Building a Commit-level Dataset of Real-world Vulnerabilities

Alexis Challande
achallande@quarkslab.com
Quarkslab
Paris, France
Inria
Palaiseau, France
Institut Polytechnique de Paris
Palaiseau, France

Robin David
rdavid@quarkslab.com
Quarkslab
Paris, France

Guénaël Renault
guenael.renault@ssi.gouv.fr
ANSSI
Paris, France
Inria
Palaiseau, France
Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

While Common Vulnerabilities and Exposures (CVE) have become a *de facto* standard for publishing advisories on vulnerabilities, the state of current CVE databases is lackluster. Yet, CVE advisories are insufficient to bridge the gap with the vulnerability artifacts in the impacted program. Therefore, the community is lacking a public real-world vulnerabilities dataset providing such association.

In this paper, we present a method restoring this missing link by analyzing the vulnerabilities from the Android Open Source Project (AOSP), an aggregate of more than 1,800 projects. It is the perfect target for building a representative dataset of vulnerabilities, as it covers the full spectrum that may be encountered in a modern system where a variety of low-level and higher-level components interact. More specifically, our main contribution is a dataset of more than 1,900 vulnerabilities, associating generic metadata (e.g. vulnerability type, impact level) with their respective patches at the commit granularity (e.g. fix commit-id, affected files, source code language).

Finally, we also augment this dataset by providing precompiled binaries for a subset of the vulnerabilities. These binaries open various data usage, both for binary only analysis and at the interface between source and binary. In addition of providing a common baseline benchmark, our dataset release supports the community for data-driven software security research.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Vulnerability management*.

KEYWORDS

security vulnerabilities; dataset; vulnerability research; patch detection; binary matching

ACM Reference Format:
Alexis Challande, Robin David, and Guénaël Renault. . Building a Commit-level Dataset of Real-world Vulnerabilities. In *Proceedings of Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3508398.3511495

1 INTRODUCTION

The CVE standard is used to publish vulnerability advisories through a common and universal mechanism. CVE databases [12, 23] present an overview of all the vulnerabilities in any system. To encompass such variety, they offer a set of features (CVE-ID, Common Vulnerability Scoring System (CVSS), description) and a method to describe impacted systems with Common Platform Enumeration (CPE) but no standardized way to report the corresponding fix. Data-driven vulnerability research relies on the availability of such information [8]. We aim to fill this gap by bringing together the vulnerabilities and their fixes in a unique dataset.

Android – one of the most deployed operating system in the world – makes it the perfect target to create such a dataset. It runs on heterogenous devices like smartphones, car ECUs, smart devices or household appliances. Although some parts of the operating system are proprietary and closed, most of the system is open sourced through the AOSP[1]. It is an aggregate of more than 1800 projects ranging from widely known projects tospecific developments for a particular device. The projects cover the full spectrum of the functionalities that may be encountered in a modern system where a variety of components interact together (system tools, media framework, drivers and more).

Android is subject to security issues, and Google publishes monthly security bulletins listing CVE fixed in the last update. While the bulletins primarily target end users, they are also an information goldmine from a security perspective. Each bulletin contains a list of fixes affecting the operating system coming from three sources, AOSP, the Linux kernel and system-on-chip manufacturers. For every vulnerability entry, the CVE-ID, type, severity and the impacted AOSP versions are also listed. More importantly, it provides a direct link to the fixing commits for AOSP's projects.

We leveraged this data to create the most exhaustive dataset of real-world vulnerabilities targeting a system as a whole. Previous datasets contain either synthetic code [7], hand-crafted vulnerabilities [9], or unrelated vulnerabilities in various software [16, 19]. Our dataset, with a commit-level precision, includes vulnerabilities targeting different languages (C, C++, Java). It brings coherence between vulnerabilities by aggregating those of a whole system.

We pushed our dataset a step further by providing pre-compiled binaries in both vulnerable and fixed state for a subset of these vulnerabilities (≈ 600 vulnerabilities). This enables binary-based Source Code Analysis Tools (SAST), Dynamic Application Security Testing Tools (DAST) or CVE checkers to assess their capabilities at binary level.

Overall, our main contributions are:

CODASPY '22, April 24–27, 2022, Baltimore, MD, USA.
. https://doi.org/10.1145/3508398.3511495

[1] https://android.googlesource.com/

- A dataset of more than 1,900 real-world vulnerabilities (based on CVEs) providing for each of them valuable metadata:
  - the vulnerability type (remote code execution, elevation of privilege);
  - the impacted programming language;
  - the patched and vulnerable versions at the exact commit-level.
- We augment this dataset by pre-compiling a subset of those vulnerabilities (600) to offer compiled binaries for multiple architectures (`ARM`, `x86`, `X86_64`, `ARM64`).
- We make our data publicly available to the community to enable other researchers to test their research on broader data.

2 RELATED WORK

Previous vulnerability-related test suites and datasets have already been introduced in the literature. This section discuss previous datasets, their limitations and how our dataset addresses them.

The National Institute of Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE) project is dedicated to improve software assurance. The Juliet Test Suite [6, 7] is a collection of 64,000 synthetic test cases for the C/C++ language, testing 118 different Common Weakness Enumerations (CWEs). As part of standards, this test suite is extensively used by security vendors but suffers from its synthetic nature as generated programs are small and relatively simple.

In the Lava [10] test suite, tailored for testing fuzzers, bugs are injected in different execution pathes of binaries utilities from `coreutils`. Lava only injects out-of-bounds memory accesses, not reflecting the complexity of real-world software bugs. The Cyber Grand Challenge (CGC) [9] sample set provides more bugs types, but the sample programs are relatively small and too simple to efficiently cover the wide range of existing binaries.

Instead of considering synthetic test-suites, the National Vulnerability Database (NVD) lists security issues affecting real-world software. Akram and Ping [3] presented a dataset of vulnerable functions and files gathered using NVD links to fix commits. Besides being discontinued, their dataset only lists vulnerable files and functions. This makes it hard to recover the context around those modifications and the fixed version is unavailable.

Li *et al.* [19, 20] created their dataset using vulnerabilities from some open-source projects. They used heuristics to detect the fix commit for a vulnerability. Thus, this information is computed while our dataset obtains it from its source, making it more reliable.

Vulnerable clone detection tools such as VFDETECT [22], VCCFinder [25] or ReDeBug [16] also use a vulnerability dataset. Their dataset only contains some vulnerabilities for their tests and is unpublished, preventing its reusability.

Nappa *et al.* [24] studied the impact of shared code on vulnerability patching, but as they relied on CPE information, they were unable to be more precise than a version granularity.

Finally, other approaches use information from publicly disclosed vulnerabilities like the CVE benchmark from the Open Security Software Foundation [4], the work from Ponta *et al.* [26] or CVEFixes [5]. All these works are complementary with ours. For example, the CVE-benchmark targets vulnerabilities languages unsupported by our dataset (JavaScript/TypeScript), and only 3% of vulnerabilities found in CVEFixes are overlaping with our dataset.

Table 1: Components Example from Bulletins

Type	Categories example
General	Framework, Media Framework, System, ...
SoC	Qualcomm Model, MediaTek Kernel, ...
Linux	Kernel ALSA, Kernel USB, ...

Table 2: August 2021 Bulletin Extract

CVE	References	Type	Severity	Updated AOSP Version
CVE-2021-0640	A-123455677	EoP	High	9, 10, 11
CVE-2021-0645	A-123455677	EoP	High	11
CVE-2021-0646	A-123455677	EoP	High	8.1, 9, 10, 11

3 ANDROID CVE DATA AGGREGATION

3.1 Bulletin Format

Since 2015, Google publishes monthly security bulletins [14] for Android. A bulletin is usually divided into two Security Patch Levels (SPLs), themselves divided into categories to facilitate the understanding of the bulletin: examples of categories are listed in Table 1. Finally, each category contains the list of vulnerabilities, with their CVE identifier, and the fixing AOSP updates. Additional information (i.e. affected versions, vulnerability severity) are also present. Table 2 shows an extract of the August 2021 bulletin.

Vulnerabilities listed in the bulletins are coming from various sources: AOSP itself, the Linux kernel, which is embedded inside AOSP, and system on chip manufacturers. For vulnerabilities affecting AOSP components, the bulletin contains a direct link to the fix commit. Publishing bulletins is not a Google specificity and other manufacturers like Samsung or Qualcomm also do it.

3.2 Architecture

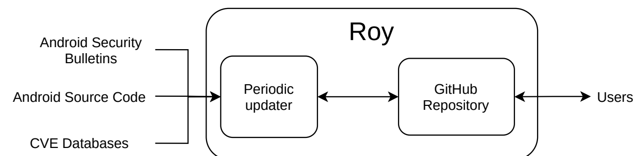

Figure 1: Roy's Architecture

We crawl Android Security Bulletins using a homemade tool named Roy. Its workflow is depicted in Figure 1. Roy starts by looking at the new bulletins since its last run. It is an iterative process and we never attempt to parse again previously analyzed bulletins. For each bulletin, Roy recovers the list of new vulnerabilities. When a link towards the fix commit exists, Roy also parses the changes provided by the commit (e.g. commit message, changed files...).

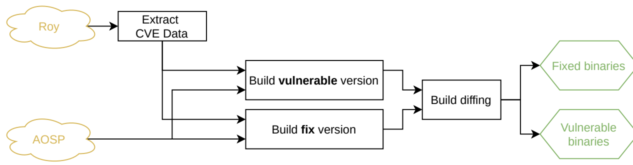

Figure 2: AOSP Builder

While thourough, Android Security Bulletins are not complete and additional information is available from other sources. We query CVE-Search [12], an open access CVE database, to augment the data for each CVE.

3.3 Pitfalls

Crawling bulletins and web interfaces is challenging since they are not designed for automated consumption. The main difficulty lies in the ever-changing bulletin format. This requires an almost monthly maintenance effort to be able to parse the latest data. However, since we store data on our side, we do not need to keep a parser for *every* bulletin, just the last ones.

Furthermore, data presented in bulletins are unstable and fields are subject to change, usually restricting the information available. For instance, bulletins after June 2017 remove the `discovered date` field. To keep this information, we query it from another source such as CVE-Search.

4 AUTOMATED AOSP BUILDING

4.1 Motivation

Vulnerabilities affect both open-source and closed-source projects where only binaries are available. To develop and test binary binary-level SAST, DAST or CVE-checkers, having a dataset also containing binararies could prove itself useful.

AOSP is also a perfect target to provide pre-compiled binaries for the vulnerabilities: it is both open-source and presents a documented build system, while having commit-precise data.

4.2 Architecture

We developed an automated AOSP builder. It compiles the binaries at the commits just before and just after the vulnerability fix. Having binaries differing by a single commit, the one responsible for the fix, allows to pinpoint the changes induced by the fix at the binary level. The builder is depicted in Figure 2. As input, it uses a fixing commit id from Roy, builds the project with the fixing commit and without it. Finally, we only keep the binaries that differ between the two builds.

4.3 Building AOSP at a Specific Commit

Building AOSP at a specific commit is challenging for the following reasons. First, building projects in the past is difficult, as we have to reconstruct the whole chain of dependencies needed by the project (compiler version, libraries used...) at that time. AOSP self-containment, for both its toolchain and all its projects dependencies, makes the environment setup less difficult. However, finding precisely each dependencies version remains challenging. A second problem arises from either bogus commits preventing compilation of the project or fixes applied to former development branches. In this case, we cannot obtain binaries differing by precisely one commit and we use a different strategy (see Section 4.5). Finally, AOSP is a huge project of more than 80 GiB of source files. Building a single version of AOSP generates another 80 GiB of binary files in approximately 3 hours on a 56-core machines.

Our process to build vulnerabilities starts by looking at the latest branch including the fixing commit and setting up all AOSP's projects to be at this version. This allows us to have a building environment working for AOSP in a whole and all the dependencies of the project. Then, we use two strategies to maximize the number of successful builds described in Section 4.5.

4.4 Build Diffing

A project may have multiple output targets, and AOSP's projects depend on numerous projects themselves. Thus, building a single project may produce several binaries in the output directory. However, we are only interested in targets changed between the vulnerable and the fixed version. Since compilation is resource intensive, build systems implement an optimization, compiling for a target only if one of its build-dependencies has changed since the last run. We leverage this property for our build diffing system.

4.5 Vulnerability Building Strategies

We use two strategies to build the binaries of a project in both the vulnerable and fixed versions.

The first one is to checkout the project at a vulnerable commit, e.g. a parent of the fix commit, build the project, then checkout to the fix commit itself and rebuild the project. This is the preferred strategy as it is the most precise. However, it is unsuited if the compilation of the project in its vulnerable state fails.

The second strategy tries the opposite approach. It builds the project at the version state (one from Android) then reverts the fixing commit on the project before compiling again in a vulnerable state. If reverting the commit succeeded, this strategy works well.

4.6 Dataset Artifacts

For each vulnerability, we consider the build process to be complete if it produces the relevant binaries in both forms (fixed and vulnerable) for each of the four architectures (`x86`, `x86_64`, `arm`, `arm64`). We also keep the binaries with debug symbols (i.e. unstripped) if they are available.

Finally, we use heuristics relying on `ctags universal` [2] to guess the names of every function affected by a patch. These heuristics are unreliable for edge cases. Because parsing source code is done without running the preprocessor, functions affected by changes in macro are undetected. More importantly, compiler optimizations (e.g. function inlining, tail-call) may change the function layout and merge affected functions inside others.

Figure 3 presents an extract of the artifact for a precompiled vulnerability. We prefix every file by its `SHA256` hash to prevent name collisions.

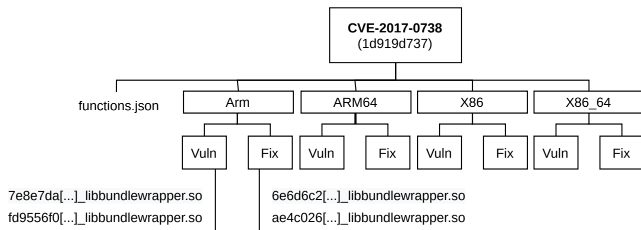

Figure 3: Extract of Artifacts for `CVE-2017-0738`

5 DATASET OVERVIEW

5.1 Commit-level Dataset

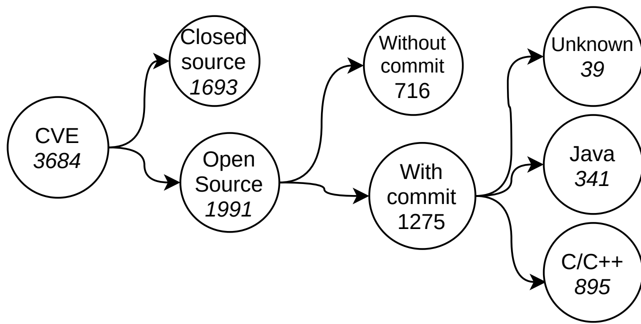

Figure 4: Dataset Vulnerabilities Repartition

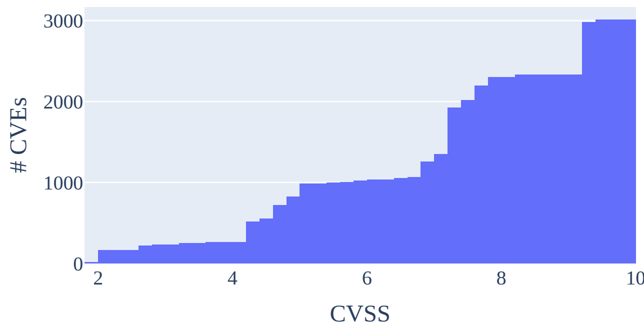

Figure 5: Cumulative CVSS Scores

Our dataset contains 3,684 CVEs detailed in Figure 4. The closed source vulnerabilities affect Qualcomm, NVIDIA or Google components. We retrieved the fix commit id for 1,275 open source vulnerabilities (64%). We support gitiles [15], the platform used by Google for the versionning of AOSP, while other platforms such as GitHub, kernel.org or CodeAurora would need additional engineering effort. 35% of the vulnerabilities present in our dataset have a CVSS score of at least 9.0. Their CVSS score cumulative distribution is plotted on Figure 5.

For vulnerabilities with a fix-commit id, 70% of them affect a C-code source and 27% target Java code. The remaining 39 are fixes impacting other types of files (e.g. XML).

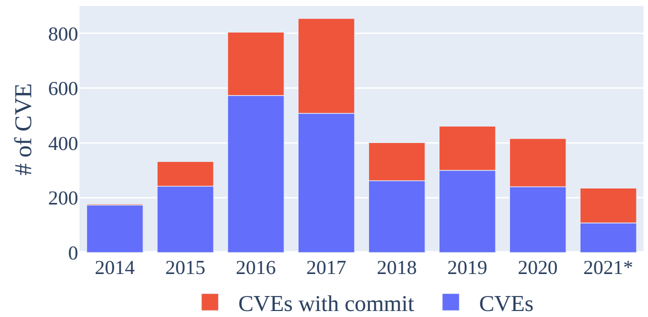

Figure 6: CVE Evolution Over Time

Figure 6 shows the CVE distribution over the years. We considered bulletins from August 2015 to September 2021, but we used only the CVE identifiers and not the bulletin dates nor the report dates. Some earlier CVEs may thus have been taken into account because they were fixed in subsequent patches. The number of reports decreases for closed source projects after 2017-2018, explaining the drop on the graph.

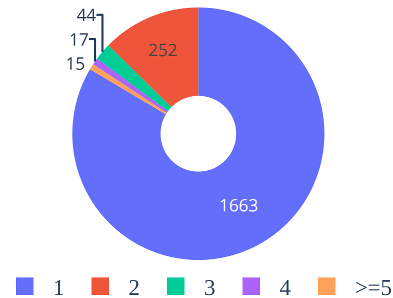

Figure 7: Fixing Commits Per Vulnerability

Most vulnerabilities are fixed by a single commit, as illustrated by Figure 7. This is the case for 84% of the dataset. A vulnerability may also be fixed by numerous commits. CVE-2015-3873 [1] affecting `libstagefright` in Android reports 20 fixing commits.

5.2 Precompiled Dataset

Among the 895 vulnerabilities targeting C/C++ code, we managed to propose precompiled binaries for 612 (68%) of them.

Based on file extensions, our dataset contains shared libraries (35%), object files (37%) and executables (13%). The exact repartition is depicted in Figure 8. The largest file is `libv8` (1.6 GiB), the static library of v8, a JavaScript engine. Several metrics are also listed in Table 3.

Table 3: Binaries Sizes in Dataset

Category	Mean	Median	Standard dev.
Unstripped	12.3 MiB	2.7 MiB	40.2 MiB
Stripped	17.7 MiB	623.8 KiB	128.2 MiB

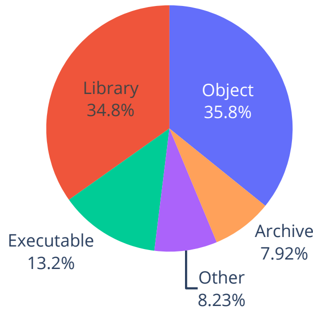

Figure 8: Binary Filetypes

Table 4 lists the top 10 most common files found in the dataset. They mostly represent complex part of the operating system (e.g. wireless communications, media management) particularly targeted by attackers. For instance, the most common file is `libbluetooth.so`, found 953 times.

Table 4: Most Common Binaries

Name	Count
libbluetooth.so	953
bluetooth.default.so	748
libnfc-nci.so	650
libstagefright.so	421
net_test_btif	417
net_test_stack	299
hevcdec	268
libaudioflinger.so	242
libbinder.so	234
libmedia.so	229

Figure 9 lists the time delta between the vulnerability publishing date and the fixing commit. Notably, only 24 vulnerabilities were fixed *after* the creation of the CVE entry, while 122 were fixed more than 6 months *before* the publication. The average vulnerability is fixed 86 days before its CVE entry.

6 DISTRIBUTION

Our dataset is publicly available on GitHub[2]. To ease data manipulation, we provide helper functions to interact with the dataset. Moreover, the precompiled dataset is also available online (117 GiB) and links are listed on the GitHub repository.

7 DISCUSSION

7.1 Applications

We provide this dataset to support data driven software security research. Examples listed below for each direction refer to existing research in the field. Nonetheless, we believe our dataset could enhance their works by having more data to test or reason about, while reducing friction as it is already compiled.

[2]https://github.com/quarkslab/aosp_dataset

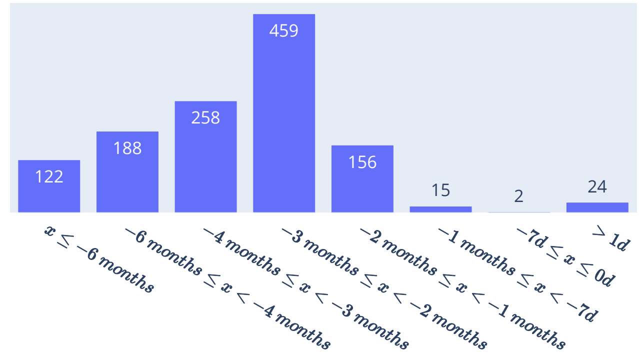

Figure 9: Delta Between the Fixing Commit and the Publication Date

Patch Characterization. Our extensive same system vulnerability set enables researchers to extract data aiming at characterizing what a patch is at both the source and binary level [13, 18]. Our data can help draw an identity card of a security patch or bootstrap on patches detection.

Silent Fix Detection. A silent fix is a commit fixing a security hole without publicity (e.g. a CVE entry, a changelog mention) [27, 32]. Silent patches create security risks for projects downstream as they are not aware of the importance of the new version. Being able to detect silent fixes could help maintainers to keep projects up to date, and reduce the risks for end users. Our dataset enables to train solutions both using the commit message (e.g. for Natural Language Processing algorithms) or analyzing the commits impact on the code.

Cross-architecture Binary Diffing/Matching. Our dataset provides the same binary for multiple architecture enabling further research on cross-architecture binary matching [28, 30], where an application maps two code snippets from different architectures together if they have the same semantics.

Moreover, some binaries in our dataset are present multiple times for different versions. They are interesting examples for cross-architecture binary diffing [11, 21], where the objective is to find a mapping between functions in the first binary to the second one.

Patch Detection. Our dataset makes a perfect base to test tools aiming at solving the *patch presence problem* [17, 29, 31]. By isolating the difference between the vulnerable and fixed version of the same binary, it enables the creation of signatures to detect if a target binary is patched.

7.2 Limitations

Android CVE Data Aggregation. An inherent limitation is the need for open source projects to retrieve the commit-id. Although open-source is prevalent in Android, many low-level components remain closed source. Our approach depends on the availability of security bulletins published by Google and their commitment in providing such information.

Data Quality. Our work implies that the commit referenced as a vulnerability fixing commit is complete, i.e. completely fixes the

vulnerability, and minimal, i.e. it does not fix any other problem nor add functionalities. Detecting if both assertions hold is out of scope for this paper and thus left unexplored.

Automated Compilation. Our automated compilation suffers from various problems and managed to compile only 612 vulnerabilities. Most issues stem from the synchronization problem between a project and its dependencies. However, each Android build has a manifest file listing the exact commit of each project during the compilation. Leveraging this information could help to solve the synchronization issues.

8 CONCLUSION

This paper shows how to build a dataset of real world representative vulnerabilities based on CVE identifiers leveraging Android Security Bulletins. This dataset is beneficial to the community by providing a common baseline vulnerability benchmark for further academic research and more robust vulnerability analysis or detection algorithms. We will continue collecting new data to keep our dataset synced with latest bulletins and CVEs.

The dataset is also augmented with more than 600 vulnerabilities precompiled for four architectures, enabling research on both source code, binary data or the interface between the two.

Our dataset is flexible so we can correlate it with other information sources or, at the opposite, tailor it for specific usages. Testing a SAST/DAST tool, a fuzzer, a sanitizer on a bug class (e.g. heap corruptions...), or conducting studies on vulnerability root causes or automatic repair are all possible usages of our dataset.

REFERENCES

[1] CVE-2015-3873. libstagefright in Android before 5.1.1 LMY48T allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted media file.

[2] Universal ctags. original-date: 2010-03-25T10:43:13Z.

[3] Junaid Akram and Luo Ping. How to build a vulnerability benchmark to overcome cyber security attacks. 14(1):60–71.

[4] Bas van Schaik and Kevin Backhouse. FPs are cheap. show me the CVEs!

[5] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software.

[6] Paul E. Black. A software assurance reference dataset: Thousands of programs with known bugs. 123:123005.

[7] Frederick Boland and Paul Black. The juliet 1.1 c/c++ and java test suite. (45). Publisher: Computer (IEEE Computer).

[8] Min-je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. End-to-end prediction of buffer overruns from raw source code via neural memory networks.

[9] DARPA. Cyber grand challenge.

[10] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE.

[11] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society.

[12] Alexandre Dulaunoy and Pieter-Jan Moreels. cve-search - a free software to collect, search and analyse common vulnerabilities and exposures in software.

[13] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities.

[14] Google. Android security bulletins.

[15] Google. gitiles - git at google.

[16] Jiyong Jang, Abeer Agrawal, and David Brumley. ReDeBug: Finding unpatched code clones in entire OS distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE.

[17] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE.

[18] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM.

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213. ACM.

[20] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection.

[21] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. αdiff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 667–678. ACM Press.

[22] Zhen Liu, Qiang Wei, and Yan Cao. VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint. In *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 548–553. IEEE.

[23] MITRE Corporation. MITRE.

[24] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy*, pages 692–708. IEEE.

[25] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 426–437. ACM Press.

[26] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cedric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE.

[27] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting "0-day" vulnerability: An empirical study of secret security patch in OSS. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. ISSN: 1530-0889.

[28] 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. pages 363–376.

[29] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. BinXRay: Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–387. ACM.

[30] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. Accurate and scalable cross-architecture cross-OS binary code search with emulation. 45(11):1125–1149.

[31] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902. USENIX Association.

[32] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 914–919. ACM Press.