SECURE PROGRAMMING PRACTICES AND OPEN-SOURCE SYSTEMS (EMPIRICAL INVESTIGATION)

Saleh M. Alnaeli[1]*, Salah Alhadi Gbeg[2], Salah A. Jowan[2], Mahmood Saad Shertil[3]

[1]Computer Science Dept., University of Wisconsin, Math, Stat, Menomonie, WI, 54751
[2]Computer Department, Faculty of Science, Al-Asmarya Islamic University, Zliten, Libya
[3] Computer Department, Faculty of Science, Elmergib University, Alkhoms, Libya
** Corresponding author: alnaelis@uwstout.edu*

ABSTRACT

In the era of Open-Source Systems (OSS), security is one of the most important issues that have a direct impact on the reliability of software systems. Security can significantly be affected by the way programmers write their code and their level of proficiency when it comes to secure programming practices. In many problem domains, open-source systems are written by participants with sufficient experience in their fields. However, it is not unusual for some of those participants to have a limited background in secure programming practices. A study that examines the presence, prevalence, and distribution of code vulnerabilities in scientificOpen-Source systems is presented. This empirical investigation statically analyzes three systemsdeveloped in C and C++ languages comprising over three million lines of source code. The study aimed to provide empirical evidence that shows some of the common vulnerabilities that are introduced to the open-source systems during the implementation phase. The findings are meant to be used for designing proper training courses and enhancing the academic computing curriculum. A cloud-based analysis tool developed by a team from the University of Wisconsin is used in this study. The findings confirm the presence and show the distribution of some vulnerabilities in the code introduced by programmers confirming the need for proper relevant training and education.

Keywords:secure programming, open-source systems, cloud computing, static analysis, vulnerabilities, unsafe functions

1. INTRODUCTION

With the exponential increase of the number of software systems that are launched as open-source systems by communities from different problem domains [1-7], security tops the issues that have gained research communities in both industry and academia. Software vulnerabilities are weak points in software systems that can maliciously be exploited by hackers. Most of the source vulnerabilities usually occur during the

implementation (coding) phase. These vulnerabilities are the result of many different factors. One well-known factor is the way programmers write their code with the presence of a lack of Focus on security standards and good practices. Writing source code without being well prepared and able to follow the secure programming practices will most likely lead to weaknesses in the source code which are often derived from poor coding behaviors, habits, and not following policies in place. Hackers usually focus their efforts on finding these weaknesses and exploiting them, often to their benefit within whatever context.

For this reason, many software vendors and complier designers have recommended that programs be trained to follow secure programming practices [8]. In fact, some programming language designers have put a nontrivial effort to improve their libraries and, in some cases, banning some vulnerable functions (e.g., C string unsafe functions). The question that is usually being asked by Open-source software communities is "Do those programmers that work with OSS follow these secure programming standards and practices?". Many studies have suggested and recommended different approaches [7,8]. In this study, authors have teamed up with a research group from The University of Wisconsin-Stout that have been working on this problem and are well-known to the research community to empirically evaluate a cloud-based tool that uses static analysis to detect and visualize some of the source code vulnerabilities that can negatively impact the software quality and security [6-8]. Additionally, our goal to uncover some relevant trends from a software engineering perspective regarding how people from the scientific community write the code. Do they follow security and good practices standards? Three Open-source systems are analyzed in this investigation chosen carefully and commonly used by the

scientific computing domain.The work provides insights and eye-opening statistics about open-source development practices concerning coding security and vulnerability. We believe that the work will also help show the importance of maintaining a high level of security on those systems and how to accomplish that using the cloud-based tool that is being used in this investigation. [6, 7].

In this empirical investigation, several metrics are used chosen among some known vulnerabilities [9-12]. Mainly, the study will be conducted within the context of the following three research questions:

Q1: Do open-source programmers follow secure programming practices?

Q2: What are the common insecure patterns found in the analyzed system?

Q3: What is the distribution of the insecure patterns, used in this study, found in the studied systems?

This is an empirical investigation which means analysis is completely conducted at the source code level on an actual source code. The source code is automatically extracted from the software repositories (GitHub). The source undergoes some transformations and then parsed using the cloud-based tool developed by a research team from the University of Wisconsin.

The remainder of this article is organized as follows. Section 2 presents a brief literature review. Section 3 describes the methodology followed in this empirical investigation and relevant information. Section 4 introduces the design of the study, evaluation, and presents findings and uncovered trends in the studied systems. It also discusses the results and the facts revealed by

the investigation. The limitations of the study and threats to validity are discussed in section 5, andsection 6 is the conclusions of the study.

2. LITERATURE REVIEW

In this study, Open-source systems from scientific problem domains are considered and we intend to support the quality, reliability, and security of existing source code. Additionally, we aim to provide some insights and facts for the computer science and relevant fields programs to improve their programming and software engineering curriculum. Currently, a substantial percentage of the scientific software systems in both the academia and enterprise market use open-source software models. Clearly, this makes the development process easier for both developers and third-party vendors and institutions. However, it poses some risks and exposes some potential vulnerabilities as the source code is accessible by everyone including attackers, and programming participants tend to be very proficient in their fields, but some could lack sufficient secure programming skills. Unfortunately, if vulnerabilities or flaws are found by attackers, they will use them to cause harm to the software system in one way or another depending on the sensitivity of the systems and services provided by those victim systems [13].

The source code vulnerabilities involve complex or poorly written code introducing holes that attackers can use to conduct malicious activities lead to exposing sensitive data, interrupting service, or damaging the attacked software system.

While there are multiple reasons and ways when it comes to introducing vulnerabilities to the open-source software systems [7-13], our discussion is within the context of using some of the well-known insecure patterns that

are used by unprepared and well-trained programmers (from secure programming perspectives) that includes unsafe C/C++ functions, unproperly handled potential divide by zero operations, dynamic memory accessed after deleting the designed memory, and unsafe implicit type casting (e.g., assigning float values to integer variables).

The bulk of previous research on this topic has focused on detecting different vulnerabilities as they are being reported by computing communities and end-users [7-13]. Approaches varied from static analysis to ad hoc techniques. However, the work presented here differs from previous work on scientific open-source systems in that we conduct an empirical study of source code vulnerabilities at the source code level and all the potential challenges with a minimum level of false-negative and false-positive cases. Additionally, the cloud-based tool used in this study makes it easier to extract the entire large-scale repositories of the analyzed systems directly from the repositories servers and transform all the source files code into an XML format that would allow the static analysis to be more practical and accurate, and extremely fast [8, 14]. Finding are visualized so that they are more readable and easier to comprehend. The tool srcML is used to convert code to XML rapidly [14]. srcML is able to markup source code with XML which allows for intelligent searching of known insecure and vulnerable patterns in codebases.

Previous studies of millions of lines of code were conducted, only detecting unsafe function calls that are considered as one of the most dangerous vulnerable function that has previously been exploited by attackers [7-11]. Alnaeli et al. [7, 8] have been actively researching and developing techniques to help developers detect unsafe function calls. Their findings

showed that the number of unsafe functions tends to rise over time which suggests a degrading of safe security practices [7, 8]. However, in this investigation, a new cloud-based tool developed by the team that is able to go beyond the unsafe function calls deduction. The study will be used to analyze three scientific systems. The tool is built on the cloud and able to scale safely. The access to the tool was granted as a part of a collaborative effort between Alasmarya Islamic University and the University of Wisconsin software teams [8]. The intention of this work is to someday be used to help establish fluid standards for scientific software developers in particular and computer science students in general. To the best of our knowledge, the empirical study presented is the only one of its kind in Libya, in both academia and industry, to date.

3. METHODOLOGY AND DATA COLLECTION

We now describe the methodology we used to detect the vulnerable patterns in the studied systems and collect the data for our case study. The softwaretool used in this studyis discussed first followed by the major security concerns (metrics). Techniques to determine the presence of the

TABLE I. THE THREE OPEN-SOURCE SYSTEMS USED IN THE STUDY

System Name	Scilab	LibreCAD	DPDK
System Domain	Cross-platform numerical computational package	2D CAD drawing tool	A set of libraries and drivers for fast packet processing (Linux)
#Lines of Code	1,167,178	269,864	2,098,533
# Source Code Files	6,354 (C/C++, .h)	848 (C/C++, .h)	3,168 (C/C++, .h)
More Information	terms of the GNU General Public License (GPL) v2.0.	terms of the GNU General Public License (GPL) v2.0.	terms of the GNU General Public License (GPL) v2.0.

vulnerabilities in the source-code are generally conservative and, in some situations, label places as having a source-code vulnerability when in factthere may not be one, or an insecure pattern is being used in a nonvulnerable manner. This is referred to as a potential vulnerability. The static analysis required to identify all the actual cases from the potential causes can be quite expensive. In some cases, it cannot be done by static analysis and requires some form of dynamic analysis. Here we limit our detection approach to a static analysis approach.

3.1 THE CLOUD-BASED TOOL USED IN THIS STUDY

We used a cloud-based software analysis tool developed by Alnaeli et al. [8], to analyze source code files and determine if they contain any source code vulnerabilities chosen in advance as defined in this section. The tool retrieves, on the fly, files with C/C++ source-code extensions (i.e., c, cpp, and h). Then the tool uses the srcML (www.srcML.org) toolkit to parse and analyze each file [14]. The srcML format wraps the statements and structures of the source-code syntax with XML elements, allowing tools to use XML APIs (e.g., XPath) to locate such things as source code blocks and to analyze expressions. Once in the srcML format, the cloud tool iteratively finds each source code block and then analyzes the expressions in the block to find the different vulnerabilities. A count of each vulnerable pattern per system is recorded. It also records the number of safer replacements found which can help understand if there are developers who are aware of the faster replacements to the unsafe ones. The final output is a visualized report of the findings of each system analysis with one or more types of source code vulnerabilities.

3.2 THE VULNERABLE CODE PATTERNS USED IN THIS STUDY

The vulnerability analysis cloud-based tool checks for several source code patterns. These patterns are detected by the tool because of the vulnerabilities they can pose to a system. Now, we will briefly discuss each of the different vulnerable patterns used in this investigation. We also describe how the tool finds and counts each of them along with any limitations of the approach.

3.2.1 Race Conditions

A major security concern is race conditions [15]. The vulnerability analysis tool checks for several race conditions under sections of code that run in parallel. This includes OpenMP parallelized for-loops, std::thread parallelized functions, and p-thread parallelized functions. All variable declarations in the parallelized block are collected. Then all variable usages are gathered. Each variable usage is assessed to determine if a modifying operation is being applied to the variable.For example, the statement, "x += 2," would count as a modifying operation. After a list of modified variables is complete, each variable is compared to the list of declared variables in the parallelized block to determine if any were declared outside of the block. Finally, any modified variables confirmed to be declared outside the block are marked as potential race conditions.False positives due to reduction, atomic, locks, critical, and other thread-safety patterns, are ruled out.

3.2.2 Unsafe Functions

Functions that are considered unsafe by the tool are functions that have been deemed unsafe by the research community [10]. For example, the standard C library includes the gets() function that is used by programmers for reading a sequence of characters (c-strings) from the user. The function

cannot verify the length of the entered string making it possible for the user to exceed the maximum expected length. As a result, an attacker trying to write LENGTH + MORE bytes into the designated buffer will always succeed if newlines were excluded from the data [16]. Therefore, locations of the memory that are adjacent to the buffer in the memory will most likely be corrupted. That means an attacker can modify or corrupt data and can even overflow the stack, leading the program to run into an unexpected status which causes unpredictable results. Our tool canparse the blocks that include the calls to unsafe functions semantically to exclude all the function calls that use wrappers so that the false negatives are significantly minimized.

3.2.3 Implicit Type Casting

Implicit type casting is detected when data is lost due to typecasting.For instance, the tool detects a specific case of buffer overflow in which a stream insertion operator "<<" is used to direct the contents of a stream into a fixed-size array without any checks on the size of the stream contents [8].

3.2.4Potential Division by Zero

Division by zero can lead to unexpected exceptions, which can be considered a security vulnerability. The tool analyzes the source code looking for any operations that involve unsafe division where zero can be a denominator. The patterns are semantically analyzed as well so that the false negatives are significantly minimized.

3.2.5 Deallocated Dynamic Memory

Deallocated memory access can allow unauthorized access to data. This situation is known as dangling pointers or null pointers [17]. For examples,

if the *delete* command is used in C/C++ to release a dynamically allocated memory pointed by a pointer and then the program tries to access the released memory via the pointer, if not properly, they can be exploited by attackers who can use it in their favor [15-17].

4. FINDINGS AND DISCUSSION

We now study the distribution of three chosen scientific systems for this empirical study. Table 1 presents the list of systems examined along with the number of files, problem domain, and LOCs (Lines of Code) for each of them. These systems were chosen because they represent a variety of applications within the scientific domain that are widely used. We have a strong feeling gained from their popularity in academia and literature that they represent a good reflection of the types of systems that would undergo development and are targeted for regular maintenance and evolution processes in general. Additionally, we believe that the chosen open-source systems are written and evolved by developers from different relevant scientific backgrounds and various levels of secure programming and practices familiarities.

4.1 Design of The Study

Our study focuses on two main aspects regarding source code vulnerabilities in the open-source systems used in the scientific domain. First, the presence of vulnerable patterns and unsafe functions in the analyzed system; this gives a handle of how much open-source systems can be trusted and how much work needs to be expected to address these vulnerabilities either manually or using automated tools. This will hopefully motivate the stakeholders of those systems to hire or adopt adequate techniques to observe their systems as they are being developed and evolved. Second, we

examine which vulnerability patterns among the ones used in this study are most prevalent. We are interested in revealing the distribution of the inhibitors in the systems. This can play an important role in prioritizing the refactoring processes that need to be undertaken to address and fix the found vulnerabilities. For example, the most prevalent one needs to be fixed first for more effective outcomes.

We propose the following research questions as a more formal definition of the study.

R1: Do open-source programmers follow secure programming practices (are source code vulnerabilities found in the studied systems)?

R2: What are the common insecure patterns found in the analyzed system?

R3: What is the distribution of the insecure patterns, used in this study, found in the studied systems?

Next, we now examine our findings within the context of these research questions.

4.2 The Detected Vulnerabilities and Their Distribution

We now present and discuss our findings and main observations, along with some general trends found in the studied systems.

Results collected in this investigation are presented in tables II and III and visualized in Fig. 1, Fig. 2, and Fig.3 for the studied systems. A count of how manytimes the considered vulnerable patterns including insecure

TABLE II. THE DISTRIBUTION OF SOURCE CODE VULNERABILITIES DETECTED IN THE STUDIED SYSTEMS

Vulnerabilities	**Scilab**	**LibreCAD**	**DPDK**
Calls to known Insecure Functions	3,983	456	8,257

Implicit Type Casting	945	253	815
Memory Access After Deletion	555	172	0
Potential Division by Zero	42	2	15
Race-Condition	0	4	1
Uninitialized Varables	1,570	210	4,082
Total	7,095	1,097	13,170

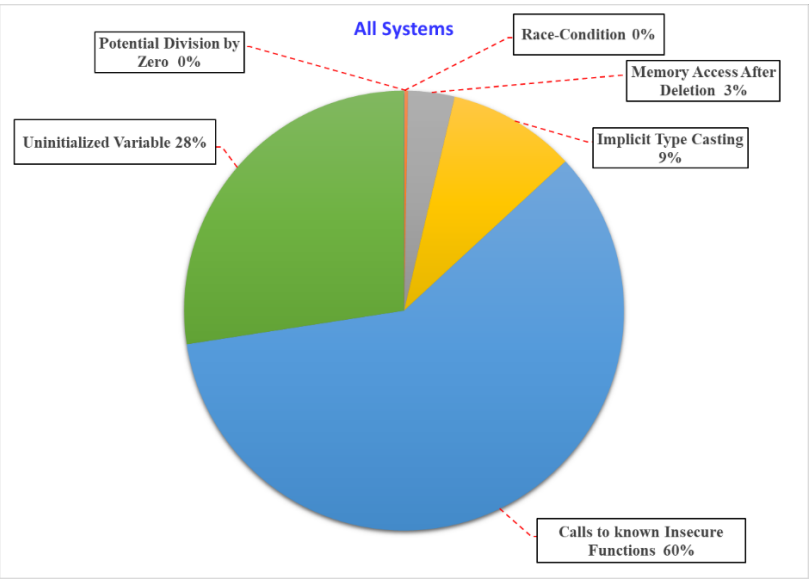

Fig. 1. Vulnerable Patterns Detected in The Studied System and Their Distribution

functions werefound in each system are shown in the table. Fig.1 shows the distribution of the vulnerable patterns detected in all the analyzed systems. The insecure function calls represent the most prevalent vulnerable patterns among all the patterns used in this study. This is somehow consistent with the literature and prior studies conducted on systems from different problem domains [hidden for blind review]. The uninitialized variables come second

for Scilab and DPDK. However, for the LibreCAD system, the implicit type casting was the second prevalent vulnerable pattern found by the tool.

Fig. 2 presents the total number of vulnerable patterns (used in this study) found in each system. The figure shows that the DPDK has the vast majority followed by Scilab and then LibreCAD. Considering the difference in the size of the systems, this is expected. But the results in general address the first research question, R1, confirming that the developers of these open-source systems are introducing vulnerabilities to the systems. Detailed distribution is shown in Fig. 3 for each system.

Fig.3 also presents the distribution of the unsafe functions called in each system individually, besides the other vulnerabilities, shown in different colors. This information is very important as It can be used to prioritize the refactoring and fixing tasks that can be conducted to improve the security and the quality of the source code for the studied systems.

One item of interest in TABLE II. is that Scilab and LibreCAD have a more serious issue when it comes to dealing with dynamic memory allocation. That is, the invalid memory access caused by improperly handling the deleted dynamic memory is 555 for Scilab and 172 in LibreCAD. However, interestingly, no access to a released pointer was deducted in DPDK**.** As Fig. 3 shows that dynamic memory allocation has been used in DPDK, this indicates that the developers of this system are efficiently able to keep tracking of the released pointers so that they are not used if the former locations are released by the program leading to no pointer dangling issues. This is not quite the case for the other two systems. Again, the numbers make all the systems vulnerable to attacks if hackers were able to exploit these vulnerabilities found in the studied systems.

Although the total numbers were relatively smaller in Libre, the ratio of the number of lines and number of the detected vulnerability make it not the most efficient when it comes to the usage of vulnerable patterns used as metric in this investigation.

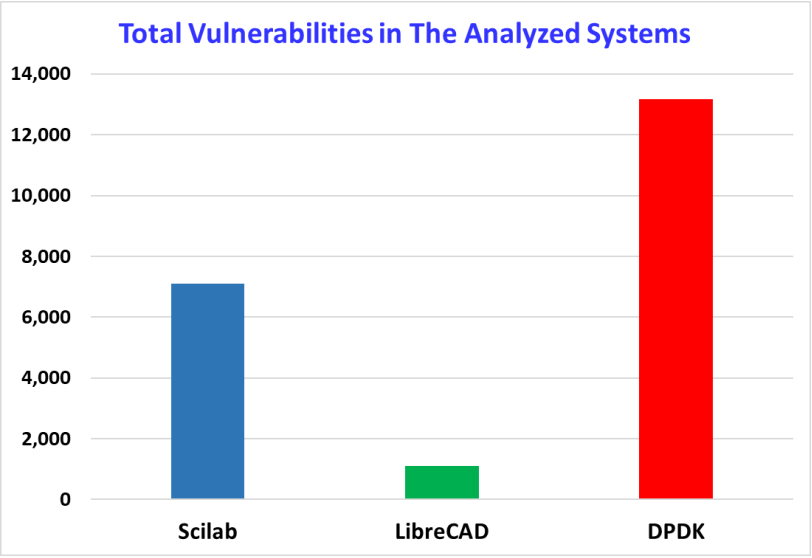

Fig. 2. Total Number of Vulnerable Patterns Found in Each System

It is interesting to note that unsafe functions are still being used in the studied systems even though it is considered a bad programming practice and their risk is well-known to the research community. We did some spot inspections and found many cases of these detected unsafely used confirming our concerns and supporting the aforementioned answers to the first and second research questions, R1 and R2. Of course, these findings are somewhat consistent with the literature and prior investigations [7].

TABLE II. shows that the presence of uninitialized variables is very significant across all the studied systems. This is something that needs to be addressed by the admins of the systems as it may impose serious vulnerabilities that affect the quality of the systems. The code uses a variable that has not been initialized, leading to unpredictable or unintended results. In other words, in C/C++ variables are not initialized by default so they typically contain junk data with the contents of stack memory before the parent block is invoked. An attacker can control or read these contents so the presence of uninitialized variables can sometimes indicate a typographic error in the code [11, 18].

The number of possible race-conditions across all systems was very trivial. This somehow indicates that the systems are all written sequentially and not in a multithreaded manner. This is a bad sign when it comes to the ability to take the advantage of the multicore architecture, however, this beyond the scope of this investigation. Additionally, the number of statements that have the potential to lead to division by zero was visible in Scilab and DPDK and very neglect in LibreCAD.

TABLE II, fig.1, and fig.2 address R2 and R3 and give more details of our findings on the distribution of detected vulnerable patterns. Additionally,

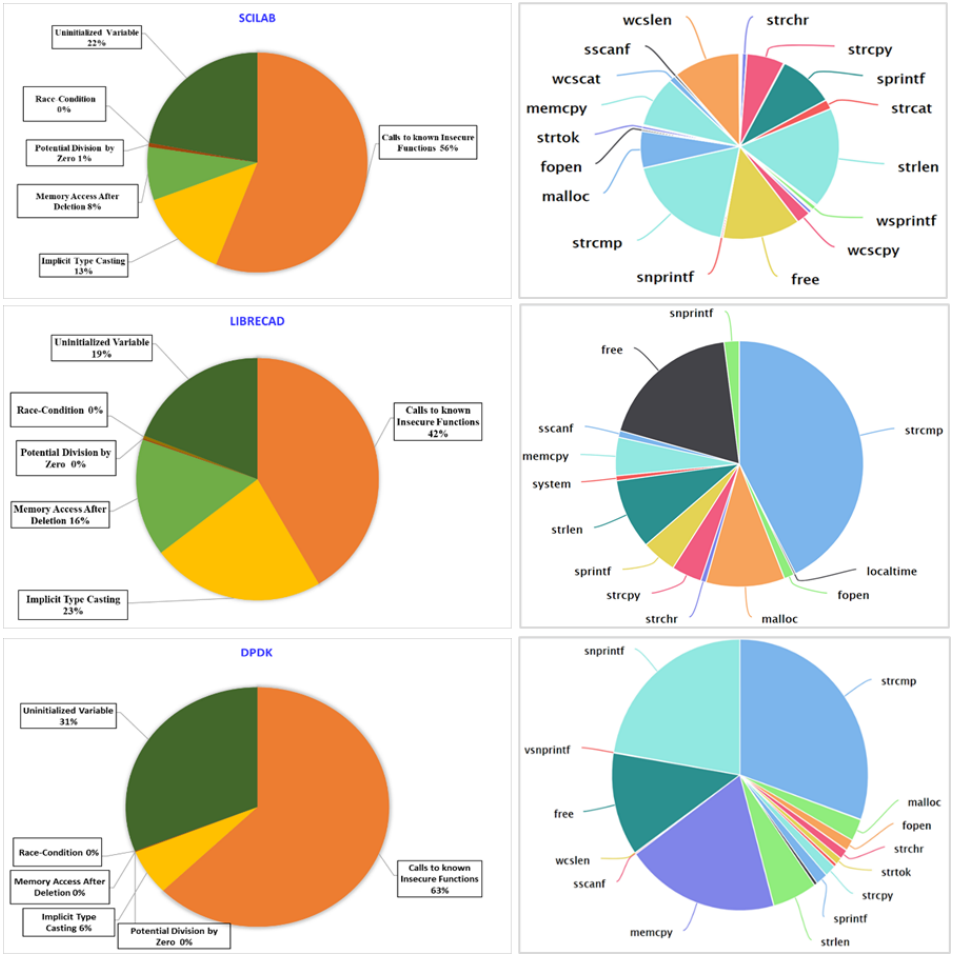

Fig. 3. Calls to Insecure functions and Other Vulnerabilities Detected in The System and Their Distribution for Each System

TABLE II presents the counts of each vulnerable pattern that occur within each system. The systems have multiple vulnerabilities. As can be seen, insecure functions' invocations are by far the most prevalent in all the studied systems, addressing R2 and R3. Fig 3. shows the distribution of the calls to unsafe functions. Clearly, the DPDK system has the largest number followed by Scibal. One reason is that the portion of the code written in C is large in these two systems.

4.3 The Safer Replacement and safer use ofUnsafe Functions

We are interested in knowing whether or not the developers of the studied systems use safe replacements introduced to the C/C++ language. We believe that this information will give us the ability to explain some of the other findings and teach us about the demography of the developers, from experience and proficiency perspectives. We used the tool to detect all of the safer replacements used in the studied systems. Additionally, we counted the number of times the unsafe functions have been used in a careful manner that is less likely to introduce any vulnerability to the system (preceded with proper wrappers). The attention was to reduce false-positive cases for more accurate results.

This will give us some insights into how unsafe functions are being used. Those patterns have not been double counted and are only shown in TABLE III. Even though the number of cases where unsafe functions were safely invocated is high in DPDK and Scilab, this is not a good practice from a software engineering perspective, since safer functions are available. That is, using the unsafe functions should not be recommended here and this is a point that can degrade the quality of the source code of these two systems. The developers of this system should be recommended to take some training and get exposed to newer and up to date safer replacements. This will allow them to suggest some refactoring tasks to remove unsafefunctions and replace them with safer options [10].

TABLE III. SAFER REPLACEMENTS USED IN THE STUDIED SYSTEMS

Vulnerabilities	Scilab	LibreCAD	DPDK
UNSAFEFUNCTIONUSEDSAFELY	213	8	793
SAFER REPLACMENTS	149	34	2,816

Another interesting observation is that the developers of the studied systemsseem to have different levels of background in secure programming practices. This is very evident via the findings shown in TABLE II and III where both of the numbers of unsafe functions and their replacements are nontrivial. Additionally, the findings have shown that some of those programmers who use unsafe functions find their way to manage to use them in a less harmful way. We did some spot inspections and found several cases of unsafe function call being used within blocks that have wrappers making it safer to use the insecure functions. Ideally, developers should be trained to replace the unsafe functions with safer replacements and refrain from using the known insecure and vulnerable functions.

DPDK shows better trends when considering the ratio of unsafe functions to the safer replacements used in the system. This indicates that the developers of this system have more skills from a security perspective compared to the other systems. As this observation cannot be generalized, a further investigation needs to be conducted to see if the unsafe and safe functions are introduced by different people or the system is being under refactoring operations that involve replacing the unsafe functions with safer ones. These trends can be investigated and uncovered if a historical study is conducted. This is something the authors are considering for a future study.

5. LIMITATIONS AND THREATS TO VALIDITY

For most of the static analysis security tools in the market, it is not unusual to have a nontrivial number of false-negative (missed vulnerabilities) and false-positive cases affecting the accuracy of the results [14]. Although we know that the cloud-based tool used in this study is constantly being

improved, we still expect it to miss some certain cases or return false positives, giving inaccurate results [8].

The dead code was not excluded from our study. That is, upon examination of the detected patterns in the study we found that some of them were part of dead code, i.e., code that would never be executed. As part of the static analysis, there was no distinction made in the study between vulnerabilities found in dead code or active code that might have an impact on the accuracy of the findings. We believe that the dead code will not impose any risk even if it contains insecure patterns. In the future, we are planning to refine the results by excluding the dead code.

6. CONCLUSION AND FUTURE WORK

This study uses a cloud-based tool to empirically examine the usage of vulnerable patterns in open-source systems from the scientific problem domain. Three large-scale systems are investigated as a case study for this empirical study. The analyzed systems are all written in C/C++ languages and comprising more than three million lines for source code. We found that the developers of these systems have used different insecure patterns and functions that, if exploited by the attackers, could cause security issues.

We found that the most vulnerable patterns found in the studied system are known to be very exploitable but could be avoided. This is evident as some developers of the studied systems were found to be aware of and using more secure and safer replacements. As such, more attention needs to be placed on dealing with and observing how open-source systems develop and evolve so that they are more secure and reliable. While we cannot completely generalize this finding to all software systems (across scientific domains) there is some indication that this is a common trend.

The recent ubiquity of the open-source development model gives rise to the need to educate developers and make them more aware of the secure programming standards and source code vulnerabilities. We echo the demand for the need to develop standards and idioms that help developers in avoiding vulnerable patterns. We found the cloud-based tool to be very helpful and easy to use. It is very scalable and can give results significantly fast in a readable way. Additionally, we recommend that the tool to be open-source itself so that more vulnerability checkers are developed and integrated into the tool. Having this tool as a plugin to C/C++ IDEs will be a huge asset to the scientific computing community.

ACKNOWLEDGMENT

Software Engineering and Security Research team at The University of Wisconsin- Stout, and Parkside for giving us access to the Cloud-Based source code vulnerabilities analysis tool.

REFERENCES

[1] St. Laurent, Andrew M. (2008). “Understanding Open Source and Free Software Licensing.”, O'Reilly Media. p. 4. ISBN 9780596553951.

[2] Levine, Sheen S.; Prietula, Michael J. (30 December 2013). "Open Collaboration for Innovation: Principles and Performance". Organization Science. ISSN 1047-7039.

[3] D. Cubranic, K.S. Booth, “Coordinating open-source software development”, IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. Stanford CA (16-18 Jun 1999), pp. 61-66

[4] J. J. Heiss.,“The meanings and motivations of open-source communities.”, Aug 2007, from Oracle,http://www.oracle.com/technetwork/articles/java/opensource-phipps-137190.html.

[5] L. Pascarella, F. Palomba, M. Di Penta, and A. Bacchelli, "How Is Video Game Development Different from Software Development in Open Source?," 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, 2018, pp. 392-402.

[6] E. Crifasi, S. Pike, Z. Stuedemann, S. M. Alnaeli and Z. Altahat, "Cloud-Based Source Code Security and Vulnerabilities Analysis Tool for C/C++ Software Systems," *2018*

IEEE International Conference on Electro/Information Technology (EIT), Rochester, MI, 2018, pp. 0651-0654, doi: 10.1109/EIT.2018.8500206.

[7] M. Block, B. Barcaskey, A. Nimmo, S. Alnaeli, I. Gilbert and Z. Altahat, "Scalable Cloud-Based Tool to Empirically Detect Vulnerable Code Patterns in Large-Scale System," 2020 IEEE International Conference on Electro Information Technology (EIT), Chicago, IL, USA, 2020, pp. 588-592, doi: 10.1109/EIT48999.2020.9208325.

[8] D. Wahyudin, A. Schatten, D. Winkler, and S. Biffl, "Aspects of Software Quality Assurance in Open Source Software Projects: Two Case Studies from Apache Project," 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007), Lubeck, 2007, pp. 229-236, doi: 10.1109/EUROMICRO.2007.19.

[9] Y. Joonseok, R. Duksan, and B. Jongmoon, "Improvingvulnerability prediction accuracy with Secure Coding Standardviolation measures," in 2016 International Conference on BigData and Smart Computing (BigComp), 2016, pp. 115-122.

[10] M. Howard. "Security Development Lifecycle (SDL) BannedFunction Calls" [Online]. Available:https://msdn.microsoft.com/en-us/library/bb288454.aspx

[11] Z. Xu and G. Liu, "STACKEEPER: A Static Source Code Analyzer to Detect Stack-based Uninitialized Use Vulnerabilities," 2018 IEEE 4th International Conference on Computer and Communications (ICCC), Chengdu, China, 2018, pp. 2180-2184, doi: 10.1109/CompComm.2018.8780675.

[12] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, 2018, pp. 372-383, doi: 10.1145/3180155.3180201.

[13] Gilad David Maayan, "The Dangers of Open-Source Vulnerabilities, and What You Can Do About It", Aug 19, 2019, https://securitytoday.com/articles/2019/08/19/

[14] M. L. Collard, M. J. Decker, and J. I. Maletic, "LightweightTransformation and Fact Extraction with the srcML Toolkit,"presented at the SCAM'11, Williamsburg, VA, USA, 2011.

[15] C.C. Michael. S. Lavenhar, "Source Code Analysis Tools - Overview," CISA Cyber Infrastructure, 2013.

[16] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "ITS4: a static vulnerability scanner for C and C++ code," in Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference, 2000, pp. 257-267

[17] L. Dong, W. Dong and L. Chen, "Invalid Pointer Dereferences Detection for CPS Software Based on Extended Pointer Structures," 2012 IEEE Sixth International Conference on Software Security and Reliability Companion, Gaithersburg, MD, 2012, pp. 144-151, doi: 10.1109/SERE-C.2012.30.

[18] The National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD. "Vulnerability Summary for the Week of September 14, 2020" [Online]. Available:https://us-cert.cisa.gov/ncas/bulletins/sb20-265