BILINGUAL PUBLISHING GROUP

Journal of Computer Science Research

https://journals.bilpubgroup.com/index.php/jcsr

ARTICLE

Detection of Buffer Overflow Attacks with Memoization-based Rule Set

***Oğuz Özger*[1], *Halit ÖztekİIn*[2*]**

[1] *Electrical-Electronics Engineering, Sakarya University of Applied Sciences, Sakarya, 54050, Turkey*

[2] *Computer Engineering, Sakarya University of Applied Sciences, Sakarya, 54050, Turkey*

ABSTRACT

Different abnormalities are commonly encountered in computer network systems. These types of abnormalities can lead to critical data losses or unauthorized access in the systems. Buffer overflow anomaly is a prominent issue among these abnormalities, posing a serious threat to network security. The primary objective of this study is to identify the potential risks of buffer overflow that can be caused by functions frequently used in the PHP programming language and to provide solutions to minimize these risks. Static code analyzers are used to detect security vulnerabilities, among which SonarQube stands out with its extensive library, flexible customization options, and reliability in the industry. In this context, a customized rule set aimed at automatically detecting buffer overflows has been developed on the SonarQube platform. The memoization optimization technique used while creating the customized rule set enhances the speed and efficiency of the code analysis process. As a result, the code analysis process is not repeatedly run for code snippets that have been analyzed before, significantly reducing processing time and resource utilization. In this study, a memoization-based rule set was utilized to detect critical security vulnerabilities that could lead to buffer overflow in source codes written in the PHP programming language. Thus, the analysis process is not repeatedly run for code snippets that have been analyzed before, leading to a significant reduction in processing time and resource utilization. In a case study conducted to assess the effectiveness of this method, a significant decrease in the source code analysis time was observed.

Keywords: Buffer overflow; Cybersecurity; Anomaly; SonarQube; Memoization

*CORRESPONDING AUTHOR:

Halit ÖztekİIn, Computer Engineering, Sakarya University of Applied Sciences, Sakarya, 54050, Turkey; Email: halitoztekin@subu.edu.tr

ARTICLE INFO

Received: 28 October 2023 | Revised: 15 November 2023 | Accepted: 22 November 2023 | Published Online: 30 November 2023

DOI: https://doi.org/10.30564/jcsr.v5i4.6044

CITATION

Özger, O., ÖztekİIn, H., 2023. Detection of Buffer Overflow Attacks with Memoization-based Rule Set. Journal of Computer Science Research. 5(4): 13-26. DOI: https://doi.org/10.30564/jcsr.v5i4.6044

1. Introduction

Internet networks are an indispensable component that facilitates information exchange between individuals and organizations; however, this infrastructure needs to be protected against threats such as network anomalies. One of these dangers is buffer overflows. Buffer overflow can lead to an attack that allows malicious codes to be executed on the target system. Therefore, the detection and prevention of buffer overflow attacks are of great importance in ensuring the security of computer systems. The attacks and impacts caused by buffer overflow errors hold a significant place in the history of technology. The Morris worm, one of the first major attacks in the history of the internet, originated from a buffer overflow error [1]. The worm named Code Red [2] on the other hand, affected millions of computers due to a buffer overflow error in Microsoft IIS servers, displaying a widespread distribution over the internet. The Heartbleed vulnerability, which emerged in the OpenSSL cryptography library, put at risk the SSL encryption protocol used across a large part of the internet [3]. Another incident stemming from a buffer overflow error is the attack on Equifax, a credit report service provider [4]. In addition, the security vulnerabilities named Spectre and Meltdown in Intel processors [5], the BlueKeep vulnerability in the Windows operating system [6], the attack on the popular messaging application WhatsApp [7] and the attack on the Microsoft Exchange email server [8] all share a common ground originating from structures susceptible to buffer overflow.

It is possible to protect against buffer overflow attacks by taking a series of precautionary measures. These include controlling the array size [9], using secure input/output functions, memory limiting [10], Address Space Layout Randomization (ASLR), stack canaries, software updates, conducting security testing, fuzzing, and sandboxing [11]. Thanks to these protection methods, many systems are known to have become more resilient [12,13]. Additionally, hardware-based measures are stronger compared to software-based measures, as they are based on the working logic of the hardware and are harder to override or bypass by software. However, these measures are only supported by modern CPUs and cannot be used in older systems. Hardware-based memory protection technologies are widely supported, particularly in modern processor architectures, such as Intel (VT-x, VT-d, etc.) and AMD (V, Vi, etc.) [14].

Hardware-based measures yield the most effective results when implemented alongside software-based solutions. Various software tools, such as dynamic and static code analysis tools, memory tracing utilities, and fuzzers, can be utilized to detect such types of cyber threats. Particularly, static code analysis is instrumental in identifying potential security weaknesses by conducting an analysis of the code without the necessity to execute it. In this realm, static code analysis tools like SonarQube offer distinctive advantages in the early detection of potential security vulnerabilities within source codes. The objective of this study is to ascertain the potential risks of buffer overflow incidents precipitated by functions commonly employed in the PHP programming language and to propose methodologies to mitigate these risks. The prominence of PHP as one of the globally most-utilized scripting languages today, coupled with the reality that a significant proportion of security vulnerabilities identified in web-based computer software in recent times are associated with PHP, underpins the rationale for selecting this language for our investigation. Static code analyzers are deployed to pinpoint security vulnerabilities, among which SonarQube is notable for its comprehensive library, adaptable customization options, and its industry-trusted reliability. Within this framework, a specialized, memoization-based rule set designed for the automatic detection of buffer overflows has been developed on the SonarQube platform.

This customized rule set, through extensive analysis executed on PHP source codes, can preemptively identify memory management faults, particularly dangerous function calls, and erroneous array operations. This facilitates the proactive correction of security vulnerabilities in the earlier stages of the development cycle. Integrated into SonarQube's rich plugin ecosystem, this rule set offers in-depth guid-

ance to developers, security analysis experts, and information security teams on fortifying PHP codes. Furthermore, the rule set can be calibrated to recognize specific security weaknesses prevalent in widely used PHP frameworks and libraries, significantly bolstering the security posture of PHP applications in the industry and elevating the standards of web application security.

2. Literature review

Aleph One [15] has presented a detailed report on the causes and effects of buffer overflow attacks, which hold a significant place among network anomalies, while Wagner and colleagues have addressed this threat with various protection methods and detection approaches [16]. Protection methods like StackGuard [17] and RAD [18] stand out as prominent approaches in the literature. Kuperman and colleagues have proposed detecting buffer overflow by combining static and dynamic analyses [19]. Le and Soffa have detected these attacks with demand-based and path-sensitive analysis [20]. Brooks has evaluated the effects of automatic vulnerability detection and exploitation techniques on buffer overflow [21]. A report [22] highlighting the critical importance of static analysis techniques for software security has been presented, and accordingly, tools like ARCHER [23] and Safe-C [24] introduced in the literature automatically detect memory access errors.

To detect and prevent memory errors at runtime, Valgrind performs dynamic program inspection to monitor potential memory errors [25], and Rinard and colleagues have conducted studies in this field using dynamic methods [26]. Additionally, FormatGuard [27] has been developed by Fen and colleagues [28], and Ruwase and Lam [29] have devised protection methods against memory overflow errors.

Buffer overflow attacks stem from memory management errors in software, providing attackers the opportunity to exploit these flaws for the execution of malicious code. Jha has conducted research on methods to close security vulnerabilities in software and enhance its security [30]. Emami, Ghiya, and Hendren have focused on the potential security challenges posed by the use of function pointers in C [31]. Liang and Sekar have generated automatic signatures against buffer overflow attacks using symbolic modeling [32]. In the same context, Newsome and Song have detected malware attacks using dynamic taint analysis [33]. Memory errors can pose a risk to the reliability of software. Qin and colleagues have introduced a method of monitoring ECC memory to detect these errors [34]. At the same time, Seward and Nethercote have outlined methods for detecting undefined value errors using the Valgrind tool [35]. Executable memory protection for Linux is a critical defence against attacks, and a comprehensive overview of this subject is provided on Wikipedia [36].

Nethercote and Seward have introduced methods for detecting software bugs by tracking dynamic features through the Valgrind framework [37]. Costa and colleagues have worked on enhancing software security by conducting dynamic input verification with Bouncer [38]. Song and his team have presented the BitBlaze method for the analysis of malicious software [39]. Liu and his colleagues have identified vulnerabilities in x86 programs using obfuscation methods and genetic algorithms [40]. Kroes and his team have provided automatic detection methods for memory management errors using Delta pointers [41]. Frantzen and Shuey have introduced hardware-assisted stack protection with StackGhost [42]. Novark and Berger have presented dynamic memory management approaches with the DieHarder tool [43]. Sayeed and colleagues have proposed protection against buffer overflow attacks through control flow integrity [44]. Andriesse and his team have offered methods for software integrity protection and blocking malicious code with Parallax [45].

In recent years, machine learning has emerged as a method used for attack detection from network traffic. Mukkamala and colleagues have conducted studies in this field [46], while Thottan and Ji have performed anomaly detection by examining the characteristics of network traffic data [47].

For the detection of attacks, tools such as dynamic and static code analysis tools, memory tracing tools, and fuzzers can be utilized. In particular, static code

analysis helps identify potential security vulnerabilities through the method of analyzing the code without executing it. To detect security vulnerabilities in software, Cova and colleagues have presented methods combining flow graphs and static analysis [48]. Lanzi and colleagues have also proposed a vulnerability detection tool for the x86 architecture [49]. At the same time, Miller and colleagues have tested the resilience of applications for MacOS X [50]. In this context, static code analysis tools like SonarQube possess unique advantages in detecting potential security vulnerabilities in software at an early stage. Its multi-language support for various programming languages allows for language-independent analysis of projects. Its customizability feature allows for the addition of specific rules. It can incorporate customized rule creation in SonarQube as well as the integration of an optimization algorithm with machine learning.

3. Speed up static code analysis

WERTYMemoization is an optimization technique used to store the results of computationally expensive operations, preventing the need for repeated execution of the same operations. This method is particularly employed in dynamic programming problems to minimize redundant calculations. In the fields of data mining and big data analysis, memoization contributes to resolving the time issues encountered when algorithms process large data sets. A HashMap function can be created for the source code, storing information about previously seen function names and whether these functions produce a "buffer overflow". Consequently, when a function listed is encountered, the code does not have to recheck if the function produces a "buffer overflow"; instead, it quickly retrieves the result from the HashMap function.

Static code analysis tools aim to identify errors, assess code quality, and pinpoint security vulnerabilities in codes written in various programming languages. However, in large-scale projects, completing each analysis can take a considerable amount of time. The integration of the memoization technique into static code analysis is targeted to quickly analyze repeating functions or method calls, thereby reducing the total analysis time. A functional comparison of prominent and widely accepted static code analysis tools in the literature is provided in **Table 1**.

The static code analysis tool SonarQube stands out among other analysis tools due to its support for numerous programming languages, visual reporting and user-friendly interface, offering broader customization options, and having a large user and developer community.

Table 1. Functional comparison of code analysis tools.

Feature/Criterion	**SonarQube [51]**	**ESLint**	**Checkmzrx**	**Coverity**
Supported languages	20+ (Java, C#, PHP, JS vb.)	Firstly, JS ve TypeScript	20+	20+
Web based interface	Yes	No (CLI tabanlı)	Yes	No
CI/CD integration	Extensive (Jenkins, Travis, Azure Pipelines vb.)	Limited	Extensive	Extensive
Customizable rules	Yes	Yes	Yes	Yes
Community support	Strong	Strong	Medium	Medium
Code quality metrics	Yes	No	No	No
Licensing and cost	Free and commercial versions	Free	Commercial	Commercial
Multi-language project support	Yes	No	Yes	Yes
Plugin and extension support	Yes (Numerous plugins available on the Marketplace)	Yes (npm packages)	Limited	Limited

4. Case study

Methods used for automatic detection of security vulnerabilities causing buffer overflow in static code analysis are presented in this section. The interface named Sonar-Scanner, which can work integrated with SonarQube, enables the visualization of analysis results and the execution of management operations. On the SonarQube platform, a special security rule has been established with the aim of detecting PHP functions that can create a security vulnerability (buffer overflow).

4.1 Memoization-based custom rule integration

The steps for incorporating the memoization principle into the rule are given below. Integration of the rule with the SonarQube analysis tool not only reduces analysis time, but also enables quicker detection of critical security risks.

Step-1: Creating a custom security rule: Leveraging the customizable structure of SonarQube analysis tool for PHP, a security rule incorporating the memoization technique is established.

Step-2: Cache management: Implementing a caching mechanism within the custom rule to store the results of functions that pose a security vulnerability risk.

Step-3: Risk analysis: Evaluating potential security risks when functions or methods are called for the first time, and storing the result in the cache.

Step-4: Utilizing the cache: When a function of the same name is called again, use the information in the cache to instantly perform a security assessment.

In the first step, key files and functions for the specific rules determined for static code analysis are utilized. The file named BufferOverflowCheck.java examines function calls, checking for the usage of specific functions such as strcpy, strcat, and fwrite. When the use of these functions is detected, a security alert is generated. The MyPhpRules.java file stores the list of existing custom rules, and these rules are added to the SonarQube rule repository. The PHPCustomRulesPlugin.java file defines the Sonar Plugin and adds the MyPhpRules class. The pom.xml is used as a Maven configuration file for building the project and managing its dependencies. The necessary modules for creating a custom rule in the SonarQube analysis tool are provided in **Figure 1**.

As shown in **Figure 2**, the SonarQube tool uses the "@Rule" annotation to customize rule definitions. There are different priority levels in these checks, which are listed as INFO, MINOR, MAJOR, CRITICAL, and BLOCKER. For the rule created

```
package org.sonar.samples.php.checks;

import org.sonar.check.Priority;
import org.sonar.check.Rule;
import org.sonar.plugins.php.api.tree.expression.FunctionCallTree;
import org.sonar.plugins.php.api.visitors.PHPVisitorCheck;

import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.nio.file.*;
import java.io.IOException;

import java.io.BufferedWriter;
import java.io.FileWriter;
```

Figure 1. Loading of modules.

```
@Rule(
  key = BufferOverflowCheck.KEY,
  priority = Priority.CRITICAL,
  name = "Buffer overflow issues should be fixed.",
  tags = {"security"}
)
```

Figure 2. Determination of rule level.

in this study, the CRITICAL priority level has been selected because buffer overflow can lead to security breaches, and such a violation can pose a critical threat to the entire system.

As shown in **Figure 3**, the constructor (Buffer OverflowCheck()) creates a HashMap named functionResultsCache. This structure is later used to quickly detect risky functions. By calling the loadCache() function, a previously created cache is loaded.

As illustrated in **Figure 4**, the loadCache() function reads the previously saved cache data from the disk and places it in the HashMap. Each line read from the disk contains a function name and a boolean value. The function name serves as the key, and the boolean value is processed into the map.

In every situation where the cache is updated, the saveCache() function is invoked as depicted in **Figure 5**. This function saves the functionResultsCache HashMap to a file.

As illustrated in **Figure 6**, the visitFunctionCall() function is triggered for each function call in the source code. After retrieving the function name, it is checked whether it has been previously added to the cache or not.

If the function name exists in the cache, the stored boolean value is used. If the value is true, a "Buffer overflow issue detected" warning is generated. If the function name has not been seen before, it is checked whether it is risky. If it is in the list of risky functions (such as strcpy, strcat, gets, etc.), it is added to the cache as true and subsequently a warning is generated. When the cache is updated, the saveCache() function is called, and the process continues for other potential SonarQube tool inspections with super. visitFunctionCall(tree) (**Figure 7**).

```
public class BufferOverflowCheck extends PHPVisitorCheck {

  public static final String KEY = "S3";
  private final Map<String, Boolean> functionResultsCache;

  public BufferOverflowCheck() {
    functionResultsCache = new HashMap<>();
    loadCache();
  }
```

Figure 3. Creation the BufferOverflowCheck class.

```
  private void loadCache() {
    try {
      List<String> lines = Files.readAllLines(Paths.get("/cache_dosyasi/cache.txt"));
      for (String line : lines) {
        String[] parts = line.split(",");
        functionResultsCache.put(parts[0], Boolean.parseBoolean(parts[1]));
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Figure 4. LoadCahce function.

```
  private void saveCache() {
    try {
      List<String> lines = new ArrayList<>();
      for (Map.Entry<String, Boolean> entry : functionResultsCache.entrySet()) {
        lines.add(entry.getKey() + "," + entry.getValue());
      }
      Files.write(Paths.get("/cache_dosyasi/cache.txt"), lines);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

Figure 5. SaveCahce function.

```
@Override
public void visitFunctionCall(FunctionCallTree tree) {
  String functionName = tree.callee().toString();

  if (functionResultsCache.containsKey(functionName)) {
    if (functionResultsCache.get(functionName)) {
      context().newIssue(this, tree, "Buffer overflow issue detected.");
    }
    return; // Cache'den bilgi alındı, işlemi sonlandır
  }
```

Figure 6. visitFunctionCall function.

```
  boolean isOverflow = functionName.equals("strcpy") ||
                       functionName.equals("strcat") ||
                       functionName.equals("strncpy") ||
                       functionName.equals("sprintf") ||
                       functionName.equals("unpack") ||
                       functionName.equals("addslashes") ||
                       functionName.equals("fread") ||
                       functionName.equals("fwrite") ||
                       functionName.equals("strtr") ||
                       functionName.equals("gzuncompress") ||
                       functionName.equals("getimagesize") ||
                       functionName.equals("iptcembed") ||
                       functionName.equals("mb_parse_str") ||
                       functionName.equals("move_uploaded_file") ||
                       functionName.equals("array_splice");

  functionResultsCache.put(functionName, isOverflow);

  if (isOverflow) {
    context().newIssue(this, tree, "Buffer overflow issue detected.");
  }

  saveCache(); // Cache değiştiği için kaydet

  super.visitFunctionCall(tree);
 }
}
```

Figure 7. Identifying functions that cause buffer overflow.

4.2 Memoization-based rule for attack detection

This structure also employs a caching mechanism to enhance performance. Specifically, the saveCache and loadCache methods are used for loading the cache file from the disk and saving it back to the disk. This custom rule can effectively detect buffer overflow issues in PHP projects, assisting in the prevention of such security vulnerabilities. The workflow of the rule added to the Sonar Qube analysis tool is provided in the steps below.

Step-1: The BufferOverflowCheck class stores function names and whether these functions may cause a buffer overflow in a HashMap.

Step-2: When any function call is seen in the PHP code, the visitFunctionCall function is triggered by the rule.

Step-3: If the function name is in the cache (HashMap), the rule can immediately generate a warning. Otherwise, the function name and whether it creates a buffer overflow is added to the cache.

In the example application below, a custom rule has been added to the SonarQube analysis tool to detect buffer overflow attacks. When dangerous PHP functions (such as strncpy, strcat, addslashes, fwrite, array_splice, etc.) are called, the application issues a security vulnerability warning (**Figure 8**).

```
        printf("<form class='frmsource' method='post'>
                        <textarea id='sourcefocus' name='sourcecode' rows='25' cols='100'>%s</textarea>
                        <input type='Submit' value='Save file' name='save'/>
                        <label>%s</label>
                </form>",$source,$status);

        if(any("status",$_SESSION)) unset($_SESSION['status']);

        if(any("save",$_REQUEST))
        {
                $new_source=$_REQUEST['sourcecode'];
                if(function_exists("chmod")) chmod($file,0755);
                $source_edit=fopen($file,'w+');
                $tulis=fwrite($source_edit,$new_source);
                fclose($source_edit);
                if($tulis)
                {
                        $_SESSION['status']="File Saved ! ".GetFileTime($file,"modify")." | ".GetFileSize(filesize($file));
                }
                else
                {
                        $_SESSION['status']="Whoops, something went wrong...";
                }
                header("location:".php_self."?a=e&r=".urle($file));
        }
}

if($_REQUEST['a']=='r')
{
        printf("<form class='new' method='post'>
                <input type='text' name='name' value='%s'/>
                <input type='Submit' value='Rename' name='rename'/>
                <label>%s</label>
        </form>",basename($file),$status);

        if(any("status",$_SESSION)) unset($_SESSION['status']);
```

Figure 8. Code block in bat.php file where the “fwrite” function is used.

Additionally, the effectiveness of the memoization method has been measured by applying the custom rule created to a PHP code found in the bat.php repository located in the “b4tm4n” repository on GitHub [45]. The source code selected for analysis consists of a total of 3962 lines. After the SonarQube analysis tool successfully analyzes the code, it displays the identified risky functions and security vulnerabilities on the interface. **Figure 9** shows the results of the analysis summary of the bat.php file available on Github.

Figure 10 shows the lines of code in the bat.php file that are potentially vulnerable to buffer overflow attacks.

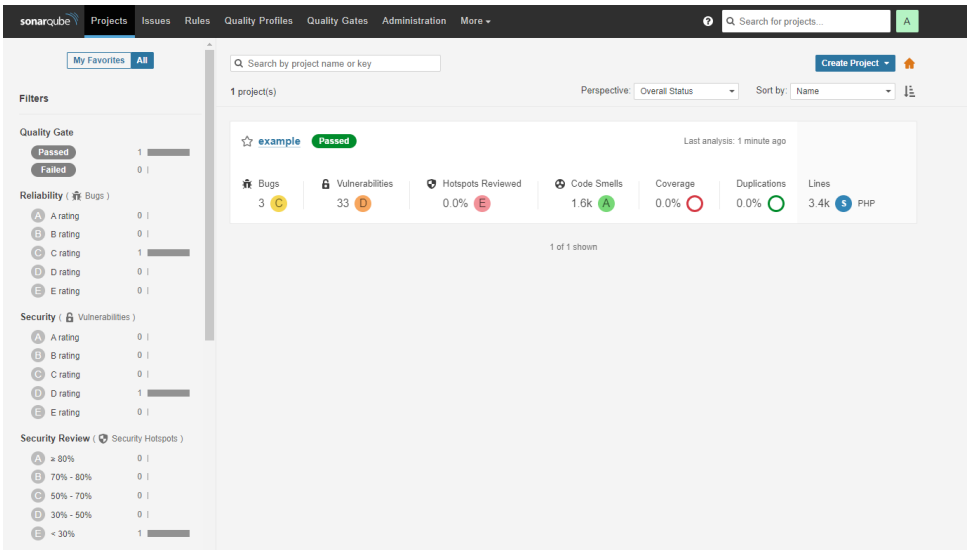

Figure 9. Analysis summary results of the bat.php file located on Github.

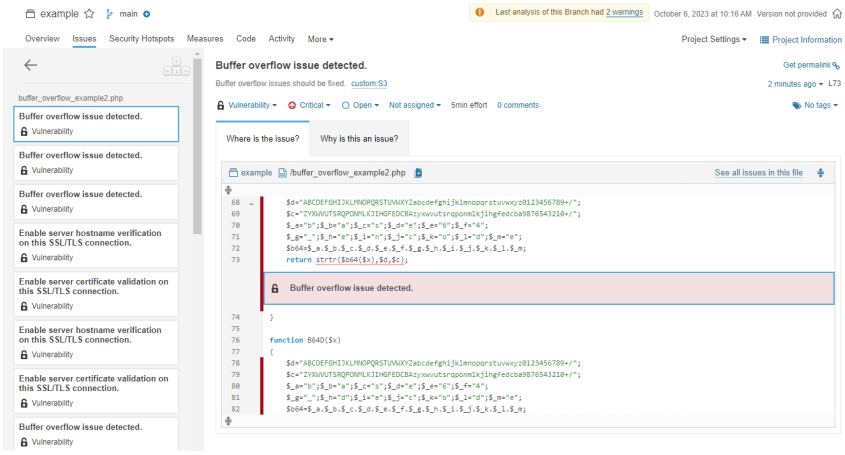

Figure 10. Code lines in the bat.php file that are potentially vulnerable to buffer overflow attacks.

During the analysis with the custom rule created, the information of the dangerous and safe functions detected thanks to the loadCache function is saved to a file named cache.txt. An example of a cache.txt file is shown below:

As shown in **Figure 11**, there are functions marked as true or false. Here, true represents the functions that may pose a security vulnerability risk, while false represents the functions that will not create a security vulnerability. The existence of the cache.txt file significantly increases the performance during the next run of the custom rule. The loadCache function reads this file at the beginning of the analysis and loads the function information into memory (RAM). Thus, when re-analyzing the same code, instead of analyzing the security risks of the same functions again, the custom rule directly retrieves the information from the memory. This is an optimization technique known as memoization. This approach shortens the analysis time and can quickly identify previously detected security vulnerabilities in each new analysis. **Figures 12-14** show the analysis times of the lines in the bat.php file analyzed above against buffer overflow attacks. While the total analysis time is 29.118 s when the cache is empty, it is 22.048 s when the cache is full. It can be seen that the use of a cache significantly reduces the analysis time.

```
sprintf,true
set_error_handler,false
strlen,false
filter_var,false
http_build_query,false
move_uploaded_file,true
GetUser,false
```

Figure 11. Sample content is taken from the cache.txt file.

```
private void loadCache() {
  try {
    List<String> lines = Files.readAllLines(Paths.get("/cache_dosyasi/cache.txt"));
    for (String line : lines) {
      String[] parts = line.split(",");
      functionResultsCache.put(parts[0], Boolean.parseBoolean(parts[1]));
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Figure 12. Memoization.

```
INFO: Analysis total time: 19.351 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 22.048s
INFO: Final Memory: 23M/87M
INFO: ------------------------------------------------------------------------
```

Figure 13. Analysis duration with cache usage.

```
INFO: Analysis total time: 26.699 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 29.118s
INFO: Final Memory: 23M/87M
INFO: ------------------------------------------------------------------------
```

Figure 14. Analysis duration without cache usage.

5. Results and discussion

In this study, we examine how a customized rule can be added to the SonarQube analysis tool to identify potential buffer overflow security vulnerabilities in codes written in PHP. The main goal is to develop a system that not only performs static code analysis but also enhances analysis performance with a caching mechanism. Specific functions that could pose a risk in PHP codes have been identified, and a custom SonarQube rule has been designed for these functions. The analysis process has been optimized using the memoization technique, reducing repeated analyses on the same code. This approach saves time and resources, particularly for large codebases.

The current work also records significant advancements in the effective detection of commonly used functions that carry the risk of buffer overflow. The increase in analysis performance has been made possible by the caching mechanism. The flexibility and extensibility of this custom rule mean it can be applied to different functions and methods. However, the study has limitations, such as being specific to the PHP language and covering only certain functions. While the initial version is capable of static code analysis only, the integration of dynamic analysis and adaptation to different programming languages represent significant potential for future work.

This research makes a notable contribution to the field of web security, especially regarding the security of PHP applications. In the future, the automatic detection and resolution of such security vulnerabilities are expected to facilitate the creation of more effective and secure applications in software development and cybersecurity. Particularly, the integration of the custom rule with dynamic analysis, the incorporation of machine learning techniques, and improvements to the user interface will push the innovations in this field further. In addition, advanced automation approaches for strengthening defence mechanisms against security vulnerabilities and making software development processes more robust could be developed, contributing to risk assessment and management strategies. Designing an extended security framework applicable in various software languages to deal with a wide spectrum of security threats, beyond specific challenges such as buffer overflow, will be an important goal for future researchers and applications. Such a framework will play a crucial role in protecting critical systems and securing sensitive data.

Author Contributions

Halit Oztekin and Oguz Ozger—conceptualization, methodology, formal analysis, investigation, supervision, validation, visualization, writing—original draft, and writing—review and editing.

Conflict of Interest

The author declares that there are no conflicts of interest.

Funding

This research received no external funding.

Acknowledgement

There is no acknowledgement for this article.

References

[1] Spafford, E.H., 1989. The Internet worm program: An analysis. ACM SIGCOMM Computer Communication Review. 19(1), 17-57. DOI: https://doi.org/10.1145/66093.66095

[2] Moore, D., Paxson, V., Savage, S., et al., 2003.

Inside the slammer worm. IEEE Security & Privacy. 1(4), 33-39.
DOI: https://doi.org/10.1109/MSECP.2003.1219056

[3] Springall, D., Durumeric, Z., Halderman, J.A. (editors), 2016. Measuring the security harm of TLS crypto shortcuts. IMC'16: Proceedings of the 2016 Internet Measurement Conference; 2016 Nov 14-16; Santa Monica California USA. New York: Association for Computing Machinery. p. 33-47.
DOI: https://doi.org/10.1145/2987443.2987480

[4] Oversight and Government Reform [Internet]. Available from: https://oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf

[5] Kocher, P., Horn, J., Fogh, A., et al. (editors), 2019. Spectre attacks: Exploiting speculative execution. 2019 IEEE Symposium on Security and Privacy (SP); 2019 May 19-23; San Francisco, CA, USA. New York: IEEE.
DOI: https://doi.org/10.1109/SP.2019.00002

[6] Remote Desktop Services Remote Code Execution Vulnerability [Internet]. Available from: https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2019-0708

[7] Bhutan Built a Bitcoin Mine on the Site of Its Failed 'Education City'; 2019.

[8] On-Premises Exchange Server Vulnerabilities Resource Center—updated March 25, 2021 [Internet]. Available from: https://msrc.microsoft.com/blog/2021/03/multiple-security-updates-released-for-exchange-server/

[9] Dowd, M., McDonald, J., Schuh, J., 2006. The art of software security assessment: Identifying and preventing software vulnerabilities. Pearson Education: London.

[10] SEI CERT Oracle Coding Standard for Java [Internet]. Available from: https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

[11] Conklin, W.A., White, G., Cothren, C., et al., 2022. Principles of Computer Security: CompTIA Security+™ and Beyond [Internet]. Available from: https://sisis.rz.htw-berlin.de/inh2010/12389366.pdf

[12] Xu, J., Patel, S., Iyer, R., et al., 2002. Architecture Support for Defending Against Buffer Overflow Attacks [Internet]. Available from: https://www.ideals.illinois.edu/items/100089/bitstreams/319746/data.pdf

[13] Anley, C., Heasman, J., Lindner, F., et al., 2007. The shellcoder's handbook: Discovering and exploiting security holes. Wiley: Hoboken.

[14] Intel® 64 and IA-32 Architectures Software Developer Manuals [Internet]. Available from: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[15] One, A., 1996. Smashing the stack for fun and profit. Phrack Magazine. 7(49), 14-16.

[16] Wagner, D., Foster, J.S., Brewer, E.A., et al., 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities [Internet]. Available from: https://www.cs.umd.edu/class/spring2021/cmsc614/papers/automated-buffer.pdf

[17] Baratloo, A., Singh, N., Tsai, T. (editors), 2000. Transparent run-time defense against stack smashing attacks. Proceedings of the 2000 USENIX Annual Technical Conference; 2000 Jun 18-23; San Diego, California, USA.

[18] Chiueh, T.C., Hsu, F.H. (editors), 2001. RAD: A compile-time solution to buffer overflow attacks. Proceedings of the 21st International Conference on Distributed Computing Systems; 2001 Apr 16-19; Mesa, AZ, USA. New York: IEEE. p. 409-417.
DOI: https://doi.org/10.1109/ICDSC.2001.918971

[19] Kuperman, B., Brodley, C., Ozdoganoglu, H., et al., 2005. Detection and prevention of stack buffer overflow attacks. Communications of the ACM. 48(11), 50-56.
DOI: https://doi.org/10.1145/1096000.1096004

[20] Le, W., Soffa, M.L. (editors), 2008. Marple: A demand-driven path-sensitive buffer overflow detector. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering; 2008 Nov 9-14; Atlanta Georgia. New York: Association for Computing Machin-

ery. p. 272-282.
DOI: https://doi.org/10.1145/1453101.1453137

[21] Brooks, T.N., 2017. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. Advances in intelligent systems and computing. Springer: Cham. pp. 1083-1102.
DOI: https://doi.org/10.1007/978-3-030-01177-2_79

[22] Chess, W., 1998. Secure programming with static analysis. Pearson Education: London.

[23] Cowan, C., Beattie, S., Johansen, J., et al. (editors), 2003. Pointguard TM: Protecting pointers from buffer overflow vulnerabilities. Proceedings of the 12th USENIX Security Symposium; 2003 Aug 4-8; Washington D.C., USA.

[24] Yong, S.H., Horwitz, S. (editors), 2003. Protecting C programs from attacks via invalid pointer dereferences. Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2003 Sep 1-5; Helsinki Finland. New York: Association for Computing Machinery. p. 307-316.
DOI: https://doi.org/10.1145/940071.940113

[25] Nethercote, N., Seward, J., 2003. Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science. 89(2), 44-66.
DOI: https://doi.org/10.1016/S1571-0661(04)81042-9

[26] Rinard, M., Cadar, C., Dumitran, D., et al. (editors), 2004. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). 20th Annual Computer Security Applications Conference; 2004 Dec 6-10; Tucson, AZ, USA. New York: IEEE. p. 82-90.
DOI: https://doi.org/10.1109/CSAC.2004.2

[27] Cowan, C., Barringer, M., Beattie, S., et al. (editors), 2001. FormatGuard: Automatic protection from printf format string vulnerabilities. Proceedings of the 10th USENIX Security Symposium; 2001 Aug 13-17; Washington D.C., USA. p. 191-200.

[28] Fen, Y., Fuchao, Y., Xiaobing, S., et al., 2012. A new data randomization method to defend buffer overflow attacks. Physics Procedia. 24, 1757-1764.
DOI: https://doi.org/10.1016/j.phpro.2012.02.259

[29] Ruwase, O., Lam, M.S., 2003. A Practical Dynamic Buffer Overflow Detector [Internet]. Available from: http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/oor/papers/cred.pdf

[30] Jha, S. (editor), 2010. Retrofitting legacy code for security. Computer Aided Verification, 22nd International Conference, CAV 2010; 2010 Jul 15-19; Edinburgh, UK. Berlin: Springer.
DOI: https://doi.org/10.1007/978-3-642-14295-6_2

[31] Emami, M., Ghiya, R., Hendren, L. (editors), 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation; 1994 Jun 20-24; Orlando Florida USA. New York: Association for Computing Machinery.
DOI: https://doi.org/10.1145/178243.178264

[32] Liang, Z., Sekar, R. (editors), 2005. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. 21st Annual Computer Security Applications Conference (ACSAC'05); 2005 Dec 5-9; Tucson, AZ, USA. New York: IEEE. p. 10-224.
DOI: https://doi.org/10.1109/CSAC.2005.12

[33] Newsome, J., Song, D., 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software [Internet]. Available from: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5803709632cf010d3923e8f85416bb95db0dd8ea

[34] Qin, F., Lu, S., Zhou, Y. (editors), 2005. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. 11th International Symposium

on High-Performance Computer Architecture; 2005 Feb 12-16; San Francisco, CA, USA. New York: IEEE. p. 291-302.
DOI: https://doi.org/10.1109/HPCA.2005.29

[35] Seward, J., Nethercote, N. (editors), 2005. Using Valgrind to detect undefined value errors with bit-precision. Proceedings of the USENIX'05 Annual Technical Conference; 2005 Apr 10-15; Anaheim, California, USA.

[36] Executable-space Protection [Internet]. Available from: https://en.wikipedia.org/wiki/Executable-space_protection

[37] Nethercote, N., Seward, J., 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. ACM Sigplan Notices. 42(6), 89-100.
DOI: https://doi.org/10.1145/1273442.1250746

[38] Costa, M. (editor), 2008. Bouncer: Securing software by blocking bad input. Proceedings of the 2nd Workshop on Recent Advances on Intrusiton-tolerant Systems; 2008 Apr 1; Glasgow United Kingdom. New York: Association for Computing Machinery.
DOI: https://doi.org/10.1145/1413901.1413902

[39] Song, D., Brumley, D., Yin, H., et al. (editors), 2008. BitBlaze: A new approach to computer security via binary analysis. Information Systems Security, 4th International Conference, ICISS 2008; 2008 Dec 16-20; Hyderabad, India. Berlin: Springer. p. 1-25.
DOI: https://doi.org/10.1007/978-3-540-89862-7_1

[40] Liu, G.H., Wu, G., Tao, Z., et al. (editors), 2008. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. 2008 Third International Conference on Convergence and Hybrid Information Technology; 2008 Nov 11-13; Busan, Korea (South). New York: IEEE. p. 491-497.
DOI: https://doi.org/10.1109/ICCIT.2008.9

[41] Kroes, T., Koning, K., Kouwe, E., et al. (editors), 2018. Delta pointers: Buffer overflow checks without the checks. EuroSys'18: Proceedings of the Thirteenth EuroSys Conference; 2018 Apr 23-26; Porto Portugal. New York: Association for Computing Machinery. p. 1-14.
DOI: https://doi.org/10.1145/3190508.3190553

[42] Frantzen, M., Shuey, M. (editors), 2001. StackGhost: Hardware facilitated stack protection. 10th USENIX Security Symposium; 2001 Aug 13-17; Washington, D.C., USA.

[43] Novark, G., Berger, E. (editors), 2010. DieHarder: Securing the heap. Proceedings of the 17th ACM Conference on Computer and Communications Security; 2010 Oct 4-8; Chicago, Illinois, USA. New York: Association for Computing Machinery. p. 573-584.
DOI: https://doi.org/10.1145/1866307.1866371

[44] Sayeed, S., Marco-Gisbert, H., Ripoll, I., et al., 2019. Control-flow integrity: Attacks and protections. Applied Sciences. 9(20), 4229.
DOI: https://doi.org/10.3390/app9204229

[45] Andriesse, D., Bos, H., Slowinska, A. (editors), 2015. Parallax: Implicit code integrity verification using return-oriented programming. 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks; 2015 Jun 22-25; Rio de Janeiro, Brazil. New York: IEEE. p. 125-135.
DOI: https://doi.org/10.1109/DSN.2015.12

[46] Mukkamala, S., Janoski, G., Sung, A. (editors), 2002. Intrusion detection using neural networks and support vector machines. Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290); 2002 May 12-17; Honolulu, HI, USA. New York: IEEE. p. 1702-1707.
DOI: https://doi.org/10.1109/IJCNN.2002.1007774

[47] Thottan, M., Ji, C., 2003. Anomaly detection in IP networks. IEEE Transactions on Signal Processing. 51(8), 2191-2204.
DOI: https://doi.org/10.1109/TSP.2003.814797

[48] Cova, M., Felmetsger, V., Banks, G., et al. (editors), 2006. Static detection of vulnerabilities in x86 executables. 2006 22nd Annual Computer Security Applications Conference (ACSAC'06); 2006 Dec 11-15; Miami Beach, FL, USA. New York: IEEE. p. 269-278.
DOI: https://doi.org/10.1109/ACSAC.2006.50

[49] Lanzi, A., Martignoni, L., Monga, M., et al. (editors), 2007. A smart fuzzer for x86 executables. Third International Workshop on Software Engineering for Secure Systems (SESS'07: ICSE Workshops 2007); 2007 May 20-26; Minneapolis, MN, USA. New York: IEEE.
DOI: https://doi.org/10.1109/SESS.2007.1

[50] Miller, B.P., Cooksey, G., Moore, F., 2007. An empirical study of the robustness of MacOS applications using random testing. Operating Systems Review. 41(1), 78-86.
DOI: https://doi.org/10.1145/1228291.1228308

[51] SonarQube Homepage [Internet]. Available from: https://www.sonarqube.org/