 2021g. URL https://cwe.mitre.org/data/definitions/78.html.

MITRE. Cwe-798: Use of hard-coded credentials, 2021h. URL https://cwe.mitre.org/data/definitions/798.html.

MITRE. Cwe-94: Improper control of generation of code ('code injection'), 2021i. URL https://cwe.mitre.org/data/definitions/94.html.

MITRE. Common weakness enumeration, 2021j. URL https://cwe.mitre.org/.

MITRE. Cwe-754: Improper check for unusual or exceptional conditions, 2023. URL https://cwe.mitre.org/data/definitions/754.html.

H. Mohammad Mehedi and A. Rahman. As code testing: Characterizing test quality in open source ansible development. In *2022 15th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Los Alamitos, CA, USA, apr 2022. IEEE Computer Society. URL https://akondrahman.github.io/publication/icst2022.

C. Morris. Some lessons learned reviewing scientific code. In *Proc 30th Intl Conference Software Eng (iCSE08)*, 2008.

N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, pages 1–35, 2017. ISSN 1573-7616. doi: 10.1007/s10664-017-9512-6. URL http://dx.doi.org/10.1007/s10664-017-9512-6.

J. Murphy, E. T. Brady, S. I. Shamim, and A. Rahman. A curated dataset of security defects in scientific software projects. In *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, HotSoS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375610. doi: 10.1145/3384217.3384218. URL https://doi.org/10.1145/3384217.3384218.

NIST. Special publication 800-63c conformance criteria, 2021a. URL https://www.nist.gov/system/files/documents/2021/04/27/800-63C\%20Conformance\%20Criteria_042621.pdf.

NIST. Source code security analyzers, 2021b. URL https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers.

R. Opdebeeck, A. Zerouali, and C. De Roover. Smelly variables in ansible infrastructure code: Detection, prevalence, and lifetime. In *2022 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2022.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/, 2022. [Online; accessed 12-July-2023].

OWASP. Command injection, 2021a. URL https://owasp.org/www-community/attacks/Command_Injection.

OWASP. Testing for weak encryption, 2021b. URL https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/04-Testing_for_Weak_Encryption.

OWASP. Source code analysis tools, 2021c. URL https://owasp.org/www-community/Source_Code_Analysis_Tools.

J. M. Perkel. Julia: come for the syntax, stay for the speed. 2019.

A. Rahman and L. Williams. Different kind of smells: Security smells in infrastructure as code scripts. *IEEE Security Privacy*, 19(3):33–41, 2021. doi: 10.1109/MSEC.2021.3065190.

A. Rahman, E. Farhana, and N. Imtiaz. Snakes in paradise?: Insecure python-related coding practices in stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 200–204, 2019a. doi: 10.1109/MSR.2019.00040.

A. Rahman, C. Parnin, and L. Williams. The seven sins: security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019b.

A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175, 2019c. doi: 10.1109/ICSE.2019.00033.

A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), Jan. 2021a. ISSN 1049-331X. doi: 10.1145/3408897. URL https://doi.org/10.1145/3408897.

A. Rahman, R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology*, 30:1–31, 01 2021b. doi: 10.1145/3408897.

A. Rahman, D. B. Bose, R. Shakya, and R. Pandita. Come for syntax, stay for speed, understand defects: An empirical study of defects in julia programs. *Empirical Software Engineering*, 28(93):33, 2023a.

A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023b. ISSN 1049-331X. doi: 10.1145/3579639. URL https://doi.org/10.1145/3579639.

A. Rahman, Y. Zhang, and J. Murphy. Verifiability package for paper. https://figshare.com/s/0e2c77afd8215cbd3be2, 2023c. [Online; accessed 15-Oct-2023].

M. R. Rahman, A. Rahman, and L. Williams. Share, but be aware: Security smells in python gists. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 536–540, 2019d. doi: 10.1109/ICSME.2019.00087.

S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato. Leveraging practitioners' feedback to improve a security linter. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3560419. URL https://doi.org/10.1145/3551349.3560419.

J. Ruohonen, K. Hjerppe, and K. Rindell. A large-scale security-oriented static analysis of python packages in pypi. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–10, 2021. doi: 10.1109/PST52912.2021.9647791.

N. Saavedra and J. a. F. Ferreira. Glitch: Automated polyglot security smell detection in infrastructure as code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556945. URL https://doi.org/10.1145/3551349.3556945.

J. Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

A. Sedgewick, M. Souppaya, and K. Scarfone. *Guide toApplicationWhitelisting*. 10 2015. doi: http://dx.doi.org/10.6028/NIST.SP.800-167.

R. Sells. Julia programming language benchmark using a flight simulation. In *2020 IEEE Aerospace Conference*, pages 1–8, 2020. doi: 10.1109/AERO47225.2020.9172277.

M. I. Shamim, F. A. Bhuiyan, and A. Rahman. Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. In *2020 IEEE Secure Development (SecDev)*, pages 58–64, Los Alamitos, CA, USA, sep 2020. IEEE Computer Society. doi: 10.1109/SecDev45635.2020.00025.

M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 251–260, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368123. URL https://doi.org/10.1145/1368088.1368123.

A. Sweeney, K. E. Greenwood, S. Williams, T. Wykes, and D. S. Rose. Hearing the voices of service user researchers in collaborative qualitative data analysis: the case for multiple coding. *Health Expectations*, 16(4):e89–e99, 2013.

L. Tan, D. Yuan, and Y. Zhou. Hotcomments: How to make program comments more useful? 01 2007.

M. Tomasi and M. Giordano. Towards new solutions for scientific computing: the case of Julia. *arXiv preprint arXiv:1812.01219*, 2018. URL https://arxiv.org/abs/1812.01219.

M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering*, 48(5):1497–1514, 2022. doi: 10.1109/TSE.2020.3023664.

B. Wallace. Compromising an entire julia cluster, 2016. URL `https://blogs.blackberry.com/en/2016/05/compromising-an-entire-julia-cluster`.

F. Zappa Nardelli, J. Belyakova, A. Pelenitsyn, B. Chung, J. Bezanson, and J. Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2:1–27, 2018.

H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering*, 48 (7):2359–2375, 2022. doi: 10.1109/TSE.2021.3058985.

Q. Zheng, P. Zhao, D. Zhang, and H. Wang. Mr-dcae: Manifold regularization-based deep convolutional autoencoder for unauthorized broadcasting identification. *International Journal of Intelligent Systems*, 36(12):7204–7238, 2021.

Q. Zheng, P. Zhao, H. Wang, A. Elhanashi, and S. Saponara. Fine-grained modulation classification using multi-scale radio transformer with dual-channel representation. *IEEE Communications Letters*, 26(6):1298–1302, 2022.

Q. Zheng, X. Tian, Z. Yu, Y. Ding, A. Elhanashi, S. Saponara, and K. Kpalma. Mobilerat: A lightweight radio transformer method for automatic modulation classification in drone communication systems. *Drones*, 7(10):596, 2023.

Q. Zheng, S. Saponara, X. Tian, Z. Yu, A. Elhanashi, and R. Yu. A real-time constellation image classification method of wireless communication signals based on the lightweight network mobilevit. *Cognitive Neurodynamics*, 18(2):659–671, 2024.