ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 3, 1502-1507 (2017)

www.astesj.com

Special Issue on Recent Advances in Engineering Systems

ASTESJ

ISSN: 2415-6698

Source Code Vulnerabilities in IoT Software Systems

Saleh Mohamed Alnaeli[*,1], Melissa Sarnowski[2], Md Sayedul Aman[3], Ahmed Abdelgawad[3], Kumar Yelamarthi[3]

[1]*CSEPA, University of Wisconsin-Colleges, 53715, USA*

[2]*Computer Science, University of Wisconsin-Fox Valley, 54952, USA*

[3]*College of Science and Engineering, Central Michigan University, 48859, USA*

ARTICLE INFO

Article history:
Received: 02 June, 2017
Accepted: 21 July, 2017
Online: 15 August, 2017

Keywords:
unsafe commands
vulnerable software
scientific
security
static analysis
historical trends

ABSTRACT

An empirical study that examines the usage of known vulnerable statements in software systems developed in C/C++ and used for IoT is presented. The study is conducted on 18 open source systems comprised of millions of lines of code and containing thousands of files. Static analysis methods are applied to each system to determine the number of unsafe commands (e.g., strcpy, strcmp, and strlen) that are well-known among research communities to cause potential risks and security concerns, thereby decreasing a system's robustness and quality. These unsafe statements are banned by many companies (e.g., Microsoft). The use of these commands should be avoided from the start when writing code and should be removed from legacy code over time as recommended by new C/C++ language standards. Each system is analyzed and the distribution of the known unsafe commands is presented. Historical trends in the usage of the unsafe commands of 7 of the systems are presented to show how the studied systems evolved over time with respect to the vulnerable code. The results show that the most prevalent unsafe command used for most systems is memcpy, followed by strlen. These results can be used to help train software developers on secure coding practices so that they can write higher quality software systems.

1. Introduction

This paper was originally presented in 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT) [1]. Additional research has been done to extend the research to examine the security of a much larger group of IoT software systems. While the IoT continues to grow to billions of devices running a huge variety of software systems, both open source and proprietary, security becomes a major concern for individuals and organizations who use the IoT in both academia and industry. The security of every software system becomes vital as each software system and device could be a target or an access point to hackers, or lawbreakers [2-4]. Most of the software and embedded systems used for IoT applications are currently available as open source systems and are therefore developed by programmers from varying disciplines and with different backgrounds. Many of those programmers have little to no background on security challenges imposed by the usage of vulnerable source code in some programming languages (e.g., C/C++) caused by commands that are well known to the research community as being unsafe and are banned by companies such as Microsoft.

According to [5, 6], most of the detected security threats are due to vulnerabilities in the code. Thus, minimizing usage of insecure commands can play an important role in protecting the software systems from any potential attacks. In order to minimize the use of unsafe commands at the source code level, developers need to be made aware of those unsafe commands and the security issues their usage can cause. Educating developers and ensuring that they follow good programming practices can minimize the time and effort spent on finding and fixing them in later stages, as well as lessen the expense of fixing a security issue if it's exploited

[*]Corresponding Author: Saleh Mohamed Alnaeli, 1478 Midway Rd, Menasha Wisconsin 54952, (920) 832-2615, USA | Email: saleh.alnaeli@uwc.edu

by an attacker. For example, C/C++ programming languages are preferred by developers because of the level of performance, flexibility, and efficiency that they offer. However, security vulnerabilities (e.g., integer vulnerability, buffer overflow, and string vulnerability) need to be addressed and avoided from the beginning when writing the source code to save the time and expense that comes with having to refactor the system at a later date to remove those security vulnerabilities. Additionally, programmers also need to make sure that they do not use commands that are known to cause security concerns (e.g., memcpy, strlen, and strcmp).

As an example, the standard C library includes a function called gets() that is used primarily for reading strings input by the user. This function accepts a pointer from data type char as a parameter and reads a string of characters from the standard input, placing the first character in the location specified by that pointer and subsequent data consecutively in memory. The gets() function will continue reading until a newline is detected, at which point the buffer is terminated with a null character. The issue is that the developer cannot determine the size or length of the buffer passed to gets() prior to runtime. Because of this, when the buffer is size bytes, an attacker trying to write size + extra bytes into the buffer will always succeed if the data excludes newlines [5]. Consequently, memory locations that are adjacent to the buffer in the memory may be overwritten, which could lead to sensitive data being modified if it is stored in those adjacent memory locations. Additionally, an attacker could even overflow the stack and lead the program to run into an arbitrary or unexpected status. A safe alternative to gets() is fgets(), which, unlike gets(), also accepts an integer number as a parameter that acts as the limit of characters copied into the string, including the null-character at the end.

The process of protecting software systems from vulnerability issues at the source code level is done by either completely removing known unsafe commands and functions, or by replacing them with safer replacements (e.g., strncpy, strncmp, strncat etc.). Here, we have extended a previous empirical examination of some of the open source software systems used for IoT to better understand how well IoT developers do when it comes to the usage of vulnerable code by studying a larger group of IoT systems and the history of some of those systems to analyze the change in the usage of unsafe commands over time. We are particularly interested in determining the most prevalent unsafe statements that occur in a wide variety of IoT software applications and if there are general trends, and are interested in seeing if the trends stay the same or change among a larger group of systems, as well as if there are historical trends for the group of systems whose history was also examined. While this work does not directly address the problem of removing unsafe source code from the systems, it does serve as a foundation for understanding the problem requirements in the context of a broad set of applications. Moreover, the focus of this research is on unsafe statement detection and identifying their distribution over time.

In this study, several software systems used for IoT and developed in C/C++ are analyzed and evaluated with respect to the usage of vulnerable commands that can affect the quality of IoT systems. Each of the studied systems are written in C/C++. For each system, the source code is analyzed a count is created for each of the unsafe functions as well as the safer replacement functions. These counts are compared against the counts of each other system to uncover trends and make observations about the usage of unsafe functions. Furthermore, the history of some of the systems, if it was available and of a sufficient length, is examined and the number of detected unsafe functions is calculated for each release.

This work focuses on addressing the following questions: how many unsafe functions are used in these systems, which of those are the most frequent, and what are their respective distributions? Are the numbers or distributions of unsafe functions changing over the history/versions of a software system? When a larger group of systems is studied, do the trends uncovered in the original study remain the same, or do new trends emerge?

This work contributes by extending one of the only studies on the usage of vulnerable functions in IoT software applications. We found that the functions memcpy and strlen represent most of the unsafe functions occurring in these systems, in both the original systems and the additional systems studied in this extension. Moreover, the findings show that, for the most part, the systems reviewed have become more vulnerable to attacks over time due to the increase in the number of the unsafe functions used over time. This knowledge will assist researchers in formulating and directing their work to efficiently address this problem when refactoring and designing new systems.

The remainder of this paper is organized as follows. Section 2 gives background information and related work on the topic of source code vulnerabilities and security of IoT and its challenges. Section 3 describes the methodology used in the study along with how we performed the analysis to create counts for each unsafe function. Section 4 presents the findings of our study of 18 open source software systems used for IoT. There is a discussion of results in the same section as well. The historical trends of 7 of the systems found are explained in section 5, followed by threats to validity and future enhancement in 6 and 7. Finally, conclusions are found in section 8.

2. Background and Related Work

Most of the previous research conducted on IoT security has focused on identifying security concerns related to the communication processes and authentication methods used with IoT. Some other studies have examined data privacy within various levels. To the best of our knowledge, no study has examined the usage or distribution of vulnerable code in IoT systems that are developed in C/C++.

A study has been conducted by VERACODE [7] to investigate a selection of always-on consumer IoT devices to understand the security posture of each product. They found that product manufacturers prioritized design instead of security and privacy, putting consumers at risk for an attack or physical intrusion. Their team performed a set of uniform tests across all devices and organized the findings into four different domains: user-facing cloud services, back-end cloud services, mobile application interface, and device debugging interfaces. The results showed that most of the tested devices exhibited vulnerabilities across most categories. The findings prove that there is a need to perform security reviews on device architecture and accompanying applications to minimize the risk to users.

Hui Suo, Jiafu Wan, Caifeng Zou, and Liu in [2] reviewed security in the IoT and analyzed security characteristics and requirements from four layers: perceptual layer, network layer, support layer, and application layer. They discussed the research status in this field from encryption mechanism, communication security, protecting sensor data, and encryption algorithm. While they confirmed the need to develop technologies and methodologies that meet the IoT needs for meeting the higher requirements of security and privacy, they did not discuss security concerns that can be caused by programming standards or by using vulnerable code.

A research group from George Mason University and National Institute of Standards and Technology in [3] presented a set of use cases that leverage commercial off-the-shelf services and products to raise awareness of security challenges in current practices and prove that there is a need for IoT security standards to be developed, as well as their possible implications. They recommended that experts begin formulating suitable guidance and identifying the right security and privacy primitives for more secure and reliable IoT products. However, the study has not discussed any security issues related to vulnerabilities at the source code level and possible security risks that might be caused by the usage of unsafe statements.

Although literature is rich with studies that focus on the methods and tools used for detecting vulnerable source code [5], no studies have been conducted specifically in the domain of IoT that evaluate the usage and the distribution of vulnerable source code in IoT software systems and applications prior to the study of which this is an extension.

This work extends a previous work on IoT security in which we conducted an empirical study of unsafe functions on the source code level. We empirically examined 18 systems to determine how many unsafe functions are used and their distributions to help software engineers develop better software systems for IoT and to show how these systems evolve over time in terms of secure programming standards based on the usage of unsafe versus safe replacement functions. We extended the work by examining a larger group of IoT systems, adding 15 more to the original group of software systems studied.

3. Methodology for detecting unsafe functions

A function is considered unsafe if it is one of the functions well-known to both the research community and industry to cause security concerns. Some of those unsafe functions and commands are already banned by compiler producers (e.g., Microsoft). Literature is abundant with lists of unsafe C/C++ commands. We used a tool, UnsafeFunsDetector, developed by one of the main authors, to analyze source code files and, if they contain any unsafe function calls, create a count for each unsafe function. First, we collected all files with C/C++ source-code extensions (i.e., c, cc, cpp, cxx, h, and hpp). For the systems whose history was analyzed, the last version of the system for each year was used. Then, we used the srcML (www.srcML.org) toolkit [1,11] to parse and analyze each file. The srcML format wraps the statements and structures of the source code syntax with XML elements, allowing tools, such as UnsafeFunsDetector, to use XML APIs to locate pieces of code, such as unsafe functions, and to analyze expressions in a quick and efficient manner. Once the system is converted into XML, UnsafeFunsDetector iteratively parses every source code unit to find each call of the unsafe functions and safe replacement functions and adjusts the counters. That is, a count of each unsafe function was recorded. Finally, all calls of unsafe functions were counted and their distributions determined.

Table 1. The 18 studied systems along with the total unsafe functions used in them and the most prevalent unsafe function.

System	Total Unsafe Functions	Most Prevalent Unsafe Function
ApacheMyNewtOS	1,524	memcpy
AtomThreads	62	strlen
Contiki	1,859	memcpy
DistortOS	3	memcpy
Embox	10,286	memcpy
FemtoOS	0	N/A
FreeOSEK	48	memcpy
Lepton	3,928	memcpy
nOS	4	memcpy
OpenTag	58	memcpy
openWSN	220	memcpy
PicoOS	12	free
POK	49	memcpy
TinyOS	772	memcpy
Tneo	1	memcpy
Trampoline	637	free
uOS-Embedded	15,556	puts
Zephyr	3,340	memcpy

The systems chosen in this study were carefully selected to represent a variety of open source systems developed in C/C++ and used for IoT and well-known to both academia and research IoT communities. Our findings are presented and discussed later in this paper, along with the limitations of our approach.

Finally, complete content and organizational editing before formatting. Please take note of the following items when proofreading spelling and grammar.

4. Findings, results and discussion

We now study the usage of unsafe functions in the studied systems and with their distributions, along with the historical trends of their distributions of for 7 of the studied systems.

OpenWSN is an open source project that provides open-source implementations of a complete protocol stack based on Internet of Things standards, for a variety of software and hardware platforms. This implementation can help both academia and industry verify the applicability of these standards to the Internet of Things for those networks to become truly ubiquitous [8].

The TinyOS is an open source, operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, smart buildings, and smart meters [9]. Contiki is a lightweight and flexible operating system for tiny networked sensors [10]. These were the original 3 IoT systems studied in the original paper, but this paper has extended the work

to analyze 18 total systems, along with the most recent 5-year history of 7 of those systems.

4.1. Design of the Empirical Study

This study focuses on three aspects regarding the security of software systems used for IoT in terms of unsafe function and vulnerable code usage. First, we examine the number of calls to known unsafe functions. This gives an idea of how much of the system needs to be refactored to remove or replace the vulnerable code and, therefore, increase its security and quality. Next, we examine which unsafe functions are the most prevalent. This can give the developers of IoT software an idea about the most prevalent unsafe function to they should make a priority, if they plan on refactoring for the purpose of improving their system's security and quality, so that they can increase their system's security in the most efficient way possible. Finally, we examine how the presence of unsafe commands (as opposed to some safer replacements functions) changes over the lifetime of a software system.

The following is our study defined by a set of formal research:

RQ1: What is the total number of unsafe functions called in each system?

RQ2: Of those called, which unsafe functions are the most prevalent?

RQ3: Over the history of a system, is the presence of unsafe functions, and the use of safe replacement functions, increasing or decreasing?

We now examine our findings within the context of these research questions.

4.2. Number of Detected Unsafe Function and their distribution

In the original study, Contiki had the largest number of unsafe functions. In this extension, uOS-Embedded has the largest number of unsafe functions at 15556 total unsafe functions, as opposed to Contiki's 1859 unsafe functions. The smallest number of unsafe functions was 0 for FemtoOS. To address RQ1, Table 1 shows the total number of unsafe functions for each of the 18 systems studied. For the systems whose history was also studied, Table 1 shows the number of unsafe functions of the most recent version. Some of the systems had very few unsafe functions called, while others had thousands of calls to unsafe functions.

In this study, snprintf is considered an unsafe function. Some compliers consider it as a safe replacement function, but it was considered unsafe in this study as it is banned by Microsoft.

For most of the systems, memcpy was the most called unsafe function. In some of the systems, memcpy was the only unsafe function called (e.g., DistortOS, nOS, and Tneo). This trend matches the trend for the original study on only Contiki, TinyOS, and openWSN. Even when a much larger group of IoT systems is studied, mempy remained the most prevalent unsafe function for the majority of the systems, which could allow us to start making better generalizations about the IoT domain when it comes to security than we could when a smaller group was studied.

To address RQ2, Table 2 shows the top 3 most prevalent unsafe functions called across the 18 IoT systems studied. When the top 3 most prevalent for each system was analyzed, memcpy appeared in the top 3 most prevalent functions of 15 systems, strlen appeared in the top 3 most prevalent functions of 10 systems, and strcmp appeared in the top 3 most prevalent functions of 6 systems. The rest of the functions were in the top 3 most prevalent functions of 2 or 3 systems, or they were not present in the top 3 most prevalent functions of any system. The difficulty in generalizing the top 3 most prevalent unsafe functions was that many of the systems varied in the unsafe functions that they called the most, and for some systems, the only unsafe function called was memcpy. While memcpy was the most prevalent unsafe function for the majority of the systems, it was not the most prevalent for all of the systems. The most prevalent for uOS-Embedded was puts, which was not in the top 3 most prevalent unsafe functions for any of the other systems studied.

Table 2. The top three most prevalent unsafe functions across most of the systems

System	memcpy	strcmp	strlen
ApacheMyNewtOS	771	221	220
AtomThreads	12	0	24
Contiki	712	90	343
DistortOS	3	0	0
Embox	2,363	1,726	1,720
FemtoOS	0	0	0
FreeOSEK	17	10	9
Lepton	766	398	662
nOS	4	0	0
OpenTag	42	0	0
openWSN	211	0	2
PicoOS	0	0	0
POK	28	4	6
TinyOS	236	133	94
Tneo	1	0	0
Trampoline	78	40	135
uOS-Embedded	1,500	2,016	1,790
Zephyr	1,778	404	616

Clearly, there is still a noticeably presence of well-known unsafe functions in most of the studied systems, which complicates security concerns when it comes to systems used for IoT. But no matter how we present the data, it is apparent that unsafe functions present one of the most serious security issues that need to be addressed through refactoring systems to remove the unsafe functions. While literature is rich with studies focusing on addressing the problems of how to remove those unsafe functions from software systems, it appears that software developers are underestimating, or lack understanding of, the real threats imposed by the use of unsafe functions.

5. Historical Change of unsafe function frequency

In order to address RQ3, we looked at the most recent 5-year history for 4 additional systems to the original 3 studied. We examined the most recent version of each year and recorded the number of unsafe functions used, along with the number of safe replacement functions used. We are interested in knowing whether

the number of unsafe functions is increasing over time and the security risks are becoming more prevalent, or if the number is

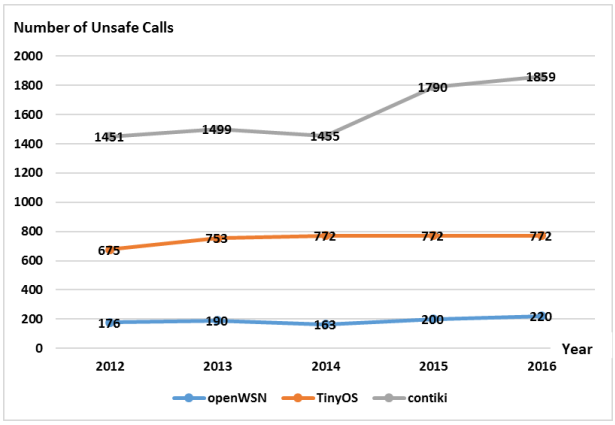

Figure. 1. The change in use of unsafe functions over time for the original three systems studied previously.

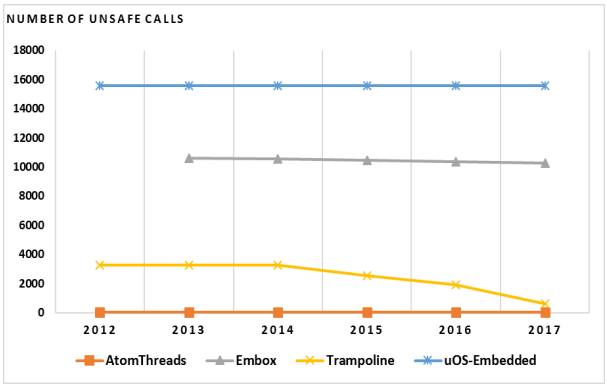

Figure. 2. The change in use of unsafe functions for the additional four systems studied.

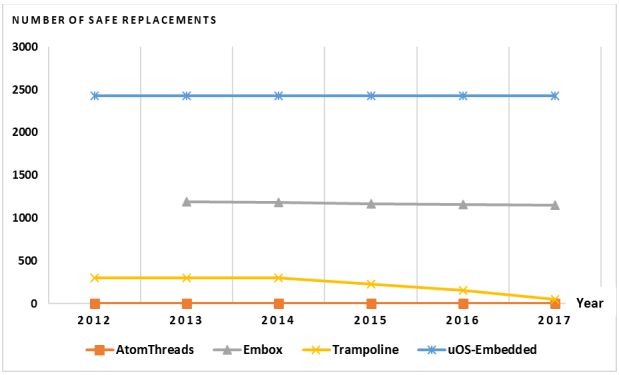

Figure. 3. The change in use of safe alternatives for the four additional systems studied.

lowering, possibly that the developers worked on removing the unsafe functions to increase the security and quality of their system. Figure. 1. shows the change in number of unsafe functions used for the original study [1]. Figure. 2. shows the change in number of unsafe functions for the 4 additional systems studied.

It can be seen that in the original 3 systems, the trends for two systems were relatively flat, while the trend for one system showed an increase in the use of unsafe functions over the 5-year period. We can also see in Figure. 2. that in the 4 additional systems, 3 of them showed very flat trends with 2 of those not having any change in the number of unsafe functions or safe replacements, shown in Figure. 3., beyond the first few years, over the 5-year period. The system, Trampoline, that did not show a flat trend instead showed a decreasing trend in the use of both unsafe functions and safe replacement functions. In 2012, Trampoline had 3265 unsafe functions. In 2017, that number had decreased to 637, although there had been a flat trend for the years 2012-2014.

While most of the systems don't show an increasing trend, as in they are not becoming more vulnerable to security risks, they are also not showing a decreasing trend, which can be seen as equally bad. The presence of unsafe functions remains the same, and the developers are not removing those functions in order to increase the security and quality of their systems. This could be that the developers of those systems are unaware of the risks imposed by the use of well-known unsafe functions, which leads to the need for better education on the topic of software security at the source code level to help those developers create higher quality systems.

6. Threats to Validity

The tools we developed and used for this study only work with languages supported by srcML (C/C++). Because of this limitation, we were unable to include systems written in languages such as Python and Java for this study.

Another limitation is that the tool we used is unable to differentiate between unsafe functions used in dead code, which means that some of the unsafe functions counted may never be called during the system's runtime. This might affect the accuracy of the results we present in terms of the systems' security and vulnerability. Additionally, the calls to unsafe functions that are included in wrappers are not excluded.

7. Future Enhancement

In the future, we are planning to improve the tool so that it only includes active code and exclude the calls to the unsafe functions that are protected with wrappers. We would also like to be able to include more systems written in different programming languages.

In this study, all calls to unsafe functions were counted including regardless the potential wrappers. We are planning to improve the tool so that it excludes the calls that are protected by proper wrappers.

8. Conclusion

This study empirically examined the usage of known unsafe functions and commands in eighteen open source software systems. The systems are all IoT applications written in C/C++ specifically for IoT architectures. There are no other studies of this type currently in the literature. The results show that usage of vulnerable functions is still common for most of the systems, although some systems had very few or even no calls to unsafe functions. Of the eighteen systems studied, memcpy was the most prevalent for the majority of the systems followed by strlen, free and strcmp. The historical trend, for selected systems, shows that developers are not working to improve a problem that still exists.

The vast majority of literature, concerning IoT security, focused on the security issues in the communication layer rather than vulnerabilities at the source code level. As such, more attention needs to be placed on dealing with source vulnerability,

reduce the usage of unsafe statements, especially the most prevalent statements to improve IoT platforms in terms of security, and educate developers on ways to both refactor their systems and to avoid the use of unsafe functions from the beginning when writing code, thus enhancing performance.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

This work was supported in part by a grant from the US National Science Foundation (NSF) Grant no. 1542368.

References

[1] S. M. Alnaeli, M. Sarnowski, M. S. Aman, A. Abdelgawad, and K. Yelamarthi, "Vulnerable C/C++ code usage in IoT software systems," in 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), 2016, pp. 348-352.

[2] H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the Internet of Things: A Review," in Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on, 2012, pp. 648-651.

[3] C. Kolias, A. Stavrou, J. Voas, I. Bojanova, and R. Kuhn, "Learning Internet-of-Things Security; 'Hands-On'," IEEE Security & Privacy, vol. 14, pp. 37-46, 2016.

[4] A. M. Gamundani, "An impact review on internet of things attacks," in Emerging Trends in Networks and Computer Communications (ETNCC), 2015 International Conference on, 2015, pp. 114-118.

[5] 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.

[6] R. K. McLean, "Comparing Static Security Analysis Tools Using Open Source Software," in Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on, 2012, pp. 68-74.

[7] Veracode, "The Internet of Things: Security Research Study". (2015). https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf

[8] The University of California. (2016). openWSN https://openwsn.atlassian.net/wiki/pages/viewpage.action?pageId=688187

[9] The TinyOS Working Group, (2013), TinyOS. http://www.tinyos.net/

[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors" in 29th Annual IEEE International Conference on Local Computer Networks, 2004.

[11] S. M. Alnaeli, A. A. Taha, and T. Timm, "On the Prevalence of Function Side Effects in General Purpose Open Source Software Systems," in Software Engineering Research, Management and Applications, R. Lee, Ed., ed Cham: Springer International Publishing, 2016, pp. 115-131.