BinPool: A Dataset of Vulnerabilities for Binary Security Analysis

Sima Arasteh*
University of Southern California
Los Angeles, CA, USA
arasteh@usc.edu

Georgios Nikitopoulos*
Dartmouth College
Hanover, NH, USA
University of Thessaly
Volos, Thessaly, Greece
Georgios.Nikitopoulos.GR@dartmouth.edu

Wei-Cheng Wu
Dartmouth College
Hanover, NH, USA
Wei-Cheng.Wu.GR@dartmouth.edu

Nicolaas Weideman
USC Information Sciences Institute
Los Angeles, CA, USA
nhweideman@gmail.com

Aaron Portnoy
Dartmouth College
Hanover, NH, USA
aaron.portnoy@dartmouth.edu

Mukund Raghothaman
University of Southern California
Los Angeles, CA, USA
raghotha@usc.edu

Christophe Hauser
Dartmouth College
Hanover, NH, USA
christophe.hauser@dartmouth.edu

Abstract

The development of machine learning techniques for discovering software vulnerabilities relies fundamentally on the availability of appropriate datasets. The ideal dataset consists of a large and diverse collection of real-world vulnerabilities, paired so as to contain both vulnerable and patched versions of each program. Naturally, collecting such datasets is a laborious and time-consuming task. Within the specific domain of vulnerability discovery in binary code, previous datasets are either publicly unavailable, lack semantic diversity, involve artificially introduced vulnerabilities, or were collected using static analyzers, thereby themselves containing incorrectly labeled example programs.

In this paper, we describe a new publicly available dataset which we dubbed `BinPool`, containing numerous samples of vulnerable versions of Debian packages across the years. The dataset was automatically curated, and contains both vulnerable and patched versions of each program, compiled at four different optimization levels. Overall, the dataset covers 603 distinct CVEs across 89 CWE classes, 162 Debian packages, and contains 6144 binaries. We argue that this dataset is suitable for evaluating a range of security analysis tools, including for vulnerability discovery, binary function similarity, and plagiarism detection.

CCS Concepts

• **Security and privacy** → **Software security engineering**.

Keywords

Binary analysis, vulnerability dataset

---

*Equal contributions.

1 Introduction

Developing and evaluating software vulnerability detection tools—particularly those that rely on machine learning [1, 2, 4, 5, 10, 11, 19, 23, 25]—fundamentally depends on the existence of robust, labeled datasets of known vulnerabilities. Assembling these datasets is often among the hardest parts of building these systems.

We argue that there are severe limitations with existing datasets: *First*, well-annotated and easy-to-compile datasets such as Juliet [14] are limited to small programs and artificially introduced bugs. The lack of diverse, real-world code examples leads to fears of overfitting and limited generalization in trained models. *Next*, many previous papers do not make their data publicly available [5, 8, 9, 22]. *In addition*, many publicly available datasets focus on binary similarity detection, rather than on vulnerability finding [23]. *Finally*, some datasets such as those assembled by Pereira et al. [18] rely on warnings reported by program analysis tools such as Flawfinder [24] and Cppcheck [13]. Naturally, this results in possibly tainted ground truth, leading to fears of incorrectly trained classifiers.

In contrast, the ideal dataset contains a large, diverse collection of programs with different and well-annotated vulnerability types. These programs should represent examples of real-world vulnerabilities, and have a matched instances of vulnerable and patched code. Furthermore, especially for vulnerability discovery at binary level, it is important to be able to compile code into executable files.

In this paper, we report on a new dataset of binaries with historical real-world vulnerabilities named `BinPool`. This is a collection of high-impact vulnerabilities obtained by crawling and collating the National Vulnerability Database (NVD) [17] with data from the Debian security tracker [3] and the archive of Debian snapshots [20].

We have already used an early version of this dataset to evaluate BinHunter [2], a graph neural network-based binary vulnerability detection system [2]. Going forward, we believe that this dataset will be very useful for practical security research.

At a high level, we assembled the dataset as follows: We mapped each CVE in the NVD database to its associated CWE, Debian packages that fix the vulnerability, and the specific functions and lines that were edited as part of the patch. We then wrote scripts to automatically build these Debian packages. By using the Debian package maintenance tool quilt to selectively apply and withhold the patch from the source code, we obtained vulnerable and patched versions of each binary. We then repeated this process at four different optimization levels. The entire process is highly automated, and currently contains 603 distinct CVEs spanning 89 different CWE categories, across 162 Debian packages, and with 6144 binaries in total. Overall, the dataset spans 910 source functions and 7280 binary functions respectively.

We note that BinPool can be used for both vulnerability discovery and binary function similarity detection. In the case of vulnerability discovery, it is applicable to both source and binary code. It includes a range of vulnerabilities, from highly specific categories such as CWE-122 (Stack-based buffer overflow) to wider categories such as CWE-119 (Improper restriction of operations with the bounds of a memory buffer). Notably, the dataset contains detailed information about the precise location of each vulnerability, including files, functions, and lines affected by the patch, both at the source and binary levels. The automation scripts and links to the dataset may found at https://github.com/SimaArasteh/binpool.

The rest of this paper is organized as follows: We describe the BinPool collection process in Section 2, the overall structure of the dataset in Section 3, and its potential applications in Section 4 respectively. In Sections 5 and 6 we describe the related work and discuss the limitations of our approach.

2 Dataset Construction

We show the workflow of the BinPool curation process in Figure 1. Its construction is enabled by three resources provided by the Debian project: First, Debian packages are structured with control files, metadata and content, and are maintained by the community through updates and security patches. Second, Debian Snapshots [20] contains an archive of historical package versions, allowing access to specific releases over time. Finally, the Debian Security Tracker [3] monitors vulnerabilities such as CVEs, enabling users and maintainers to stay informed about security issues. We use the beautifulsoup library to crawl data from these resources.

2.1 Vulnerability Data Collection

The Debian Security Tracker [3] provides a frequently updated JSON file with CVE-IDs and version information about the fixed packages.[1] We gather the relevant version numbers and recover the corresponding CWE categories by consulting the NVD database [17]. We then use the Debian Snapshots archive to gather a link to the package source code. We record this information in a publicly available Google Spreadsheet.

2.2 Package Build Process

Next, Debian offers an automatic system for building packages using the build-dep and dpkg-buildpackage tools. The build-dep tool installs the necessary dependencies and the dpkg-buildpackage tool automatically compiles Debian packages from their source code. Our automation system leverages these Debian tools to fully streamline the package building process.

We retrieve source code of the fixed versions of each package from Debian Snapshots. Each package includes a directory, debian/patches, which contains a sequential list of patches to be applied to the source code. We use the quilt tool to selectively apply or remove the specific patches that fixed the vulnerability. Finally, we build each variant (buggy / patched) of the package at four different optimization levels, including with debug symbols.

2.3 Metadata Extraction

The last conceptual step is to extract metadata information from these packages. This information includes files, functions, and source and binary locations that were modified by the patch.

The package build process results in a number of deb files being produced as output. Informally, deb files are used to distribute and install Debian packages. We extract and search through these packages to locate the individual binaries (ELF files) that were affected by the patch. Now: to extract the metadata, we parse the patch file and recover the files and modules that were edited as part of fixing the vulnerability. We then parse the modules in the Debian source using the clang compiler front-end to extract the relevant function names. Finally, we obtain file names, function names and lines modified as part of the patch.
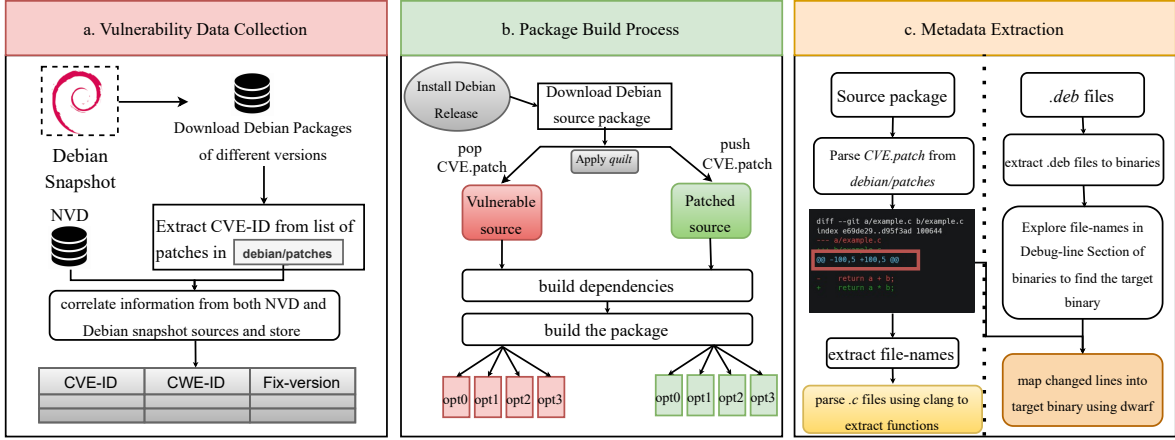
In order to find the target binary among extracted deb files, we map file names in the patch into the corresponding compilation units in the binary using the debug line section in DWARF. Recall that each compilation unit includes a source file and associated headers that are compiled together to produce a single object file. This mapping process allows us to determine which binary contains the patched file or compilation unit. After finding the target binary, we once again use debug information embedded in the binary to extract the precise memory offsets corresponding to the lines of code modified in the patch. All steps in this dataset construction pipeline are fully automated and written in Python.
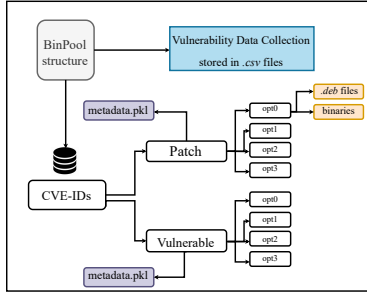
3 Structure of the BinPool Dataset

There are three principal components in BinPool dataset: The metadata (stored in pkl files), the binaries in question, and a central CSV file containing information about CVEs, CWEs, version numbers, and links to source code. We also gather all the metadata in a JSON file called binpool_info.json. The links to these resources may be found in the main artifact repository: https://github.com/SimaArasteh/binpool. We show the layout of these directories in Figure 2. To the best of our knowledge, BinPool is the first such dataset to provide this level of detailed information. We also present some aggregate statistics about the dataset in Table 1.

4 Possible Applications

The immediate intended application of BinPool is in the development and evaluation of vulnerability detection techniques. We

---

[1]https://security-tracker.debian.org/tracker/data/json

Figure 1: Construction process of the `BinPool` dataset. (*a*) In the first phase, we collect data about the vulnerabilities by gathering CVE-IDs, CWEs, and the affected and fixed package versions from the Debian snapshots and NVD databases. (*b*) In the second phase, we build packages for both vulnerable and patched versions. (*c*) In the last phase, we extract detailed metadata, including function names and vulnerability locations (both at source and binary levels).

Figure 2: Structure of the `BinPool` dataset. The central CSV file contains information about CVE and CWE-IDs, version numbers, and links to source code. In parallel, the dataset is organized according to the vulnerability IDs. Each vulnerability includes metadata about the function names, module names, and affected code locations in `pkl` files, and versions of the vulnerable and patched `deb` files and binaries obtained from different optimization levels.

Table 1: Aggregate statistics about `BinPool`.

Measurement	Value
Number of unique CVEs	603
Number of CWEs	89
Number of Debian packages	162
Total number of binaries	6144
Total number of source modules	768
Total number of source functions	910
Total number of binary functions	7280

expect the collection of functions with diverse real-world semantics to form a challenging dataset and for the framework to be an ongoing benchmark collection technique for bug-finding tools. An early smaller-scale version of the dataset already formed a signficant part of our evaluation methodology for BinHunter [2].

Beyond just machine learning techniques, we expect the dataset to be useful for benchmarking other program analysis systems such as angr [21]. We also note that many vulnerability detection tools are based on sophisticated reasoning pipelines involving data flow and control flow analysis and information about types of vulnerabilities. As such, many of these intermediate analyses—such as precise inter-procedural data flow analyses in binaries—are of independent research interest, and we expect `BinPool` to also form a possible benchmarks for these applications.

Finally, beyond just vulnerability discovery, we expect `BinPool` to be useful in other binary analysis problems. As one example, recall that we provide multiple versions of each binary package, compiled using different optimization levels. These matched binaries might form a good benchmark for code search and function similarity detection algorithms.

5 Related Work

In this section, we review the history of vulnerability datasets at both the source code and binary levels. We compare BinPool features with state-of-the-art datasets for both source code and binaries with different applications in Table 2.

5.1 Source-Level Datasets

In part because of their relative ease of collection, most existing datasets of vulnerable code are at the source level. One particularly famous example is the Juliet dataset [14], which (among other languages) provides a collection of C and C++ programs with artificially injected vulnerabilities. The dataset provides macros to

Table 2: A comparison between existing datasets and BinPool. VD, CP and BSD stand for vulnerability discovery, compiler provenance and binary similarity detection respectively.

Dataset	source-level	binary-level	Application	# CVEs	# projects
CrossVul	Yes	-	VD	5131	1675
ReposVul	Yes	-	VD	6,134	1,491
BigVul	Yes	-	VD	3754	348
MegaVul	Yes	-	VD	8,254	992
BinBench	-	Yes	BSD, CP	-	131
BinaryCorp	-	Yes	BSD	-	9819
BinKit	-	Yes	BSD	-	51
BinPool	-	Yes	BSD,VD	603	162

switch between vulnerable and non-vulnerable versions of code. As such, because of its scale and ease of use, it has been used to train and evaluate numerous program analysis tools.

Of course, one important complaint against the Juliet dataset is that it includes artificial, as opposed to real-world examples of bugs. One convenient approach to do this (and similar to our approach in `BinPool`) is to use the NVD dataset to identify examples of actual vulnerabilities in production open-source code [16]. As such, this provides a reliable indicator of ground truth. Another alternative to create these datasets [12] is to identify vulnerabilities using static analysis tools such as Cppcheck [13] and Flawfinder [24].

Another aspect of datasets is the granularity of information provided about the location of the vulnerability. For example, datasets such as Bigvul [6], Megavul [15], and Crossvul [16] provide large collections of vulnerable source code with precise information about the software fault. These are typically extracted by using commit information. Crossvul covers over 40 programming languages, whereas Bigvul and Megavul focus specifically on vulnerabilities in C/C++ programs. Compared to Bigvul, Megavul encompasses more vulnerabilities and open-source projects, and it includes information on vulnerable functions using a Tree-sitter parser. While Crossvul is more diverse in covering multiple programming languages, it lacks detailed function information.

Among these datasets, Reposvul stands out by offering a repository-level dataset. This dataset aims to address three issues in existing collections. First, many patches are not strictly security-related and are mixed with non-security changes. To address this, Reposvul uses large language models (LLMs) and static analysis tools to identify security-specific patches. Second, most datasets focus only on function-level vulnerabilities, overlooking the importance of inter-procedural vulnerability analysis. Reposvul addresses this by capturing relationships between functions involved in a patch. Lastly, it identifies outdated patches by tracking commit histories.

5.2 Binary-Level Datasets

While there are many diverse source-level vulnerability datasets, binary-level datasets are limited. The main challenge is that compiling programs from source code frequently requires considerable manual effort. Our solution to this challenge in `BinPool` is to use the existing build system supplied as part of Debian to obtain a scalable and well-tested build system for a large repository of packages.

Existing binary-level vulnerability datasets face several challenges. Firstly, many of these datasets are not publicly accessible. Secondly, most are derived from a small set of open-source

projects, compiled across various optimization levels and architectures. These datasets are primarily designed for detecting similarities between different architectures and optimization levels, rather than focusing specifically on vulnerability discovery. Additionally, the limited range of projects they include can lead to overfitting in machine learning models.

For example, the Genius dataset [7] includes only 154 vulnerable functions sourced from BusyBox, OpenSSL, and Coreutils, compiled across three architectures, four optimization levels, and two compilers. The Vulseekerpro dataset [8] features only 15 unique CVEs, compiled with different optimization settings. Among the various datasets of binary programs, Jtrans [23] is the most diverse, featuring programs compiled from multiple projects. However, it is specifically developed for binary similarity detection and is limited to identifying particular CVEs across different optimization levels.

6 Limitations

Although our dataset includes a diverse and comprehensive collection of real-world vulnerabilities with detailed metadata, it still has only a number of CVEs per CWE category. As a result, although the dataset is ideal for *evaluating* vulnerability detection tools, it might be insufficient for *training* classifiers. As a consequence, the training process might need to be supplemented with other sources of information, including datasets such as Juliet, in a manner similar to our work on BinHunter [2].

A second limitation of `BinPool` is that it was collected purely using information about modifications to source code, and therefore does not include some important sources of information about the errors in question. For example, one might be able to augment the data with examples of failing test cases, error traces resulting from symbolic execution engines such as angr [21], or include more detailed information about inter-procedural data flows. Of course, collecting such information would require significant extensions to our data collection pipeline, but might conceivably lead to more sophisticated vulnerability detection tools.

7 Conclusion

In this paper, we have introduced a public dataset of historical vulnerabilities in Debian packages. The dataset provides detailed information, including CVE and CWE identifiers, version numbers, and lists of the functions, files, and lines modified as part of the fix, at both source and binary levels. The dataset is suitable for both developing and evaluating both vulnerability discovery and binary function similarity tools. In particular, we envision `BinPool` to be appropriate as a test set for evaluating machine learning-based techniques, and can potentially also be utilized with program analysis tools such as angr.

We intend to continuously run the data collection pipeline and grow the dataset over time. We also plan to include more detailed information, such as the results of inter-procedural data flow analyses, thereby capturing how vulnerabilities arise from the interaction of multiple functions within each program. We hope that this information leads to more effective and better evaluated techniques for vulnerability discovery.

References

[1] Shushan Arakelyan, Sima Arasteh, Christophe Hauser, Erik Kline, and Aram Galstyan. Bin2vec: learning representations of binary executable programs for security tasks. *Cybersecurity*, 4:1–14, 2021.

[2] Sima Arasteh, Jelena Mirkovic, Mukund Raghothaman, and Christophe Hauser. Binhunter: A fine-grained graph representation for localizing vulnerabilities in binary executables . In *Proceedings of the 2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2024.

[3] Debian Security Team. Debian security tracker. https://security-tracker.debian.org/, 2025. Accessed: 2025-01-13.

[4] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp)*, pages 472–489. IEEE, 2019.

[5] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. Discovre: Efficient cross-architecture identification of bugs in binary code. In *Ndss*, volume 52, pages 58–79, 2016.

[6] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[7] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 480–491, 2016.

[8] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 803–808, 2018.

[9] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 896–899, 2018.

[10] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pages 85–96, 2016.

[11] Yuede Ji, Lei Cui, and H Howie Huang. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 702–715, 2021.

[12] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 544–555, 2022.

[13] D. Marjamaki. Cppcheck. http://cppcheck.sourceforge.io/, May 2007. Accessed: 2025-01-13.

[14] FGG Meade. Juliet test suite v1. 2 for c/c++ user guide. *no. December*, 2012.

[15] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. Megavul: Ac/c++ vulnerability dataset with comprehensive code representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 738–742. IEEE, 2024.

[16] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1565–1569, 2021.

[17] National Institute of Standards and Technology (NIST). National vulnerability database (nvd), 2023. https://nvd.nist.gov/.

[18] José D'Abruzzo Pereira, João Henggeler Antunes, and Marco Vieira. A software vulnerability dataset of large open source c/c++ projects. In *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 152–163, 2022.

[19] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[20] Debian Project. Debian package tracker – snapshot service, 2023. https://snapshot.debian.org/.

[21] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[22] Yunlong Song. Genius: Generic neural shape analysis, 2019. https://github.com/Yunlongs/Genius.

[23] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2022.

[24] David A. Wheeler. Flawfinder. https://github.com/david-a-wheeler/flawfinder. Accessed: 2025-01-13.

[25] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 363–376, 2017.