Contents lists available at ScienceDirect

Cyber Security and Applications

journal homepage: http://www.keaipublishing.com/en/journals/cyber-security-and-applications/

Survey of techniques to detect common weaknesses in program binaries

Ashish Adhikari*, Prasad Kulkarni*

University of Kansas, Department of EECS, Lawrence, 66045, KS, USA

ARTICLE INFO

Keywords:
CWE detection
Static analysis
Software binaries

ABSTRACT

Software vulnerabilities resulting from coding weaknesses and poor development practices are common. Attackers can exploit these vulnerabilities and impact the security and privacy of end-users. Most end-user software is distributed as program binaries. Therefore, to increase trust in third-party software, researchers have built techniques and tools to detect and resolve different classes of coding weaknesses in binary software. Our work is motivated by the need to survey the state-of-the-art and understand the capabilities and challenges faced by binary-level techniques that were built to detect the most important coding weaknesses in software binaries. Therefore, in this paper, we first show the most critical coding weaknesses for compiled programming languages. We then survey, explore, and compare the *static* techniques that were developed to detect each such coding weakness in software binaries. Our other goal in this work is to discover and report the state of published open-source implementations of static binary-level security techniques. For the open-source frameworks that work as documented, we independently evaluate their effectiveness in detecting code vulnerabilities on a suite of program binaries. To our knowledge, this is the first work that surveys and independently evaluates the performance of state-of-the-art binary-level techniques to detect weaknesses in binary software.

1. Introduction

Technology and software have become integral to our daily lives. More software is now present in more systems, including many embedded devices, like refrigerators and microwave ovens, to cars and planes. Additionally, new software features continue to be added as the hardware, including the processor, memory, and storage, becomes faster, larger, and/or more capable. Thus, software programs also continue to grow in size and perhaps, complexity.

Given this growth in the amount of software in use, it is no surprise that the number of reported code vulnerabilities has been increasing in number and severity for many years [1]. At the same time, software vulnerabilities have been found to cause many disastrous real-world attacks [2,3].

Software vulnerabilities are caused by weaknesses or flaws in the program code. These weaknesses may then be exploited to compromise the security or integrity of the system. Code in any language can be insecure when it is not developed with due care. However, some programming languages are designed with features that make them immune or more resistant to certain types of weaknesses. Such *safer* languages typically provide built-in mechanisms for memory management, input validation, type safety, and other security-related features. Languages like Rust and Go belong to this category of safe programming languages.

Alternatively, *unsafe* programming languages, like C and C++, are low-level languages with poor built-in memory, type, and thread safety. Code bugs and missing safety oversight for vulnerable code constructs are widespread in programs written using these languages [4]. In spite of these safety concerns and even though memory-safe language alternatives are available, C/C++ remains popular due to the large amount of existing legacy code, and low-level features of these languages that are desired by many performance and memory critical, embedded, and real-time systems. Consequently, C and C++ separately and consistently rank among the top five most popular programming languages according to the TIOBE index [1].

In spite of this existing state of affairs where code bugs, vulnerabilities, and exploits are commonplace, most available software has not been independently and rigorously evaluated for its security properties. Most ordinary customers don't have the option of knowing the security, safety, and reliability properties of the software they buy and use. Such unevaluated or under-evaluated third-party software libraries may also

[1] https://www.tiobe.com/tiobe-index/.

Peer review under responsibility of KeAi Communications Co., Ltd.

* Corresponding authors.
E-mail addresses: adhikariashish2@ku.edu, adhikariashish2@gmail.com (A. Adhikari), prasadk@ku.edu (P. Kulkarni).

https://doi.org/10.1016/j.csa.2024.100061
Received 31 October 2023; Received in revised form 9 May 2024; Accepted 22 May 2024
Available online 28 May 2024

be integrated into other products, that may then even be shipped by developers we trust.

Researchers have made important strides to resolve the issues regarding code bugs vulnerabilities, and software exploits. Efforts have been made to understand and categorize the important software weaknesses. A curated community-developed list of the top software and hardware weaknesses was made, which is called the *Common Weakness Enumerations* (CWE) [2]. Each year, a new top CWE list is developed and released that lists the weaknesses that contributed most to the vulnerabilities discovered. This list can help developers and security practitioners address the top vulnerabilities by educating them and permeating the knowledge on how they can be eliminated.

When source code is available, manual code reviews to identify software defects are still a common practice. Researchers have also developed many automated techniques and tools to find weaknesses and vulnerabilities in high-level source code. Lint and PC-Lint may be some of the oldest automated tools developed to detect programming errors and stylistic defects in C and C+ + programs [5]. Similar and more recent source-level tools, commonly called SAST or Static Application Security Testing tools, include Sonarqube [3], CodeSonar [4], Coverty [5], Flawfinder [6], Klocwork [7], and many others [8]. Most of the available source-level SAST tools are commercial and intended to be used at the developer's end to improve software quality.

However, most software is distributed in its binary form and without access to the original source code. *Binary-level* software is typically harder to comprehend and evaluate compared to *source-level* code that is written in a higher-level programming language. Source-level SAST tools cannot analyze third-party binary executables and libraries and are blind to security-reducing decisions made by the compiler, including removal of security checks and memory clearing operations [9] as dead code [10]. Thus, these SAST tools cannot be used to independently confirm the security of the distributed software for end-users of software.

A handful of binary-level SAST tools have also been developed to detect weaknesses and vulnerabilities in software binaries. Examples include Grammatech's CodeSonar for binaries [11] and Veracode SAST [12]. However, these tools are commercial and also intended to be employed at the developer's end to analyze the interaction of any third-party binary libraries with the developer's code base. These tools are mostly inaccessible to independent evaluators and the precision and coverage of these binary-level SAST tools has never been independently explored and evaluated.

Other researchers have also developed binary-level techniques to detect weaknesses and vulnerabilities in binary software, to understand the behavior of closed-source software and malware, to assess compliance with standards, and to enhance software security. These techniques are not only crucial for developers who frequently utilize third-party libraries in their software, but also useful for end-users of software when choosing between competing software options. Our goal in this work is to survey and evaluate the current state-of-the-art techniques designed to address the most important software weaknesses for program binaries.

Specifically, we make the following contributions to this work.

1. We present the top CWEs or weaknesses affecting compiled language binaries.
2. For each top CWE, we survey and describe the cutting-edge static approaches built to detect that weakness in program binaries.
3. We find the binary-level approaches that are available open-source, build and independently evaluate them when possible, or reveal the challenges when not.
4. We build working prototypes for two open-source binary-level tools, and compare their accuracy and shortcomings for several important CWEs using multiple standard benchmark programs.

The rest of this paper is organized as follows. We present related work in Section 2. We describe the methodology we used to conduct this survey in Section 3. We discuss the major static binary analysis techniques in Section 4. We survey the techniques that have been developed to detect each of the top 10 CWEs for compiled language binaries in Section 5. We search for open-source implementations of the techniques covered in Section 5, attempt to build and evaluate them, when available, and report our results and observations in Section 6. We discuss the issues and directions for future advancement of static analysis techniques for vulnerability detection in Section 7. We present the limitations of our current work in Section 8. Finally, we list avenues for future work and present our conclusions in Sections 9 and 10, respectively.

2. Related works

In this section, we review the existing literature on surveys and techniques for detecting Common Weakness Enumerations (CWEs) and software vulnerabilities for binaries and compare them with our work in this paper. A summary of the most relevant related works is presented in Table 1.

We did not find any previous work that compiled and presented a comprehensive survey on techniques and tools for the detection of multiple top CWEs. However, there are several studies that have surveyed detection techniques for specific CWEs. A majority of these works only study and evaluate source-level techniques. For example, Byun et al. utilized the CMBC tool on the Juliet Test Suite to evaluate and detect CWEs [6]. Other research explored Natural Language Processing (NLP) techniques to generate source code embeddings, that then aid in the automatic detection and classification of software vulnerabilities [7]. Cruzes et al. conducted a thorough survey investigating techniques for detecting only the Buffer Overflow (BO) vulnerabilities [8]. They organized the techniques into multiple categories and found common limitations of the techniques in each of their categories. While comprehensive, their study specifically focused on buffer overflow vulnerabilities and their techniques focused on source-code level techniques.

In this work, we primarily focus on comparing (binary-level) *static* analysis based techniques for CWE detection. We found several other works that similarly focused on studying static analysis based techniques for CWE detection. Lipp et al. conducted an empirical study on the effectiveness of static C code analyzers for real-world vulnerability detection [15]. They assessed the ability of several open-source tools and one commercial static C analyzer and found that all current tools do a poor job at detecting real-world vulnerabilities, even when they performed well on artificial/smaller benchmarks. Katherina et al. investigated the strengths and weaknesses of static code analysis tools in detecting CWEs and other vulnerabilities [17]. Although the specific names of the tools were not mentioned, their evaluation revealed that the tested commercial tools did not exhibit statistically significant differences in their ability to detect security vulnerabilities. They underscored the need for further advancements in vulnerability detection techniques. In another study, Pereira et al. evaluated two static analysis tools for their applicability in large projects [11]. They found that these tools exhibited diverse performances, with `Flawfinder` having higher false alarms but fewer true negatives, while `cppcheck` showed high true negatives

[2] https://cwe.mitre.org/.
[3] https://www.sonarqube.org/downloads/.
[4] https://www.grammatech.com/codesonar-cc.
[5] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.
[6] https://dwheeler.com/.
[7] https://www.perforce.com/products/klocwork.
[8] https://owasp.org/www-community/Source_Code_Analysis_Tools.
[9] https://cwe.mitre.org/data/definitions/14.html.
[10] https://cwe.mitre.org/data/definitions/561.html.
[11] https://www.grammatech.com/codesonar-sast-binary.
[12] https://www.veracode.com/products/binary-static-analysis-sast.

Table 1
Summary of related works We categorize the *Technique* used in the research works as follows: SC – Source Code, BA – Binary Analysis, SA – Static Analysis, DA – Dynamic Analysis, SE – Symbolic Execution and ML – Machine Learning.

Research Paper/ Tool	Short Description	Technique
Ahmed et al. (2022) [9]	A survey about machine learning techniques and datasets being used for software vulnerability detection. Research studies focused on were recent. CNN and RNN can give better performance than others.	ML
Yosifova et al. (2021) [10]	Predicting vulnerability type in Common Vulnerabilities and Exposures (CVE) database with machine learning classifiers. They gave their evaluation of different ML classifiers for the detection of CVE.	ML
Pereira et al. (2020) [11]	Uses open-source C/C+ static analysis tools on large projects. They use two static analysis tools and study their applicability to detect data protection vulnerabilities and coding practices vulnerabilities.	SC, SA
Zaharia et al. (2021) [12]	CWE pattern Identification using Semantical Clustering of Programming Language Keywords. They use source code to detect the CWEs incorporating the programmers' code behavior.	SC, ML
Tiantian et al. (2018) [13]	A survey of automatic software vulnerability detection, exploitation, and patching techniques. Binary-based techniques are also studied. Nice breakdown of techniques.	SA, DA, ML
Alenezi et al. (2020) [14]	Machine Learning approach to predict computer operating systems vulnerabilities. They use five ML methods to predict the vulnerabilities based on CVSS and evaluate them. Random forest seems to be a good classifier. No neural network models.	ML
Byun et al. (2020) [6]	Analysis of software weakness detection of CBMC based on CWE. Evaluate the ability of CBMC to detect the CWEs. Found the tool to be effective on division by zero, conversion error, and buffer overflows.	SA
Saletta et al. (2020) [7]	Use NLP in source codes for identifying CWEs. The classification of 13 CWEs was done. Some strictly related CWEs are misclassified for a Java file.	SC, ML
Lipp et al. (2022) [15]	Empirical study on the effectiveness of static C code analyzers. They evaluated five open-source and one commercial static C code analyzer. They found that 47%-80% of the real-world vulnerabilities are missed by them. A combination of static analyzers delivered better performance.	SC, SA
Cruzes et al. (2018) [8]	Survey on techniques for detecting Buffer Overflow (BO) vulnerabilities. A comprehensive survey on buffer overflow detection techniques where multiple techniques and tools are categorized and reviewed. The binary analysis techniques are also discussed and according to their findings, few recommendations are made.	SC, SA, BA, DA, ML, SE
Lin et al. (2020) [16]	Survey on deep learning-based approaches for software vulnerability detection. Recent studies adopting deep learning techniques for software vulnerability detection are done with their challenges and weaknesses.	ML, SC, BA
Katherina et al. (2015) [17]	Evaluation of static code analysis tools in detecting CWEs and vulnerabilities. The study evaluated three tools and found no statistical difference in their ability to detect security vulnerabilities for C/C+ and Java.	
Shoshi- taishvili et al. (2016) [18]	Comparison of binary analysis techniques and introduction of angr framework. Many different binary techniques are studied and implemented in their framework;angr. The effectiveness of the techniques was evaluated. The difficulties of combining many techniques are discussed.	SA, BA, SE,
Xue et al. (2020) [19]	Study on machine learning-based analysis of program binaries with taxonomy and associated challenges. The paper explores challenges in binary code analysis, discusses various machine learning techniques, and presents a framework and its application.	ML, BA

but lower false positives. This work is relevant to our evaluation of tools for CWE detection as both tools are CWE-compatible. However, all these earlier works only studied source code based tools. Instead, our goal in this work is to study and evaluate binary-level CWE detection tools.

Given the popularity of machine-learning techniques, researchers have also explored their use for software vulnerability detection [10,12–14]. These studies involved the classification and prediction of weaknesses and vulnerabilities using machine learning methods. Notably, Zaharia et al. employed the CWE classifier on the intermediate representation of source code to identify security patterns [12]. They demonstrated the effectiveness of machine learning techniques in aiding the detection and characterization of software vulnerabilities. Recent developments also highlight the use of deep learning techniques to understand vulnerable code patterns and semantics. Lin et al. conducted a survey that reviewed the literature on deep learning-based approaches for software vulnerability detection [16]. They examined both the techniques employed and the challenges faced by such techniques for accurately detecting vulnerabilities.

All of these previous works only studied source-code based detection techniques. Instead, our research aims to explore detection techniques, open-source tools, and associated challenges to detect CWEs for compiled language binary programs that can be detected without access to the source code.

A limited number of previous studies have investigated binary analysis based techniques to uncover vulnerabilities and weaknesses in binary code. Shoshitaishvili et al. conducted a comprehensive comparison of various binary analysis techniques, such as symbolic execution, fuzzing, static analysis and exploit generation, and hardening [18]. They introduced angr, a powerful binary analysis framework capable of performing such analyses and aiding their comparison in a uniform framework. This work also implemented and compared diverse approaches, including fuzzing and static analysis, for identifying and mitigating vulnerabilities. Similar to angr, several other frameworks have been build that implement the building-block algorithms and provide an API to conduct binary-level analysis, including Ghidra [20], Radare [21], IDA Pro [22], BAP [23], and DynInst [24]. However, most of these tools are not implemented with built-in algorithms and techniques to directly perform CWE detection. A comprehensive study by Xue et al. examined machine learning-based analysis of program binaries, providing a taxonomy of techniques along with their associated challenges [19]. Although we did not find surveys that focus specifically on techniques to detect CWEs in program binaries, the aforementioned studies are relevant to our research. Our emphasis lies in studying, collecting, and assessing works and tools, especially those that are open-source, to detect the presence of the most important CWEs that occur in compiled binary code.

2.1. Comparison with existing literature

Our survey paper stands out in the existing literature due to its specific focus on the detection of Common Weakness Enumerations (CWEs) and software vulnerabilities in binaries, a topic that has received limited attention in prior research. While much of the existing literature centers on source-level analysis [7,11,12,15], our work fills a crucial gap by examining these issues specifically at the binary level.

In contrast to related surveys, such as the empirical studies conducted by Lipp et al. on the effectiveness of static C code analyzers [15], our paper is dedicated to the exploration of static techniques for detecting weaknesses and vulnerabilities in binaries. While some surveys cover a broad range of topics that discuss a range of techniques implemented on the source level [25], our paper maintains a clear focus on binary-level techniques and tools for the detection of weaknesses, providing evaluations of the current state of the art.

Furthermore, while several other works focused on specific categories of weaknesses or techniques [8,16], our survey provides a comprehensive overview of static analysis techniques for detecting vulnerabilities across various categories in the binary context.

Additionally, while some studies focused solely on binary analysis without delving into weaknesses and vulnerabilities [18], our paper explores the technical aspects of binary analysis and also specifically addresses the detection of weaknesses and vulnerabilities, offering evaluations of current tools.

Our overarching goal is to contribute to the dissemination of knowledge about the current state of research and techniques in binary-level detection of weaknesses and vulnerabilities, filling a critical gap in the existing literature.

3. Methodology

The landscape of CWE detection techniques is diverse, encompassing both well-studied and lesser-known approaches. To explore the current state-of-the-art tools and techniques employed in CWE detection, we conduct an extensive review of research papers and evaluate open-source tools. We describe our research methodology in this section. We display a schematic of our methodology in Fig. 1.

Selection of CWEs In our study, we primarily focus on compiled languages – that generate executable binary files. To identify the most critical weaknesses that could result in significant software vulnerabilities, we refer to the top 25 CWEs, along with an additional set of 15 weaknesses, published by the MITRE Corporation. Then, we narrow down this set of weaknesses to the top 10 CWEs that are specifically applicable to compiled languages. The list of the top 10 CWEs relevant to compiled binaries is shown in Table 2. The first column shows their rank in the overall CWE database, while the remaining columns display their ID and a short description, respectively.

Selection of relevant research studies

We conduct the following steps to find related research works. First, we identify *primary studies* that are chosen based on their direct relevance to the subject matter. This step involved identifying studies that specifically addressed CWEs, their detection techniques, and related works. Next, we selected papers based on their references to related works. This step allows for the broader inclusion of studies that have been cited or referenced in the context of CWE detection.

We used a few other criteria to choose research works considered in this study. In addition to works on CWE detection on binaries, we also found papers on CVEs (Common Vulnerabilities and Exposures) and vulnerability detection techniques.

Our selection process also prioritized newer techniques, including those incorporating machine learning (ML) and artificial intelligence (AI) approaches. We gave a special preference to open-source projects and research studies to facilitate tool evaluation and repeatability and comparison of results. The selected papers were further categorized into open-source research projects and closed-source projects.

Selection of research tools to study state-of-art One of our goals in this work is to conduct a fair comparison of the capabilities and performance of the available open-source implementations of techniques to detect CWEs in software binaries. Therefore, for each selected CWE, we attempted to find, build, and evaluate the implementations of proposed mitigation techniques on a set of common standard benchmarks.

Unfortunately, we observed that the number of open-source tools specifically designed for detecting CWEs in binaries was limited. Fur-

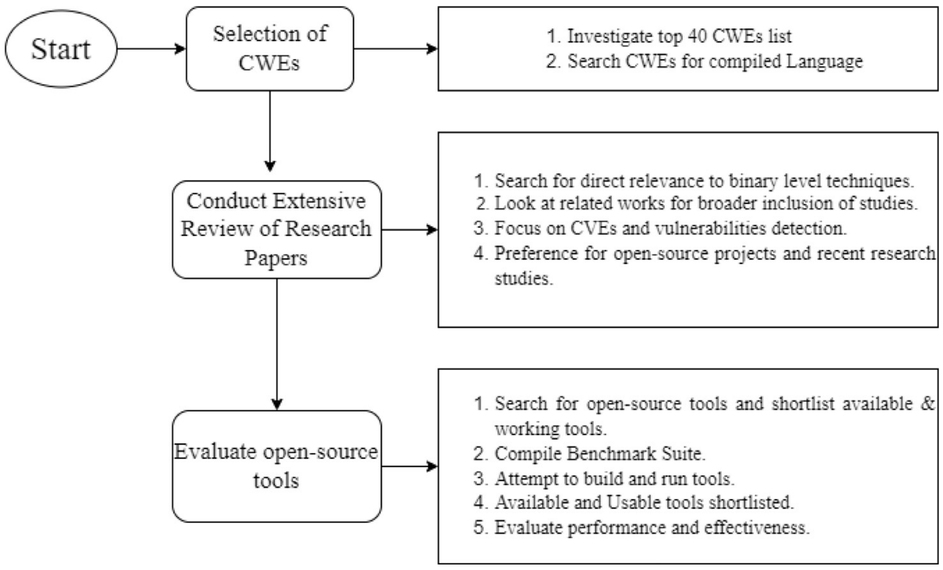
Start
Selection of CWEs
1. Investigate top 40 CWEs list
2. Search CWEs for compiled Language
Conduct Extensive Review of Research Papers
1. Search for direct relevance to binary level techniques.
2. Look at related works for broader inclusion of studies.
3. Focus on CVEs and vulnerabilities detection.
4. Preference for open-source projects and recent research studies.
Evaluate open-source tools
1. Search for open-source tools and shortlist available & working tools.
2. Compile Benchmark Suite.
3. Attempt to build and run tools.
4. Available and Usable tools shortlisted.
5. Evaluate performance and effectiveness.

Fig. 1. A flow-graph schematic of our research methodology

Table 2
TOP 10 CWEs for compiled language program binaries.

Rank	ID	CWE
1	***CWE-787***	***Out-of-bounds Write***
4	***CWE-20***	***Improper Input Validation***
5	***CWE-125***	***Out-of-bounds Read***
7	CWE-416	Use After Free
11	***CWE-476***	***NULL Pointer Dereference***
13	***CWE-190***	***Integer Overflow or Wraparound***
15	***CWE-798***	***Use of Hard-coded Credentials***
19	***CWE-119***	***Improper Restriction of Operations within the Bounds of a Memory Buffer***
31	***CWE-843***	***Access of Resource Using Incompatible Type ('Type Confusion')***
36	CWE-401	Missing Release of Memory after Effective Lifetime

thermore, many of these tools were not in working condition due to dependency issues, configuration problems, or a lack of maintenance. The dynamic nature of software, policy changes, and lack of developer motivation are factors contributing to the usability issues of these tools.

Some tools required fulfilling multiple dependencies, making their installation and usage demanding. Several others made various assumptions about the working environment or about the format of the programs tested. Despite the challenges, the tools that worked and were relevant to the research were shortlisted, built, and used for benchmarking and evaluation.

Benchmark selection Multiple benchmarks, such as SPEC2017 [26], SARD [27], and Juliet [28] test suite, were used to evaluate the performance of the selected CWE detection tools. All these programs were compiled with the Clang/LLVM compiler with no optimizations. Some of the benchmarks, like SARD and Juliet, provide a ground truth regarding errors and vulnerabilities in each program. For others, like the SPEC benchmarks, we do not have information about any known software defects. For such cases, and to compare the performance and detection accuracy of binary-level tools with a source-level approach, we also employed a popular and well-regarded source-level CWE detection tool, called Sonarqube [29].

4. Analysis techniques

Researchers have developed and used different techniques to detect software vulnerabilities. In this section, we present a brief introduction to the main methods used by vulnerability detection techniques for software binaries. We limit the scope of this work to static analysis based techniques. We do not discuss dynamic analysis based software weakness detection techniques, such as fuzzing [30,31] in this work. Dynamic analysis based techniques can produce fewer false positives, but only analyze code traces that are reached during execution with the limited set of provided inputs.

4.1. Static analysis

Static program analysis consists of algorithms and heuristics to extract information about the program, without executing the program being analyzed [32]. Static analysis is a popular technique for finding program vulnerabilities. The binary code is statically analyzed to create detailed models of the application's data and control flow. These models are then used for various tasks from vulnerability detection, and race condition detection to performance optimization. The static analysis enables inspection of the entire program code but is known to result in several false positives and false negatives [33].

4.2. Symbolic execution

Symbolic execution was devised as a way of executing a program abstractly, by replacing normal program inputs with arbitrary symbols that characterize a set of classes of inputs [34]. Symbolic execution can find abstract inputs to explore different paths of the program, which makes this technique useful for vulnerability analysis. This is a popular security technique [35–37]. However, this technique suffers from a high run-time overhead and the path explosion problem, which restricts its applicability [8]. Research is ongoing to determine ways to avoid the path explosion issue, such as filtering out the irrelevant dependencies in symbolic values.

4.3. Taint analysis

Taint analysis is a technique used to identify and monitor values that could be influenced by or derived from plausibly malicious program inputs. Taint analysis can be applied statically [38–40] or dynamically [41]. The tracking of tainted data and their propagation can be used to detect program errors. The taint analysis method may find it hard to explore all paths, and some errors might go undetected. This technique delivers few false positives but can have false negatives as all paths might not be explored.

4.4. Machine learning

Recently, machine learning based techniques have gained much popularity for vulnerability detection and other security tasks [42,43]. Different ML models have been used in these studies, from classical ML models, like random forest and SVM, to deep learning models, including RNN and transformers [44]. Usually, a vulnerability database is needed to train the dataset, which might not always be readily available. ML based techniques are often coupled with static analysis to gather the datasets from the binary and then trained to get the classification results. Training the ML model can also be expensive in terms of time and compute resources, though inference is typically much faster.

5. Survey of techniques to detect coding weaknesses in software binaries

In this section we present a comprehensive survey of static techniques and tools developed to detect the ten most important coding weaknesses for binary software. We organize this section according to the order of the top coding weaknesses as listed in Table 2.

5.1. Buffer overflow

Buffer overflow is a notorious memory error that has plagued software security for decades. Yet, even after much research and effort to address this issue, errors related to buffer overflows are still the dominating cause of failures and attacks in binary software.

The buffer overflow issue occurs when reading from or writing to memory that exceeds the buffer or memory allocated at that region. This flaw has numerous consequences like the execution of unauthorized code or commands. Attacks such as denial of service, crashes, resource consumption, remote code execution, etc. are also possible [45]. Buffer overflows can expose sensitive data and cost businesses huge monetary and reputation losses. These weaknesses have been responsible for vulnerabilities in several instances, such as CVE-2021-22991 and CVE-2020-29557.

Buffer overflow is a broad term that encompasses several distinct CWEs. Of these, CWE-787, CWE-125, and CWE-119 are in our list of top 10 CWEs plaguing binary software, as listed in Table 2. Others, like CWE-121 and CWE-122, are not in our top 10 list. Below, we provide further details about the more prominent CWEs related to buffer overflows.

- CWE-787 Out-of-bounds Write: This CWE tops the list of the 25 most dangerous software weaknesses. This error occurs when the software writes the data past the end, or before the beginning of the intended buffer. Usually, this unintended overwrite can result in the corruption of data, program crashes, or incorrect execution of code.
- CWE-125 Out-of-bounds Read: This weakness occurs when the program code reads data past the end, or before the beginning, of the intended buffer. Exploiting this weakness can allow attackers to read sensitive information from other memory locations or cause a program crash. When the excess data (out of the bound) is read, it can expose sensitive program data.
- CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer: This is a more general weakness category that indicates a memory read or write that is outside the intended buffer boundary.

Static analysis techniques used to detect BO include program slicing, pointer analysis, and delta debugging. Additionally, some researchers

Table 3
Summary of binary-level studies to detect the Buffer Overflow Weakness.

Research	Open sour- ce?	Technique	Evaluation Results	Tools/Li- braries Used	Benchm- arks	Limitations
Wang et al. (2021) [35]	No	Static analysis, Taint analysis, concolic exec.	88% accuracy	None	Juliet and SARD	Synthesized, small benchmarks, poor evaluation on real-world programs, closed-source
Padaman-bhuni, Tan(2014) [46]	No	Static analysis, dynamic analysis, ML	92% recall, 81% precision	Pin, WEKA, CodeSurfer	MIT Lincoln	Small benchmarks, low accuracy, closed-source
Padaman-bhuni, Tan(2015) [48]	No	Static analysis, ML	75% recall, 84% precision	ROSE, WEKA, IDA Pro	MIT Lincoln	Small benchmarks, low accuracy, closed-source
Gao et al. (2020) [50]	No	Taint analysis, static analysis	94.3% precision, 86.2% recall	None	Self-generated	Small benchmarks, bigger binaries take longer to complete, many false positive, closed-source.
Liang et al (2017) [51]	No	Dynamic taint analysis, data recovery, dynamic instrumentation	Nine out-of nine real-world heap overflow programs	QEMU, udis86	9 real-world programs	Small benchmarks, closed-source
Xiangkun et al. (2017) [36]	No	Dynamic, taint analysis, symbolic execution, fuzzing	47 new vulnerabilities	QEMU, solver Z3	17 real-world programs	Small benchmarks, misses true vulnerabilities, slow, huge trace size, closed-source
Dahl et al. (2020) [52]	Yes	RNN, static analysis	CCR of 99%	web scraper	self-generated	Restricted to small benchmarks, cannot analyze bigger programs
Gotovchits et al. (2018) [53]	No	Static analysis, static and dynamic taint analysis	Five Zero day errors	BAP, μflux	COTS, Coreutils	Small benchmarks, no comparison with other similar tools
Xu et al. (2022) [54]	Yes	Symbolic execution, dynamic analysis	22 out of 29 program errors	angr, radare2	24 CTF and 5 CVE programs	Tool installation issues,limited maintenance, path explosion
Valgrind [55]	Yes	Synamic analysis and instrumentation, JIT	-	Memcheck	-	Limited support for stack, static array overflow, slow
CWE Checker[56]	Yes	Static analysis, symbolic execution	-	Ghidra, BAP	-	Many false positives and false negatives, slow

combine multiple techniques to increase their effectiveness. There has been less research on detecting buffer overflow issues for software binaries compared to source-level techniques. In this section, we provide a summary of the binary-level research efforts on this topic, which is also presented in Table 3.

Wang et al. (2021) [35] developed a new tool, named BOF-Sanitizer, to locate buffer overflows, where they combined a metric and rank to find the vulnerable locations where potential buffer overflow could exist. Then the taint analysis method is used to find the vulnerable input parameters that are symbolized by dynamic symbolic execution technology and sent to a detection engine and a custom memory model where the buffer overflow is detected. Their approach achieved 88% accuracy on small benchmarks in the SARD and Juliet test suites. They also claimed to find 91 out of 100 vulnerabilities in some other real-world programs.

Padamanbhuni and Tan [46] proposed a vulnerability prediction technique by identifying potentially vulnerable program constructs during program analysis and getting the buffer usage pattern from code attributes extracted from those constructs. Then, machine learning methods were used to predict the buffer overflow. They performed their program analysis to accurately model an instructions semantics using ROSE, a binary analysis framework, and off-the-shelf tools like WEKA [47] and IDA Pro [22]. The same authors also combined static and dynamic techniques to identify buffer overflows [48]. They automatically extracted code attributes from C/C++ programs and use the Pin tool [49] for performing the dynamic analysis and the WEKA data mining tool to train the vulnerability prediction models.

Gao et al (2020) [50] tried detecting buffer overflows based on abnormal program execution. They took instances of successful and abnormal executions where a group of input data is passed to a program. They took the memory of successful execution recovery as a buffer boundary and judge whether the boundary has overflowed in abnormal execution.

Some researchers focus on specific sub-categories of buffer overflow. Xiangkun et al. (2017) [36], proposed a heap overflow detection tool called HOTracer which models heap overflows as spatial inconsistencies between heap allocation and heap access operations. They performed analysis on program traces and then recognized the heap allocation and heap access operation pairs and checked whether there are spatial inconsistencies to detect the potential vulnerability. They tested their tool on certain software programs like KM player, VLC, iTunes, etc.

Researchers developed a tool, named HCSIFTER (2017) [51], to detect heap overflows through dynamic execution without the need for source code. The tool detected five of the nine overflows in their tested programs. The tool can also assess the programs for their exploitability by executing the program binary and analyzing the crash points and exploit points.

Dahl et al. (2020) [52], proposed a stack-based buffer overflow detection method using recurrent neural networks. They treated the assembly code as a natural language and process it using recurrent neural networks based on long short-term memory cells. The dataset/benchmark is self-generated and may not represent real-world data. The Correct Classification Rate (CCR) was indeed similar but the dataset it used comprised small functions that resembled the SARD benchmarks.

Baradan et al. (2022) [37], proposed a unit-based symbolic execution method for detecting four classes of memory corruption vulnerabilities in executable codes. The units are small program codes that might contain vulnerable statements and they are statically identified. Each unit is then subjected to symbolic execution to calculate the path and vulnerable constraints of each statement. Solving these constraints would reveal vulnerabilities, if any.

Gotovchits et al. (2018) [53] proposed a tool, named Saluki, for detecting taint-based security properties in binary code. The tool tried to find different CWE types like missing sanitization checks, command injection, or checks on buffer lengths. It used ôflux, a context and path-sensitive analysis technique to recover data dependence facts in binaries and tried to perform a sound logic system for reasoning over these facts i.e. if they satisfy a security property. Some of the CWEs it tries to find include CWE-252, CWE-89, CWE-337/676, CWE-120, and CWE-78.

Xu et al.,(2022) [54] attempted to find the stack buffer overflow vulnerabilities and generate an exploit. Their BofAEG tool used symbolic execution and dynamic analysis to detect vulnerabilities. Out of 24, 22 of the vulnerabilities were found in their self-collected programs.

BAP is a binary analysis platform developed at Carnegie Mellon University that enables the analysis of binary programs [23]. It includes various different analyses, microexecution interpreters, standard interpreters, and a symbolic execution engine. There is a BAP toolkit repository that has different tools that perform different checks, including buffer overflow detection. Although relatively easy to use, it only detects heap overflows.

CWE_checker is a large suite of binary-level tools that can detect multiple classes of errors in program binaries [56]. The CWE_checker uses Ghidra to disassemble binaries into one common intermediate representation and then implements its analyses on this IR. CWE_checker has implemented checks for many different bugs in program binaries, including those related to buffer overflows, such as CWE 119, CWE-125, and CWE-787.

Valgrind [55] is a popular run-time framework that provides several binary-level debugging and profiling tools. Memcheck is one Valgrind tool that can help find memory leaks in program binaries during execution.

5.2. CWE-20 – improper input validation

CWE-20 is caused when the program input is not validated or is incorrectly validated. Proper input validation requires the input supplied to be checked to determine if it is valid and conforms to the program's expectations The absence of input validation can result in severe exploits, including buffer overflow and resource consumption attacks. This CWE has been linked to many CVEs, including CVE-2021-22205 and CVE-2008-3477. Researchers have claimed that developers often overlook this issue due to inadequate knowledge and training, even though it is relatively easy to detect and fix [57].

We found the detection and fixing of this weakness is typically performed at the source code level. Unfortunately, we did not find any dedicated binary analysis tools designed for the detection of CWE-20.

5.3. CWE-416 – use after free

The CWE-416 error occurs when the program tries to reference the memory that has already been freed. Thus, there are three things the program must do to trigger this error, (a) allocate heap memory, (b) free the memory, and (c) access the freed heap memory again. This error can have multiple consequences like corruption of valid data, crashes, execution of arbitrary code, denial of service, execution of unauthorized code or commands, etc. This weakness is also called the dangling pointer error. Some CVEs associated with this CWE are CVE-2020-6819 and CVE-2021-0920.

Zhang et al. [58], presented a multi-level directed greybox fuzzing tool, called MDFuzz, to detect the use-after-free errors by covering only specific heap operations. Although this is a fuzzing-based technique, it utilizes static analysis to automatically recognize three critical targets related to heap operations: allocating heap memory, freeing memory, and accessing the heap memory. It then improves the directed fuzzing process by using a novel seed selection strategy and probability-based multi-level seed queue. The tool was evaluated on 7 real-world applications.

Zhu et al. [59] developed the UAFDetector tool that also combines static analysis techniques with dynamic mechanisms. This paper focused on improving the CFG construction with the help of dynamic binary instrumentation techniques to resolve indirect jumps. The technique performs alias analysis and pointer tracking. They use IDA Pro and BinNavi for building their tools. This tool is evaluated on the Juliet benchmarks and real-world programs with known vulnerabilities. The evaluation found 2.39% false negative with the Juliet benchmarks, and 5 out of 6 real-world cases were detected.

GUEB [60] is a static analyzer to perform use-after-free detection on binaries. It uses value set analysis and tracks pointers and states of the heap objects. The program sub-graph is extracted when GUEB detects the use of the freed pointer. This tool also uses IDA Pro and BinNavi to perform its analysis. Large binaries cannot be analyzed using this tool.

Yan et al. [61] introduced a static UAF detector called Tac, that utilized machine learning to bridge the gap between typestate and pointer analyses. They utilized support vector machines to learn the correlation between program features and UAF-related aliases. They tried to find the true UAF bugs with reduced false positives by removing imprecise aliases using machine learning. They used program slicing and performed their path-sensitive type state analysis in addition to machine learning to get the desired output.

In addition, the open-source tools we used earlier, cwe_checker [56] and BAP [23] also detect use-after-free errors in binary code. Both these tools have a high false positive rate.

Many other execution-time tools have also been built to detect use-after-free errors in binary code. Such tools include dynamic fuzzers, like UAFUZZ [62], and instrumentation frameworks, like Valgrind [55]. Memcheck is one of the tools that employ Valgrind to find several memory leaks in C and C++ program binaries. These are out-of-scope for this paper.

5.4. CWE-476 – null pointer dereference

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit. When a program is trying to dereference a null pointer, it is accessing memory at an invalid address, which typically can lead to unexpected behavior, crashes, and security vulnerabilities. It can be difficult to detect and fix this error in the case of large programs. This weakness can be exploited to cause serious attacks that include the ability to bypass the security logic, make the program reveal some debugging information, or cause abnormal program crashes and other DoS attacks [63]. Some vulnerabilities associated with this CWE include CVE-2020-6078 and CVE-2020-29652.

Static and dynamic taint analysis, symbolic execution, fuzzing, data flow and control flow analysis, and dynamic binary instrumentation techniques are some popular mechanisms to address CWE-476 in recent research. Again, we only focus on presenting binary-level static analysis techniques in this work.

The CWE-Checker tool can find this error for cases where a pointer is explicitly set to Null before the pointer is used in the function [56]. However, we found that this tool does not yet find the null pointer dereference if a *NULL* parameter sent to a function is dereferenced.

A tool, called NPDHunter uses an intra-procedural pointer and taint analysis based approach to detect null pointer dereferences in binary code [64]. This work uses an improved pointer aliasing analysis to categorize and identify untrusted source cases, and then perform taint-style vulnerability testing to detect whether the data from an untrusted source propagates to a sensitive sink without proper sanitization. The authors here note that static detection methods on binaries, including their own, are limited due to challenges caused by complex code structures including loops, indirect calls and jumps, etc. Even so, their technique achieved no false negatives for benchmarks in the Juliet suite. However, there were a large number of false positives. They also evaluated the cwe_checker tool and reported that cwe_checker had 70% false positives for this check with the same benchmarks. Unfortunately, this tool is not available as open-source, and our attempts to contact the authors were unsuccessful. They use BAP for IR generation.

Tobais et al. developed a tool, called TEEREX that uses a combination of symbolic execution and static analysis to identify potential sources of null pointer dereference errors in SGX enclaves in binary code; later run-time instrumentation is used to monitor the execution to detect any actual null pointer accesses [65]. The authors showed that null pointer dereferences can be used to cause memory corruption and compromise the security of the enclave.

Gotovchits et al. proposed a taint-style tool, Saluki, for statically checking security properties and detecting different vulnerabilities [53]. They combined static analysis with taint analysis to perform path-sensitive and context-sensitive recovery of the data dependence facts in binaries. They then checked if the data dependence facts adhere to their rules to report potential vulnerabilities. They applied their tool to five real-world applications and the ARM coreutils binaries. They can check for numerous potential errors, including “Unchecked Return Values”. This error can be associated with the Null pointer dereferencing error as denoted by CWE-690. They also use BAP for IR generation. BAP and CWE_checker also provide tools to detect CWE-476.

We have found that standalone static analysis techniques are generally not used to detect this error. Static techniques are often coupled with other techniques such as taint analysis, dynamic analysis, or fuzzing. For instance, Vishnyakov et al. presented an approach that uses symbolic execution, dynamic analysis, and hybrid fuzzing to detect various real-world software flaws, including null pointer dereferences [66]. They implemented their hybrid fuzzing tool by combining the Sydr [67] with libFuzzer [68] and AFL+ + [69]. They used slicing to improve symbolic execution. This work used their self-created OSS-Sydr-Fuzz repository and showed that their approach achieved higher coverage than other related tools.

5.5. CWE-190 – integer overflow or wraparound

Integer overflow is one of the most common types of software vulnerabilities that occurs when a calculation or operation results in a value that is outside the range of values that can be stored in an integer data type. When this occurs, the value can wrap to become a very small or negative number. These integer overflows can cause the program to use incorrect numbers and respond in unintended ways. For instance, if the malformed value generated by integer overflow is used to determine how much memory to allocate, it will cause a buffer overflow which is known as Integer Overflow to Buffer Overflow (IO2BO) vulnerability [70]. Other attacks that are possible by exploiting these weaknesses include denial of service, program crashes, resource consumption issues, and arbitrary code execution. CVE-2018-10887 and CVE-2019-1010006 are examples of actual vulnerabilities that were caused by CWE-190 Integer Overflow. Below, we review static approaches devised to detect this code weakness.

Wang et al. presented a tool, called IntScope, that can automatically detect the integer buffer overflow vulnerabilities in x86 binaries [71]. It lifts the disassembled code into its intermediate representation (IR) and performs a path-sensitive data flow analysis and identifies the vulnerable points for the integer overflow using symbolic execution and taint analysis. Their mechanism used various tools like IDA Pro [22], Bestar [72], GiNaC [73], and STP [74]. This tool was evaluated on two of the Microsoft programs and was successful in detecting all known vulnerabilities. Additionally, it found more than 20 zero-day integer overflows in these programs along with several false positives.

Muntean et al. built a tool, named INTREPAIR, to detect and fix integer overflows in software Binaries [75]. This technique employs symbolic execution. The tool only focuses on program paths that are fault-prone like assignment or multiplication. They conducted their evaluation on the Juliet test suite and another synthesized benchmark set of 50 programs. This tool successfully detected all actual overflows, but also produced many false positives (the actual number is not reported).

Huang et al. proposed a hybrid method to detect integer overflow errors [76]. They used static taint analysis to find the program points to instrument. The instrumented test code at each *use* and *def* site checks for the overflow. The delayed runtime test minimizes the false positives.

Zhang et al. also proposed a hybrid method that combines symbolic execution, static analysis, and dynamic taint analysis to detect the integer overflow to buffer overflow vulnerabilities in program binaries [77]. They used the Juliet test suite to evaluate their approach and found zero false positives and zero false negatives.

5.6. CWE-798 – use of hardcoded credentials

This weakness occurs when the developer uses hard-coded credentials, such as passwords or cryptography keys, which it uses for sensitive purposes, for both inbound and outbound variant authentication. This weakness can allow an attacker to bypass the authentication. A simple string search could sometimes reveal the hard-coded credentials in the binary. This weakness can cause attacks, such as gaining unintended privileges and execution of unauthorized code or commands. This CWE has been associated with real-world vulnerabilities, including CVE-2022-30314 and CVE-2010-2772.

This error can be mitigated by storing the passwords, keys, and other credentials outside of the code in a strongly protected and encrypted configuration file or a restricted database. The access control should be limited in the case of the hard-coded credentials.

Binary analysis tools such as IDA Pro [22], Ghidra [20], Radare [21], and Angr [18] can be utilized to detect possible strings that reveal hard-coded credentials in the binary. Various debuggers can also be used to detect the hardcoded credentials. But both these approaches need some manual work.

There are not a lot of research papers or released tools that propose techniques to detect the presence of hard-coded credentials in binary software. The BAP binary-level tool claims to detect this weakness [23].

Source code level SAST tools, like Sonarqube [29], Veracode [78], and Checkmarx [79] use static analysis techniques to detect the hard-coded values in the program. GitGuardian [80] is another source-level approach that detects the hard-coded secrets in code repositories and performs real-time monitoring to detect secrets in every new commit.

5.7. CWE-843 – access of resource using incompatible type

CWE-843, also called *Type Confusion*, occurs when a program initializes a resource, such as a pointer, object, or variable, using one data type but later accessing it with an incompatible type. This issue can potentially cause other logical errors as the resource may not have the expected properties, and can also result in out-of-bounds memory accesses. This bug has caused several real-world vulnerabilities, such as CVE-2010-4577 and CVE-2011-0611.

Accurate type detection for binaries, especially after *stripping*, is challenging due to factors such as vanished type casting operators, missing class information, and unknown runtime type information. Consequently, most previous research to detect this error assumes access to the program source code. For instance, Haller et al. proposed TypeSan, a type confusion detection tool that extends the LLVM compiler [81]. TypeSan identifies invalid casts by instrumenting the code to monitor object allocations and potentially unsafe casts. Similarly, tools like Effectivesan [82], Htade [83], and Hextype [84] also rely on compilers to detect this weakness.

Kim et al. introduced a hybrid tool called BinTyper, which combines static and dynamic analysis to detect type confusion in C+ + binaries [85]. Through static analysis, BinTyper recovers the class hierarchy and layout of the binary. It then uses dynamic analysis to identify type confusion when an application accesses a member variable of a polymorphic object. BinTyper was evaluated with Google PDFium and successfully detected some type confusion bugs. However, this method is limited to detecting errors when objects access polymorphic objects, and the coverage is restricted to executed code. This is the only binary-level approach we found to detect CWE-843, and this tool is not open-source.

5.8. CWE-401 – missing release of memory after effective lifetime

This weakness, also called a *Memory Leak*, occurs when allocated memory is not released after it has been used, and which slowly consumes the remaining memory. This error is often caused by improper handling of malformed data or unexpectedly interrupted sessions. It can also be caused by confusion over which part of the program is responsible for freeing the memory. This error can cause denial of service, and excessive resource consumption (both CPU and memory). Additionally, this issue can be hard to detect and fix quickly since the effect can take some time to show itself. This CWE has been identified as the underlying cause of several vulnerabilities, like CVE-2005-3119 and CVE-2022-38177.

We found that most techniques to detect memory leaks operate on the source code and many have a dynamic component. We did not find any approach that is both binary-level and uses only static analysis. Binary-level techniques to detect this error often conduct static analysis to identify and insert instrumentation points, which then monitor the code at run-time to detect or prevent leaks.

Andrzejak et al. introduced an intriguing machine learning approach to detect memory leaks [86]. This is a source-level technique where they instrument the `malloc` and `free` calls in C/C++ programs to gather data on allocated memory fragments, their lifetimes, and sizes to compute feature vectors. These properties were then used to train a machine learning classifier to detect memory leaks.

A number of approaches instrument program binaries to detect memory leaks during program execution. For example, Trishul et al. presented a tool named SWAT that instruments the program binary to trace memory allocation and free requests [87]. The profiling is used to construct a heap model and to monitor load/stores to allocated objects with low overhead. They monitor the staleness of each object and check if relevant instructions have been executed to predict memory leaks.

In another work, Koizumi et al. presented the BIGLeak algorithm that performs dynamic binary analysis to group objects based on their allocation context and monitors each group's size using I/O-based snapshots [88]. Their detection algorithm incorporates intermittency analysis, enabling the rapid identification of both low and high-risk leaks. When combined with dynamic binary analysis using context-aware execution sampling, they claim to achieve low run-time overheads. They demonstrated nearly 100% precision in detecting leaks when real-world software was employed.

Popular binary instrumentation frameworks, like Valgrind and DynamoRio also provide tools, called Memcheck [55] and Dr. Memory [89], respectively to detect memory leaks. Other tools to detect this error include Electric Fence [90], mtrace [91], PurifyPlus [92], and Deleaker [93].

6. Evaluation of open-source tools

In the previous section we surveyed the static techniques and tools that were developed to detect the most common weaknesses in binary software. Along with publishing their work, it is now not uncommon for researchers to also release an open-source version of their implementation on platforms such as *github*. To better understand and evaluate the performance of these techniques on a common set of benchmark programs, we attempted to find, build, and test each technique implementation, if available, for our benchmark set that was previously described in Section 3. We also contacted the authors if we encountered issues during this process.

Of the ten most common binary-level weaknesses we survey in this paper, we do not find any open-source implementation of techniques to detect the *improper input validation (CWE-20)* and *access of resource using incompatible type (CWE-843)* software errors. Therefore, we do not include them in this section. In the remainder of this section we report our findings on the availability, status, and performance of the open-source implementations of the techniques for the other common software errors.

6.1. Buffer overflow detection

In this section, we report our findings on the availability and ability of open-source tools for buffer overflow detection for software binaries. As mentioned previously, the top CWEs that correspond to buffer overflow detection include CWE-787, CWE-125 and CWE-119.

We found that many of the released tools for buffer overflow detection did not work as documented or as expected. Dahl et al. (2020) published an open-source implementation of their published work [52]. The released software scripts were designed to compile datasets of functions with potential vulnerabilities, which were then fed into an RNN model for classification. We found that the underlying code and workflow were relatively easy to understand. Likewise, the provided datasets and results were readily accessible. However, when we attempted to utilize our independent benchmarks, we encountered many difficulties. We discovered that the programs required source-level modifications that went against our primary focus on analyzing unmodified binaries. For instance, functions without arguments couldn't be used. Also, analyzing large programs was deemed impractical according to the authors.

Baradan et al. have released an open-source implementation of their symbolic execution based technique [37]. This tool required an installation of the angr framework, which we installed. However, this tool too had limitations when source code was unavailable. Most notably, functions with default constant data or void datatype were not analyzed and needed to be modified for evaluation. This prevented us from using this tool to assess other benchmarks, as it stalled during the symbolic execution step.

Another open-source tool released by Xu et al. appeared to have limited maintenance, as the associated GitHub page was inactive [54]. Not surprisingly, this software required non-trivial installation steps. We encountered challenges due to dependency issues, necessitating the downgrading of Angr and other dependencies for specific library versions. Even after sustained effort, we were unable to successfully execute this tool.

BAP and CWE_checker are the only binary-level static buffer overflow detection open-source tools that worked for us. Additionally, for comparison of binary-level and source-level approaches to buffer overflow detection, we used a popular source-code analysis tool, named Sonarqube [29]. Sonarqube is a commercial tool that also provides a free cloud-based service.

The results of our evaluation on the SARD (SARD-88 and SARD-89), Juliet, and SPEC benchmarks are shown in Table 4. The SARD and Juliet suites contain small programs with ground-truth results. Programs in the SARD benchmark are available in different categories based on their size and type (bad and benign, min, med, and large). The Juliet Test Suite has over 3000 programs. SPEC benchmarks are larger real-world programs, but do not provide a ground truth. Therefore, we use the results from the source-level Sonarqube as the baseline results for the SPEC benchmarks.

Thus, our results reveal that both the binary-level tools offer poor buffer overflow detection accuracy with many false positives and false negatives, even for the small SARD and Juliet benchmarks. Surprisingly, even the source code analyzer, Sonarqube, is not able to detect all of the weaknesses performing just slightly better than binary tools in some cases. BAP shows decent performance for heap overflow detection, but it does not support buffer overflow detection in other memory regions.

Overall, we find that there is a scarcity of reliable and effective open-source buffer overflow detection tools for binary analysis. Our evaluation revealed significant limitations and challenges associated with the available open-source tools. The tools exhibit poor accuracy, limited applicability to real-world scenarios, and challenges in installation and maintenance.

Table 4
Evaluation of open-source tools to detect Buffer Overflows.

Buffer Over	Ground	CWE_checker			BAP			Sonarqube		
Benchmarks	Truth	TP	FP	FN	TP	FP	FN	TP	FP	FN
SARD 88_bad	14	2	2	10	0	0	14	2	0	12
SARD 88_benign	14	12	2	2	0	0	14	12	2	0
SARD 89_min	291	1	0	290	0	0	291	183	0	108
SARD 89_med	291	1	0	290	0	0	291	183	0	108
SARD 89_large	291	9	0	282	0	0	291	182	0	109
SARD 89_benign	291	291	0	0	0	0	291	291	0	0
Juliet Stack	3198	331	573	2867	0	0	3198	561	0	2637
Juliet Heap	3870	1470	3349	2400	517	21	3353	535	0	3335
SPEC	399*	1	160	398	7	0	392	399*	0	0

Table 5
Evaluation of open-source tools to detect Use After Free defect.

Benchmarks	Ground Truth	CWE_checker			BAP			Sonarqube		
UAF		TP	FP	FN	TP	FP	FN	TP	FP	FN
Juliet Test Suite	394	17	18	377	96	42	298	333	0	61
spec	10*	0	2	10	1	306	9	10*	0	0

6.2. CWE-416 – use after free

In this section, we review and test open-source binary-level static analysis based tools to detect use-after-free errors.

The GUEB tool developed by Josselin Feist [94] is one such tool that is available on GitHub. However, it has not been actively maintained for a long time. We found that the tool's installation process involves numerous dependencies, making it challenging to use for evaluation purposes.

We again use BAP and cwe_checker to determine the accuracy of existing state-of-the-art tools to detect UAF errors. We did not find any other open-source binary-level tools to detect UAF that worked for us. The Juliet test-suite includes many UAF benchmarks. Additionally, we also use the bigger SPEC 2017 benchmarks to conduct our evaluation. We use the Sonarqube source-level SAST tool to compare the performance of the binary-level tools and use the Sonarqube results as the baseline for the SPEC benchmarks.

Our evaluation results are displayed in Table 5. Thus, we can see that while Sonarqube with access to the source-code performs well, it is not fully accurate in detecting use after free vulnerabilities. The BAP tool demonstrated better accuracy than cwe_checker in detecting UAF vulnerabilities in the Juliet benchmarks. The cwe_checker tool only detected 17 out of 394 total cases in the Juliet test suite. Additionally, the false positives were notably higher for the BAP tool, while the false negatives were significantly higher for cwe_checker.

In summary, our evaluation findings indicate that the accuracy of all the analyzed tools is limited when assessing use after free vulnerabilities in both small and large benchmarks. False positives and false negatives are prominent, highlighting the need for improved algorithms and tools to increase the accuracy and effectiveness in detecting UAF weaknesses.

6.3. CWE-476 – null pointer dereference

In this section, we review and test open-source binary-level static analysis based tools to detect null pointer dereferences. We found that most researchers did not make their works public or maintain their repositories. Therefore, again, the BAP toolkit and cwe_checker were the only tools available to use to evaluate NPD detection for binaries. We again use Sonarqube to compare the results from the binary-level tools and use the Sonarqube results as the ground truth for SPEC benchmarks.

Table 6 shows the results of our evaluation of the binary-level open-source tools for NPD detection on program binaries. For the Juliet test suite, cwe_checker detected 186 out of 306 NPD cases, while the BAP toolkit outperformed all other tools by detecting 240 cases. However, both tools had a significant number of false positives, with cwe_checker having around half of the total cases as false positives. The BAP toolkit exhibited a lower false positive rate of 80. Both cwe_checker and the BAP toolkit achieved the same detection rate of 13 out of 200 NPD cases for the larger SPEC benchmarks. However, cwe_checker had a significantly higher false positive rate.

Interestingly, the source code analyzer, Sonarqube, did not perform well in NPD detection with about half of the cases being correctly identified. Thus for NPD detection, binary analysis based tools outperformed our source code based tool in terms of detection accuracy.

6.4. CWE-190 – integer overflow or wraparound

CWE-190 Integer Overflow is a widely recognized issue with numerous detection techniques proposed. Yet, the error is challenging to identify statically as it can be deeply embedded within a program and may only become evident under specific input conditions. We found CWE_checker to be the sole open-source tool that uses static techniques to detect integer overflows in software binaries.

Our evaluation results for CWE_checker are displayed in Table 7. The evaluation was conducted using the Juliet test suite benchmark. Almost 4000 cases were analyzed, of these only 252 were successfully detected. Interestingly, Sonarqube only detected 100 of these instances of CWE-190. Thus, the prevalence of a large number of false negatives even in the small Juliet programs suggests that existing approaches may not be sufficiently accurate for effectively identifying this issue.

6.5. CWE-798 – use of hardcoded credentials

Although both Sonarqube and BAP claim to detect this error, both these tools were unable to detect any cases in our set of Juliet and SPEC benchmarks. We found that BAP only checks for the hard-coded socket addresses, but does not support the detection of hard-coded passwords. One likely reason for the inability of existing tools to detect this error is that detection of the hard-coded passwords in the form of character strings is difficult as it can detect all other strings as the potential credentials resulting in a large number of false positives.

Table 6
Evaluation of open-source tools to detect Null Pointer Dereference defect.

Benchmarks	Ground Truth	Cwe_checker			BAP			Sonarqube		
NPD		TP	FP	FN	TP	FP	FN	TP	FP	FN
Juliet Test Suite	30	186	147	120	240	80	66	180	0	126
SPEC	200*	13	228	187	13	19	187	200*	0	0

Table 7
Evaluation of open-source tools to detect Integer Overflows.

Benchmarks	Ground Truth	Cwe_checker			Sonarqube		
Integer Overflow		TP	FP	FN	TP	FP	FN
Juliet Test Suite	3960	252	0	3708	100	0	2860
SPEC	0*	0	53	0	0	0	0

Table 8
Evaluation of open-source tools to detect Memory Leaks.

Benchmarks	Ground Truth	valgrind			Sonarqube		
Memory leak		TP	FP	FN	TP	FP	FN
Juliet Test Suite	565	98	0	467	357	83	208
SPEC	22*	13	3	9	22*	0	0

6.6. CWE-401 – missing release of memory after effective lifetime

We did not find a static analysis based binary-level approach or open-source tool to detect this weakness. Therefore, in this section, we compare the detection accuracy of one popular binary-level memory detection tool (Valgrind's memcheck) with that of the source-level SAST tool, Sonarqube. Our results with the Juliet test suite and SPEC benchmarks are displayed in Table 8. We find that Valgrind was only able to detect about 17% of errors correctly for the small Juliet benchmarks. Sonarqube performs better with an accuracy of around 63% for the same benchmarks. Thus, new techniques and tools may be necessary to improve the detection accuracy of this bug, especially for program binaries.

7. Discussion

Static analysis techniques are often used during code analysis and inspection. However, they often suffer from a high number of false positives and false negatives [95]. Lipp et al. reported that even state-of-the-art source-level static analysis tools that produce accurate results on small programs still miss a significant percentage of vulnerabilities in real-world benchmarks, ranging from 47% to 80% [15]. Most of these studies were conducted with source-level static code analysis techniques or tools.

Intuitively, this challenge is exacerbated when statically analyzing binary programs due to the loss of crucial program information, including types, names, and high-level code syntax and structure in binary codes. Yet, binary analysis is important in several contexts when source code is either unavailable (viruses and other malware, or third-party code) or is lost (old/legacy binaries), or to examine the actual program that runs on a machine after compiler optimizations. Unfortunately, the prevalence of false positives and false negatives has led to an under-utilization of static analysis tools and techniques due to the costs associated with further manual inspection [96]. Our evaluation in this work confirms that these issues regarding accuracy and scalability persist in current state-of-the-art tools. Resolving this challenge is one of the most critical future frontiers for binary-level static analysis research.

In this study, we find that several researchers are currently addressing this challenge by exploring combinations of static techniques with other methods. Many recent works show a shift towards integrating static analysis with other techniques. Some studies have combined static and dynamic analysis to mitigate weaknesses in both approaches [36,48,53], while others have leveraged machine learning and deep learning alongside static techniques, dynamic fuzzing, and symbolic execution [50–52,54,56]. This trend reflects the recognition that static techniques alone have limitations and can benefit from synergies with other complementary code analysis methods.

8. Limitations and threats to validity

Our current study has several limitations that should be considered when interpreting the results. Firstly, our evaluation of binary-level static analysis tools was constrained by the availability of open-source options. Several techniques have proprietary implementations and the codes are not available online. Additionally, even some open-source projects have outdated dependencies and unmaintained repositories that make it challenging to evaluate those techniques. In Table 9, we report the other open-source tools that we attempted to employ, but failed to build and/or use in this study for different reasons. Thus, we were unable to assess many of the tools and techniques that have been reported in the literature.

Secondly, our benchmark suite used to evaluate the performance of binary-level static analysis tools has a few limitations. The SARD and Juliet test-suites provide a ground truth regarding errors and vulnerabilities in the codes. However, these are small programs that may not

Table 9
Summary of Vulnerabilities and Tools.

Vulnerability	Tool	Issue
Buffer Overflow	RNN for Vulnerability Detection [52]	Only works with provided source code, not others.
	UBSYM [37]	Tool installs but does not run on our benchmarks, requires source code change.
	bofAEG [54]	Tool installs, but none of our benchmarks finish execution due to path explosion issue.
Use After Free	GUEB [94]	Code is not maintained, Installation errors due to dependencies.
	UBSYM [37]	Installs but does not run on our benchmarks, requires source code change.
Null Pointer Dereference	No other open-source tools found.	
Integer Overflow or Wraparound	No other open-source tools found.	
Use of Hardcoded Credentials	No other open-source tools found.	
Memory Leak	No static source-open tools found.	

be representative of real-world programs. In contrast, the larger SPEC programs lack ground truth information for assessing the presence of vulnerabilities. We rely on the source-level analyzer, Sonarqube, to provide a ground truth for the SPEC benchmarks. However, as we have mentioned before, source-level analyzers are not completely accurate either.

Finally, our focus on binaries produced by the C/C++ languages limits the generalizability of our findings to other languages and binaries compiled using multiple languages. Future research should aim to address these limitations by incorporating a wider range of tools, benchmarks, and language-specific analyses.

9. Future work

There are many avenues for future work. First, we plan to explore error categories beyond the top 10 CWEs for program binaries. We also plan to review dynamic and run-time error detection approaches to complement our static analysis focus from this work.

Second, a major finding from this work is that there is a lack of open-source research and state-of-the-art tools to accurately detect the important CWEs in program binaries. Our future goal is to learn from existing approaches to construct such an open-source tool to precisely detect errors in binary software. Likewise, we plan to develop tools to detect Common Weakness Enumeration (CWE) vulnerabilities that currently lack dedicated tools or research, such as CWE-843 (Access of Resource Using Incompatible Type) and CWE-798 (Use of Hard-coded Credentials).

Third, we plan to develop advanced deobfuscation and decompilation techniques that can handle obfuscated and optimized code more effectively, aiming to recover higher-level abstractions from low-level binary representations.

Fourth, developing static analysis techniques that can handle executables built using different languages and multiple architectures, such as ARM, x86, and others, can also be one of the crucial future works that enable comprehensive analysis of modern software systems. We plan to undertake this research in the future.

Fifth, improving the accuracy of binary-level static analysis tools remains a critical challenge in the field. Enhancing true positives while reducing false positives is essential for reliable vulnerability detection. An emerging approach involves combining the results of static analysis with other techniques, such as run-time monitoring or machine learning algorithms. This integration has shown promise in improving overall accuracy and reducing false positives by leveraging complementary strengths.

Finally, it is crucial to understand how inaccuracies in binary-level static analysis can impact dependent tasks like Control-Flow Integrity (CFI). Identifying and mitigating these impacts will be essential for ensuring the effectiveness of security mechanisms that rely on accurate static analysis results.

10. Conclusions

Our goal in this work was to comprehensively review and compare past research in static analysis based approaches to detect the most important CWE categories for program binaries. Another major goal was to evaluate the accuracy of open-source tools built to detect each studied program's weaknesses. We made many significant, interesting, and novel discoveries and observations in this work. First, we found that we currently lack tools and techniques to accurately detect many important classes of errors in binary software. Second, we found that much research is not available in the open-source domain, and even the tools that exist are often not maintained and lack critical support. Third, many research works only evaluate their techniques on small benchmarks, and their results may not adequately represent performance in real-world applications. Fourth, many CWE detection techniques suffer from a high incidence of false positives and false negatives, underscoring the need for refinement and enhancement of existing techniques and tools. Thus, this work distinguishes itself as the first survey of binary-level CWE detection techniques, and the first independent assessment of binary-level open-source tools for identifying software weaknesses, offering valuable insights and setting the stage for further advancements in this critical field.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Ashish Adhikari: Writing – review & editing, Supervision, Software, Project administration, Methodology, Investigation. **Prasad Kulkarni:** Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization.

Acknowledgements

This work is sponsored in part by the National Security Agency (NSA) Science of Security Initiative.

References

[1] N. N. V. Database, Cvss severity distribution over time, (https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time, Retrieved June 9, 2023).

[2] C. (Cybersecurity, I. S. A. Advisory, AA22-117A: conti ransomware, (https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-117a). Retrieved May 17, 2023.

[3] CISA (Cybersecurity and Infrastructure Security Agency), AA22-216A: conti ransomware, (https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-216a). Retrieved May 17, 2023.

[4] L. Szekeres, M. Payer, T. Wei, D. Song, Sok: eternal war in memory, in: 2013 IEEE Symposium on Security and Privacy, 2013, pp. 48–62, doi:10.1109/SP.2013.13.

[5] S.C. Johnson, M. Hill, Lint, a c program checker (1978). https://api.semanticscholar.org/CorpusID:59749883.

[6] M. Byun, Y. Lee, J.-Y. Choi, Analysis of software weakness detection of cbmc based on cwe, 2020, pp. 171–175, doi:10.23919/ICACT48636.2020.9061281.

[7] M. Saletta, C. Ferretti, A neural embedding for source code: security analysis and cwe lists, in: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2020, pp. 523–530, doi:10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00095.

[8] D.S. Cruzes, M.L. Chaim, D.S. Santos, What do we know about buffer overflow detection? a survey on techniques to detect a persistent vulnerability, Int. J. Syst. Softw. Secur. Protect. 9 (3) (2018) 1-33, doi:10.4018/IJSSSP.2018070101.

[9] S.J. Ahmed, D.B. Taha, Machine learning for software vulnerability detection: a survey, in: 2022 8th International Conference on Contemporary Information Technology and Mathematics (ICCITM), 2022, pp. 66–72, doi:10.1109/ICCITM56309.2022.10031734.

[10] V. Yosifova, A. Tasheva, R. Trifonov, Predicting vulnerability type in common vulnerabilities and exposures (cve) database with machine learning classifiers, 2021, doi:10.1109/ELECTRONICA52725.2021.9513723.

[11] J.D. Pereira, M.P.A. Vieira, On the use of open-source c/c++ static analysis tools in large projects, 2020 16th European Dependable Computing Conference (EDCC) (2020) 97–102.

[12] S. Zaharia, T. Rebedea, S. Trausan-Matu, Cwe pattern identification using semantical clustering of programming language keywords, 2021, pp. 119–126, doi:10.1109/CSCS52396.2021.00027.

[13] T. Ji, Y. Wu, C. Wang, X. Zhang, Z. Wang, The coming era of alphahacking?: a survey of automatic software vulnerability detection, exploitation and patching techniques, in: 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), 2018, pp. 53–60, doi:10.1109/DSC.2018.00017.

[14] F. Alenezi, C. Tsokos, Machine learning approach to predict computer operating systems vulnerabilities, 2020, pp. 1–6, doi:10.1109/ICCAIS48893.2020.9096731.

[15] S. Lipp, S. Banescu, A. Pretschner, An empirical study on the effectiveness of static c code analyzers for vulnerability detection, Association for Computing Machinery, New York, NY, USA, 2022, doi:10.1145/3533767.3534380.

[16] G. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: a survey, Proc. IEEE 108 (10) (2020) 1825–1848, doi:10.1109/JPROC.2020.2993293.

[17] K. Goseva-Popstojanova, A. Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Inf. Softw. Technol. 68 (2015) 18–33,

doi:10.1016/j.infsof.2015.08.002. https://www.sciencedirect.com/science/article/pii/S0950584915001366
[18] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, Sok: (state of) the art of war: offensive techniques in binary analysis, in: 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 138–157, doi:10.1109/SP.2016.17.
[19] H. Xue, S. Sun, G. Venkataramani, T. Lan, Machine learning-based analysis of program binaries: a comprehensive study, IEEE Access 7 (2019) 65889–65912, doi:10.1109/ACCESS.2019.2917668.
[20] National Security Agency, Ghidra, (GitHub repository). https://github.com/NationalSecurityAgency/ghidra, Retrieved May 13, 2023.
[21] R. Team, Radare2 github repository, (https://github.com/radare/radare2, Retrieved May 13, 2023).
[22] Hex-Rays, IDA Pro, (https://hex-rays.com/ida-pro/, Retrieved May 3), 2023.
[23] Cifuentes, Cristina and Levin, Mark and Ramos, Jaime and others, BAP (binary analysis platform), (https://github.com/BinaryAnalysisPlatform/bap). Retrieved May 13, 2023.
[24] Dyninst Development Team, Dyninst, (GitHub repository). https://github.com/dyninst/dyninstRetrieved May 13, 2023.
[25] A.C. Eberendu, V.I. Udegbe, E.O. Ezennorom, A.C. Ibegbulam, T.I. Chinebu, A systematic literature review of software vulnerability detection, Eur. J. Comput. Sci. Inf. Technol. 10 (1) (2022) 23–37.
[26] SPEC CPU2017, (Standard Performance Evaluation Corporation). https://www.spec.org/cpu2017/, Retrieved March 2, 2023.
[27] National Institute of Standards and Technology (NIST), Software assur- ance reference dataset (SARD) benchmarks, NIST Software Assurance Metrics And Tool Evaluation (SAMATE). https://samate.nist.gov/SARD/test-suites/89.
[28] National Institute of Standards and Technology (NIST), Juliet Test Suite, (NIST Software Assurance Metrics And Tool Evaluation (SAMATE)b). https://samate.nist.gov/SARD/test-suite/JULIET.html.
[29] SonarQube, (https://www.sonarsource.com/products/sonarqube/). Retrieved January 4, 2023.
[30] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: a survey for roadmap, ACM Comput. Surv. 54 (11s) (2022), doi:10.1145/3512345.
[31] P. Godefroid, Fuzzing: Hack, art, and science, Commun. ACM 63 (2) (2020) 70-76, doi:10.1145/3363824.
[32] P. Thomson, Static analysis, Commun. ACM 65 (1) (2021) 50-54, doi:10.1145/3486592.
[33] T. Muske, A. Serebrenik, Survey of approaches for postprocessing of static analysis alarms, ACM Comput. Surv. 55 (3) (2022), doi:10.1145/3494521.
[34] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385-394, doi:10.1145/360248.360252.
[35] W. Wang, M. Fan, A. Yu, D. Meng, Bofsanitizer: efficient locator and detector for buffer overflow vulnerability, in: 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), 2021, pp. 1075–1083, doi:10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00168.
[36] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, D. Feng, Towards efficient heap overflow discovery, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 989–1006. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jia.
[37] S. Baradaran, M. Heidari, A. Kamali, M. Mouzarani, A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes, Int. J. Inf. Secur. 22 (2023) 1–14, doi:10.1007/s10207-023-00691-1.
[38] T. Yavuz, C. Brant, Security analysis of iot frameworks using static taint analysis, in: Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy, in: CODASPY '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 203-213, doi:10.1145/3508398.3511511.
[39] D. Boxler, K.R. Walcott, Static taint analysis tools to detect information flows, in: Proceedings of the Int'l Conf. Software Eng. Research and Practice (SERP'18), 2018.
[40] GrammaTech Inc., GrammaTech, (Official Website).https://www.grammatech.com/ Retrieved May 1, 2023.
[41] E.J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: 2010 IEEE Symposium on Security and Privacy, 2010, pp. 317–331, doi:10.1109/SP.2010.26.
[42] A. Aumpansub, Z. Huang, Learning-based vulnerability detection in binary code, in: 2022 14th International Conference on Machine Learning and Computing (ICMLC), in: ICMLC 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 266-271, doi:10.1145/3529836.3529926.
[43] S.M. Ghaffarian, H.R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey, ACM Comput. Surv. 50 (4) (2017), doi:10.1145/3092566.
[44] N.S. Harzevili, A.B. Belle, J. Wang, S. Wang, Z. Ming, Jiang, N. Nagappan, A survey on automated software vulnerability detection using machine learning and deep learning, 2023, 2306.11673
[45] Open Web Application Security Project (OWASP), OWASP Buffer Overflow, (OWASP). https://owasp.org/www-community/vulnerabilities/Buffer_Overflow Retrieved May 13, 2023.
[46] B.M. Padmanabhuni, H.B.K. Tan, Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning, in: 2015 IEEE 39th Annual Computer Software and Applications Conference, volume 2, 2015, pp. 450–459, doi:10.1109/COMPSAC.2015.78.
[47] University of Waikato, Weka Data Mining Tool, (Weka Wiki). https://waikato.github.io/weka-wiki/, Retrieved August 20, 2023.
[48] B.M. Padmanabhuni, H.B. Kuan Tan, Auditing buffer overflow vulnerabilities using hybrid static-dynamic analysis, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, 2014, pp. 394–399, doi:10.1109/COMPSAC.2014.62.
[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: PLDI '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 190-200, doi:10.1145/1065010.1065034.
[50] T. Gao, X. Guo, Buffer overflow vulnerability location in binaries based on abnormal execution, in: 2020 4th Annual International Conference on Data Science and Business Analytics (ICDSBA), 2020, pp. 29–31, doi:10.1109/ICDSBA51020.2020.00015.
[51] L. He, Y. Cai, H. Hu, P. Su, Z. Liang, Y. Yang, H. Huang, J. Yan, X. Jia, D. Feng, Automatically assessing crashes from heap overflows, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 274–279, doi:10.1109/ASE.2017.8115640.
[52] W.A. Dahl, L. Erdodi, F.M. Zennaro, Stack-based buffer overflow detection using recurrent neural networks, 2020, 2012.15116.
[53] I. Gotovchits, R. Van Tonder, D. Brumley, Saluki: finding taint-style vulnerabilities with static property checking, in: Proceedings of the NDSS Workshop on Binary Analysis Research, volume 2018, 2018.
[54] S. Xu, Y. Wang, L. Coppolino, Bofaeg: automated stack buffer overflow vulnerability detection and exploit generation based on symbolic execution and dynamic analysis 2022 (2022), doi:10.1155/2022/1251987.
[55] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Not. 42 (6) (2007) 89-100, doi:10.1145/1273442.1250746.
[56] N.-E. Enkelmann, T. Barabosch, CWE Checker, (GitHub repository). https://github.com/fkie-cad/cwe_checker Retrieved May 1, 2023.
[57] L. Braz, E. Fregnan, G. Çalikli, A. Bacchelli, Why don't developers detect improper input validation? '; drop table papers; –, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 499–511, doi:10.1109/ICSE43902.2021.00054.
[58] Y. Zhang, Z. Wang, W. Yu, B. Fang, Multi-level directed fuzzing for detecting use-after-free vulnerabilities, in: 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2021, pp. 569–576, doi:10.1109/TrustCom53373.2021.00087.
[59] K. Zhu, Y. Lu, H. Huang, Scalable static detection of use-after-free vulnerabilities in binary code, IEEE Access PP (2020) 1–1, doi:10.1109/ACCESS.2020.2990197.
[60] J. Feist, Finding the Needle in the Heap: Combining Binary Analysis Techniques to Trigger Use-After-Free, Université Grenoble Alpes, 2017 Ph.D. thesis. https://theses.hal.science/tel-01681707v2/document
[61] H. Yan, Y. Sui, S. Chen, J. Xue, Machine-learning-guided typestate analysis for static use-after-free detection, Association for Computing Machinery, New York, NY, USA, 2017, doi:10.1145/3134600.3134620.
[62] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, M. Lemerre, Binary-level directed fuzzing for Use-After-Free vulnerabilities, in: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), USENIX Association, San Sebastian, 2020, pp. 47–62. https://www.usenix.org/conference/raid2020/presentation/nguyen.
[63] OWASP, Null dereference, https://owasp.org/www-community/vulnerabilities/Null_Dereference Retrieved May 10, 2023.
[64] W. Jin, S. Ullah, D. Yoo, H. Oh, Npdhunter: efficient null pointer dereference vulnerability detection in binary, IEEE Access 9 (2021) 90153–90169, doi:10.1109/ACCESS.2021.3091209.
[65] T. Cloosters, M. Rodler, L. Davi, TeeRex: discovery and exploitation of memory corruption vulnerabilities in SGX enclaves, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 841–858. https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters
[66] A. Vishnyakov, D. Kuts, V. Logunova, D. Parygina, E. Kobrin, G. Savidov, A. Fedotov, Sydr-Fuzz: continuous hybrid fuzzing and dynamic analysis for security development lifecycle, in: 2022 Ivannikov ISPRAS Open Conference (ISPRAS), IEEE, 2022, pp. 111–123, doi:10.1109/ISPRAS57371.2022.10076861.
[67] Google, Google OSS-Fuzz: Continuous Fuzzing for Open Source Software, (https://github.com/google/oss- fuzz, Retrieved July 12), 2023.
[68] L. Project, LibFuzzer - LLVM 13 documentation, 2023, https://llvm.org/docs/LibFuzzer.html, Retrieved July 12, 2023.
[69] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, AFL++ : combining incremental steps of fuzzing research, 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Association, 2020. https://www.usenix.org/conference/woot20/presentation/fioraldi
[70] The MITRE Corporation, CWE-680: Integer Overflow to Buffer Overflow (IO2BO), (MITRE CWE). https://cwe.mitre.org/data/definitions/680.htmlRetrieved July 20, 2023.
[71] T. Wang, T. Wei, Z. Lin, W. Zou, Intscope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution, 2009.
[72] T. Wei, J. Mao, W. Zou, Y. Chen, A new algorithm for identifying loops in decompilation, in: H.R. Nielson, G. Filé (Eds.), Static Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 170–183.
[73] C. Bauer, A. Frink, R. Kreckel, Introduction to the ginac framework for symbolic computation within the c++ programming language, ArXiv cs.SC/0004015 (2000).
[74] V. Ganesh, D.L. Dill, A decision procedure for bit-vectors and arrays, in: Proceedings of the 19th International Conference on Computer Aided Verification, in: CAV'07, Springer-Verlag, Berlin, Heidelberg, 2007, p. 519-531.
[75] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, C. Eckert, Intrepair: informed repairing of integer overflows, IEEE Trans. Softw. Eng. 47 (10) (2021) 2225–2241, doi:10.1109/TSE.2019.2946148.

[76] Z. Huang, X. Yu, Integer overflow detection with delayed runtime test, 2021, pp. 1–6, doi:10.1145/3465481.3465771.

[77] B. Zhang, C. Feng, B. Wu, C. Tang, Detecting integer overflow in windows binary executables based on symbolic execution, in: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016, pp. 385–390, doi:10.1109/SNPD.2016.7515929.

[78] Veracode, (https://www.veracode.com/). Retrieved May 1, 2023.

[79] Checkmarx, (https://checkmarx.com/). Retrieved April 4, 2023.

[80] GitGuardian, (https://www.gitguardian.com/). Retrieved May 10, 2023.

[81] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, E. van der Kouwe, Typesan: practical type confusion detection, in: CCS '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 517-528, doi:10.1145/2976749.2978405.

[82] G.J. Duck, R.H.C. Yap, Effectivesan: type and memory error detection using dynamically typed c/c++, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 181-195, doi:10.1145/3192366.3192388.

[83] X. Fan, S. Long, C. Huang, C. Yang, F. Li, Accelerating type confusion detection by identifying harmless type castings, in: Proceedings of the 20th ACM International Conference on Computing Frontiers, in: CF '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 91-100, doi:10.1145/3587135.3592205.

[84] Y. Jeon, P. Biswas, S. Carr, B. Lee, M. Payer, Hextype: efficient detection of type confusion errors for c++, in: CCS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 2373-2387, doi:10.1145/3133956.3134062.

[85] D. Kim, S. Kim, Bintyper: type confusion detection for c++ binaries, BlackHat Europe, 2020. https://www.blackhat.com/eu-20/briefings/schedule/bintyper-type-confusion-detection-for-c-binaries-21351

[86] A. Andrzejak, F. Eichler, M. Ghanavati, Detection of memory leaks in c/c++ code via machine learning, in: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2017, pp. 252–258, doi:10.1109/ISSREW.2017.72.

[87] M. Hauswirth, T.M. Chilimbi, Low-overhead memory leak detection using adaptive statistical profiling, SIGPLAN Not. 39 (11) (2004) 156-164, doi:10.1145/1037187.1024412.

[88] Y. Koizumi, Y. Arahori, Risk-aware leak detection at binary level, in: 2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC), 2020, pp. 171–180, doi:10.1109/PRDC50213.2020.00028.

[89] D. Bruening, Q. Zhao, Practical memory checking with dr. memory, in: International Symposium on Code Generation and Optimization (CGO 2011), 2011, pp. 213–223, doi:10.1109/CGO.2011.5764689.

[90] ElectricFence, (https://github.com/kallisti5/ElectricFence, Retrieved March 4), 2023.

[91] mtrace, (https://man7.org/linux/man-pages/man3/mtrace.3.html, Retrieved May 3), 2023.

[92] PurifyPlus, (https://www.ibm.com/docs/en/announcements/archive/ENUS204-063, Retrieved May 3), 2023.

[93] Deleaker, (https://www.deleaker.com/, Retrieved May 3), 2023.

[94] J. Feist, GUEB: a static analyzer performing use-after-free detection on binary, 2018, (https://github.com/montyly/gueb Retrieved July 12), 2023.

[95] I. Elkhalifa, B. Ilyas, Static code analysis: a systematic literature review and an industrial survey, 2016,

[96] T. Muske, A. Serebrenik, Survey of approaches for postprocessing of static analysis alarms 55 (3) (2022), doi:10.1145/3494521.