ARVO: Atlas of Reproducible Vulnerabilities for Open-Source Software

Xiang Mei*, Jordi Del Castillo†, Pulkit Singh Singaria*, Haoran Xi†
Abdelouahab Benchikh*, Tiffany Bao*, Ruoyu Wang*, Yan Shoshitaishvili*
Adam Doupé*, Hammond Pearce‡, Brendan Dolan-Gavitt§
*Arizona State University, †New York University, ‡University of New South Wales, §XBOW
*{n132, psingari, tbao, fishw, yans, doupe}@asu.edu, am.benchikh@esi-sba.dz
†{jordi.d, hx759}@nyu.edu, ‡hammond.pearce@unsw.edu.au, §moyix@xbow.com

***Abstract*—Achieving reproducibility, quantity, and diversity in vulnerability datasets has long been viewed as an inherent three-way trade-off, where improving one dimension often comes at the cost of the others. In practice, reproducibility has been the dimension most often neglected. This has limited what can be automatically extracted from historical bug datasets, and has reduced their utility for downstream security research.**

In this work, we propose a method to produce a new security dataset which ensures reproducibility for diverse vulnerabilities at scale by identifying the key obstacles to large-scale bug reproduction and addressing them with general solutions. Using this method, we introduce full reproducibility to the largest open source software vulnerability dataset (OSS-Fuzz) and construct the ARVO dataset (an Atlas of Reproducible Vulnerabilities in Open-source software). ARVO is a large-scale dataset consisting of over 6,100 real-world vulnerabilities across 311 projects. Focusing on reproducibility, ARVO differs from existing datasets by providing each vulnerability in a form that can be consistently rebuilt, triggered, and analyzed across versions. Reproducibility also enables automatic identification of the corresponding patch for each vulnerability and supports direct interaction with vulnerabilities after code changes, capabilities that existing large-scale datasets do not provide. In our evaluation, ARVO successfully reproduces 81% of vulnerabilities and achieves 89.4% accuracy on the located patches. We also discuss ARVO's influence on both upstream practices and downstream security research.

1. Introduction

Vulnerabilities in software are both common and damaging: in 2024 alone, around 40,000 vulnerabilities were tracked by the National Vulnerability Database (NVD). The prevalence of these flaws has spurred decades of research in automated vulnerability discovery and remediation, but historically, evaluation datasets to measure and understand these advancements have been elusive and ad hoc [1].

Databases which track vulnerabilities to alert users about security issues with software, such as the NVD, are primarily designed for system maintainers so they can verify affected versions and apply patches for known vulnerabilities in currently deployed software. This mission does not naturally align with the needs of security research. The NVD, and other datasets like it, typically lack critical metadata required for vulnerability research, including triggering inputs, reproducible environments, and ground-truth remediation data. This hampers research: without a reproducible environment, it is difficult to measure the efficacy of dynamic analysis techniques, a lack of triggering vulnerability inputs hampers automated patch generation, and the verification of generated candidate patches is impossible without a ground truth reference.

Researchers have worked to produce a better dataset, but current approaches are insufficient. Synthetic techniques, such as those represented by LAVA [2] and challenge problems from the Cyber Grand Challenge [3], have questionable real-world overlap and implications.

Other techniques are more realistic, but require extensive manual effort leading to associated scaling problems. For example, despite over 3,600 hours of human work to reproduce reported CVEs, Mu et al. [4] only succeeded in reproducing a total of 368 vulnerabilities. Other manually-crafted datasets show even greater scaling limitations: Magma contains just 7 projects with 118 unique vulnerabilities [5]. This manual effort precludes continued generation of new data for these datasets, and when used as evaluation benchmarks, the datasets tend to become "stale" over time as researchers (perhaps unintentionally) tune their systems to eventually "overfit" on the data.

An emerging genre of dataset generation leverages large repositories of projects with historical vulnerability data to *automatically* generate vulnerability datasets. The most popular dataset is Google's OSS-Fuzz [6], which provides findings from continuous fuzzing of over 1,000 open-source projects. Unfortunately, repositories such as OSS-Fuzz were not designed to support this purpose: 63% of the historical, vulnerable versions of software we attempted in our comparative evaluation fail to build due to errors caused by dependency drift, build system bitrot, and various exotic issues unaddressed by current techniques. As a result, recent dataset generation techniques based on OSS-Fuzz, such as OSS-Fuzz-OSV [7], are able to reproduce only around 37% of vulnerabilities. Worse, even the vulnerability metadata in these datasets contains *errors*: our research found 1,518 cases of incorrect vulnerability data (e.g., crashes or fixes) out of 10,440 records in OSS-Fuzz, an error rate of 14.5%. OSS-Fuzz-OSV inherits these issues downstream, resulting in a lower-quality dataset.

Full reproducibility, the ability to rebuild vulnerabilities from source code and reliably trigger the intended crash, has been absent from prior OSS (Open Source

Software) security datasets. We found that reproducibility is valuable for providing a verifiable feedback loop to downstream security research, such as patch locating and automated patch generation with large language models. Building on this observation, we identify and solve the key challenges hampering building a full reproducible vulnerability datasets for security research. Based on the solutions, we built ARVO[1], the "Atlas of Reproducible Vulnerabilities." ARVO is both a framework designed to address the shortage of continually-updating, high-quality research vulnerability datasets and a comprehensive bug dataset in its own right. ARVO aims to achieve a high level of reproducibility across a large number of real-world projects and vulnerabilities, providing a robust set of real-world vulnerabilities for a research vulnerability dataset. We focus on memory-safety-related bugs in C/C++ projects that manifest as a crash on a proof-of-concept input. We choose C/C++ for its widespread use and the significant impact of these bugs, though our methodology naturally extends to other languages.

ARVO has the following key capabilities: (1) *Recompilation*: ARVO can rebuild historical versions of software afflicted by various vulnerabilities. (2) *Precise fix identification*: ARVO automatically locates the precise developer patch that fixes the vulnerability. (3) *Customization*: ARVO supports arbitrary modifications to reproduced vulnerabilities, enabling downstream applications such as the porting of vulnerabilities between software versions. (4) *Triggering inputs*: Each vulnerability has a proof-of-concept "triggering" input that can be used to test for the presence of the vulnerability. (5) *Accessibility*: We provide prebuilt container images for each vulnerability, allowing issues to be reproduced with a single command (e.g., `docker run -it arvo/example:42486945-vul arvo`).

ARVO works at a large scale, reproducing 6,138 vulnerabilities across 311 projects (81% of attempted reproductions). This represents a +44 percentage-point improvement over the state of the art ($\approx 2.19\times$) and, unlike the state of the art, effectively handles complex, multi-component software projects. Additionally, we automatically evaluated developer-provided patches and confirmed that 89.4% of these patches fix the corresponding vulnerability. By cross-comparing ARVO and its upstream, ARVO identified and reported over 300 that were *improperly* detected as fixed *and remained unpatched to the present day*, and more that were incorrectly detected as fixed and remained unpatched for years.

Our results demonstrate that, in addition to reliably reproducing a high-quality vulnerability dataset for vulnerability research, ARVO's efficacy can uncover system issues in upstream components. This has not gone unnoticed, and ARVO is already shaping both its upstream and its downstream. *Upstream*, Google, the maintainers of OSS-Fuzz, are currently finishing the process of integrating ARVO as OSS-Fuzz's vulnerability reproduction component. *Downstream*, ARVO served as a benchmark for *multiple* teams in DARPA's AI Cyber Challenge (AIxCC) [8], and academic efforts such as CyberGym [9] (an AI-agent cybersecurity benchmark) use ARVO dataset as a data source.

1. https://github.com/n132/arvo

Contributions. This paper makes the following contributions:

1) We propose a new form of security dataset: introducing reproducibility to security datasets to build verifiable research vulnerability datasets at scale and with diversity.
2) We identify the key challenges in improving reproducibility for research vulnerability datasets and demonstrate our methods for addressing and mitigating these issues.
3) We design ARVO, a system that introduces reproducibility to security datasets, enables recompilation of historical vulnerabilities, and identifies more precise patch commits.
4) We present the ARVO dataset, a reproducible, recompilable, and automatically updating dataset of over 6,000 real-world vulnerabilities in open-source C/C++ projects.

Additionally, to support open research, we made ARVO itself—the framework, evaluation infrastructure, images, and metadata—**open-source**, so that other researchers can build on our work. This includes 12,276 Docker images for reproducing each vulnerability (`vul` and `fix` versions for each of our 6,138 successful reproductions) and for re-compiling projects after any valid modifications of the source.

2. Background and Related Work

Security research depends heavily on security datasets. Two properties in particular—quantity and diversity—have long been recognized as essential as they enable more comprehensive evaluation. However, despite significant effort and the existence of large-scale vulnerability datasets, most vulnerabilities remain unusable, especially for binary-focused research.

For example, the recent auto-patching work PATCHAGENT [10] was evaluated on 178 vulnerabilities from 30 programs, while nearly 40,000 vulnerabilities were added to NVD in a single year. Prior work [4] highlighted the gap between the vast number of discovered vulnerabilities and the limited research-usable subset: reproducibility. By investing significant manual effort, the authors made 368 vulnerabilities usable for security research.

2.1. OSS-Fuzz and OSV

As the Internet's "critical infrastructure" [11], open-source software continues to gain prominence: as of Oct. 2024, GitHub hosted over 518 million [12] public repositories. However, this reliance on open-source software also amplifies the impact of security vulnerabilities in widely used dependencies. One of the most impactful efforts in this space is *OSS-Fuzz* [6], a large-scale fuzzing project launched by Google. Since its launch in 2016, OSS-Fuzz has continuously fuzzed more than 1,000 open-source projects, finding and helping to fix over 10,000 vulnerabilities as of August 2025. OSS-Fuzz monitors repositories, builds the software as new commits are made, fuzzes with sanitizers (e.g., AddressSanitizer [13]), reports crashes, and periodically checks whether vulnerabilities have been fixed.

OSS-Fuzz-Vulns is a subset of OSS-Fuzz that provides precise impacted version ranges for its included vulnerabilities. The located patches are provided to help software users determine whether their deployed versions are affected by known issues. This dataset, integrated into the Open Source Vulnerabilities project (OSV) [7], is often referred to as OSS-Fuzz-OSV.

OSS-Fuzz-OSV is a semi-automated pipeline focused on providing precise affected-version ranges: OSS-Fuzz-OSV avoids long-term reproduction challenges by reproducing bugs immediately after they are reported in OSS-Fuzz and archiving the located patches. This approach takes advantage of recency: reproducing a crash right after discovery is much easier than reproducing one from years earlier. It also biases OSS-Fuzz-OSV to vulnerabilities with fewer dependencies, and the reproducibility is therefore not maintained over time.

2.2. Patch Locating

For known vulnerabilities, the corresponding source-code fixes, called *patches*, are vital to security research. Source code patches can be used to detect their presence in binaries [14]–[16] and to enable hot-patch generation [17], [18]. Revision control systems make patching possible by recording historical changes. However, automatically identifying the relevant patch remains an unsolved problem, even when a PoC is available. While datasets such as CVE and NVD aim to alert system maintainers about vulnerabilities and track affected software versions, they merely record reported patches and do not verify or identify the actual fixing patch.

The available PoCs could, in principle, be used to test whether the corresponding patch is present in a binary. In practice, however, historical binaries are difficult to rebuild because existing reproducibility solutions are limited, so PoCs can hardly be used to detect the patch. Consequently, current methods typically rely on keyword matching in commit messages and code. Automated methods such as CVEfixes [19], VCCFinder [20], and PatchScout [21] are designed to map each CVE vulnerability to its patch. Although they have shown good performance when evaluated on CVE datasets, their accuracy is limited when it comes to locating patches for non-CVE bugs. CVEs often receive considerable attention, making it easier to gather detailed information about them, which aids in patch identification. However, these methods struggle to generalize to the domain of generic OSS bugs, which represent the majority of cases.

2.3. Related Security Datasets

Existing datasets neglect reproducibility. Most focus on achieving either quantity or diversity, while only a few datasets provide limited reproducibility. To highlight these trade-offs, we review prior work across three types of security datasets: crafted, generated, and collected. To our knowledge, none achieves the combination of reproducibility, quantity, and diversity that ARVO provides. Table 1 summarizes ARVO dataset with several representative security databases. We divide these databases into three categories according to how they are constructed and discuss their advantages and limitations.

Crafted Datasets are manually built and are therefore often limited in size. They are typically limited in both diversity and scale. For example, the CGC dataset [3] (276 vulnerabilities) was for evaluating automated vulnerability discovery methods, while the ExtractFix [22] dataset (30 vulnerabilities) was for evaluating vulnerability patching. Some crafted datasets also include high-quality real-world bugs. Magma [5], for instance, is a fuzzing benchmark dataset that backports high-quality vulnerabilities into a recent version. Nevertheless, their limited scale and manual, non-automated construction restrict their usefulness for binary security research.

Generated Datasets, in contrast, are automatically created at a large scale to synthesize or identify vulnerabilities. However, they often lack diversity and complexity. For example, FormAI [23] produced extensive datasets (112,000 vulnerabilities) but the average program code length is only 79 lines, far simpler than real-world software. Although FormAI covers over 60 CWE types, they come from only nine categories, reflecting a common limitation: generated datasets often lack diversity and focus on similar vulnerabilities.

A second limitation is accuracy. D2A [24] achieves only 53% label accuracy, making it unreliable for research uses. To gain more accuracy, LAVA [2] takes a different approach by inserting bugs into existing complex software and verifying reachability with PoCs. However, the trade-off is poor diversity, as LAVA focuses solely on buffer overflows.

Collected Datasets, unlike generated ones, are derived from various real-world projects. This provides greater diversity and complexity , but comes at the cost of scale, and reproducibility is particularly challenging. As a result, these datasets often embody a trade-off between quality (supported features) and quantity, and their limitations become clearer when compared against one another.

Datasets such as Big-Vul [25], PrimeVul [26], DiverseVul [27], and CVEFixes [19] offer a substantial number of vulnerabilities. Yet, none provide binaries or re-compilation environments, making them unsuitable for Fuzzing or auto-patching generation evaluation. They also face accuracy issues: in a manual verification of labeling accuracy [27], DiverseVul's labeling correctness is 60%, while CVEFixes and Big-Vul have lower accuracy rates of 51.7% and 25%, respectively.

In contrast, OSS-Fuzz-OSV prioritizes high-quality data over size. It offers PoC–binary pairs, which support research using dynamic methods. However, its binaries are **not recompilable**. Its official reproducer struggles to recompile old or complex vulnerabilities, with an overall success rate of 37%. We return to these accuracy concerns in Section 5.2, where OSS-Fuzz is compared directly against ARVO.

The ARVO dataset that we present in this paper is, to our knowledge, the *first* to achieve large-scale reproducibility with real-world software, combining quantity, diversity, and reproducibility. We are not aware of any other public dataset achieving recompile-level reproducibility at a comparable scale and across such a wide range of projects. We

TABLE 1: Prior existing datasets and comparison with ARVO. *Type*: how the dataset was constructed (Crafted = manually built; Generated = automatically synthesized; Collected = aggregated from real projects). *Re-compilable*: each vulnerability can be rebuilt from source on demand. *Automated*: dataset construction does not require per-vulnerability manual effort. *PoC*: each vulnerability includes a triggering input. *Patch*: each vulnerability is linked to its fix commit. *Real-World*: vulnerabilities are derived from real-world projects.

Dataset	Type	# Vulns	# Projects	Re-compilable	Automated	PoC	Patch	Real-World
CGC	Crafted	276	249	✓	✗	✓	✓	✗
ExtractFix	Crafted	30	7	✗	✗	✗	✓	✓
Magma	Crafted	118	7	✓	✗	✓	✓	✓
FormAI	Generated	112,000	1	✗	✓	✗	✗	✗
D2A	Generated	18,653	6	✗	✓	✗	✓	✓
LAVA	Generated	2,265	4	✓	✓	✓	✗	✗
Big-Vul	Collected	3,754	348	✗	✗	✗	✓	✓
PrimeVul	Collected	6,968	755	✗	✗	✗	✗	✓
DiverseVul	Collected	7,514	295	✗	✗	✗	✓	✓
CVEFixes	Collected	5,495	1,754	✗	✓	✗	✓	✓
OSS-Fuzz-OSV	Collected	3,381	331	✗[†]	✗[‡]	✓	✓	✓
ARVO	**Collected**	**6,138**	**311**	✓	✗[§]	✓	✓	✓

[†] The OSS-Fuzz-OSV data is collected via an initial reproduction of each issue, but reproducibility is not guaranteed over time. Reproduction is inherited from OSS-Fuzz rather than being solved by OSS-Fuzz-OSV. Although OSS-Fuzz-OSV is a reproducible subset of vulnerabilities of OSS-Fuzz, which implies fewer reproducibility issues, it fails to reproduce complex (multi-component) or older vulnerabilities over time.
[‡] OSS-Fuzz-OSV is mostly automated; a minority of reports are supplemented by manual input from project maintainers.
[§] ARVO's per-vulnerability pipeline is fully automated; only a few hours of system-wide maintenance per season are needed to redirect defunct external resources flagged by the pipeline.

further discuss the challenges and benefits of reproducibility in Section 3, and ARVO's solutions in Section 4.

3. Reproducibility

As shown in Section 2.3, prior datasets emphasized scale or diversity but neglected reproducibility, leaving researchers to rely on only a small fraction of these datasets after substantial manual effort to reproduce individual cases.

Achieving full reproducibility (rebuild plus retrigger) introduces several challenges, which we highlight in the rest of this section.

3.1. Reproduction for Vulnerabilities

Reproducibility is the property that allows vulnerabilities to be reliably reconstructed and studied, rather than remaining as text-based reports. It depends on two complementary elements:

Reproducing Resources are the metadata that can help reproduce a vulnerability. These include vulnerability descriptions, source code of the related components, the reproduction environment, compilation methods or scripts, an example of a vulnerable binary, the Proof of Concept (PoC) inputs that trigger the vulnerability, and the corresponding patches. Not all of these resources are always necessary, but each contributes to making reproduction more reliable.

Reproducing Pipeline is the procedure that uses available resources to consistently replay the expected behavior. An effective pipeline should tolerate missing or partial resources while still maintaining reliable success.

Prior work such as OSS-Fuzz-OSV connects **Reproducing Resources** with the **Reproducing Pipeline** by reproducing a bug shortly after it is reported and archiving the result, rather than maintaining the ability to rebuild and trigger it on demand. As build environments and dependencies drift, reproducibility degrades over time. What has been missing is a methodology that sustains reproduction at scale, rebuilding the vulnerable binary from source and triggering the intended crash through the given PoC.

3.2. Benefits

Reproducibility enriches the information that can be extracted from historical vulnerabilities. It improves data quality and enables downstream researchers to reliably build and evaluate new security techniques.

Improving data quality. Reproducibility makes it easier to filter false positives and make the data verifiable, addressing a recurring problem in large-scale collections. We quantify this in Section 6.3.

Enabling richer information extraction. Reproducibility enables extracting metadata that is otherwise hard to recover, such as the vulnerability's fixing patch via commit bisection. We evaluate this in Section 5.2.

Supporting downstream research. Current datasets still fall short of the needs of downstream security research, such as program repair [10], [28], [29]. A reproducible ARVO dataset covering a large number of real-world vulnerabilities can be used to evaluate patch quality, and its vulnerability-patch pairs can be integrated as training data for machine-learning-based methods. Reproducibility also enables systematic vulnerability backporting for fuzz testing. Because ARVO dataset can rebuild historical versions and revert fixing commits, it can automatically backport known vulnerabilities into older software versions and verify each vulnerability's reachability by its PoC. As we show in Section 6.2, this automation lets ARVO dataset scale backporting across many projects, complementing manually curated efforts such as Magma.

3.3. Challenges

Constructing a large-scale, reproducible vulnerability dataset is far from trivial. The primary difficulty lies in reviving historical software versions, where compilation and environment setup frequently fail in subtle ways. We summarize the key challenges below and describe ARVO's solutions in Section 4, as well as a comprehensive ablation study in Section 6.1.

Incompatible Dependencies. Although historical source code is usually preserved in revision control systems, recompilation itself is a complex process. It typically requires many dependencies, including not only specific libraries, but also certain versions of compilers and system tools provided by the operating system. Open-source software often depends on other components as well. For example, compiling `ffmpeg` involves more than a dozen independent open-source projects. If a reproducer ignores version control for these dependencies, compilation will likely fail due to incompatible interfaces. Therefore, any reproducible vulnerability dataset must capture and restore the exact versions of its dependencies.

Missing Resources. In real-world scenarios, software compilation often requires fetching resources from the Internet. These may include dependency source code downloaded via `git clone` or special tools retrieved with `wget` or `curl`. While this approach may work initially, over time the original URLs often become invalid as projects migrate to new hosting platforms or reorganize their repositories. For example, the PCRE library migrated from an FTP-hosted SVN repository to a GitHub-hosted git repository, breaking historical build scripts that relied on the old address. Since these resources are essential for successful reproduction, any reproducible dataset must capture or redirect them.

Fragile and Mercurial Build Processes. The build process is the most critical step in reproducing a vulnerability. To keep the diversity and complexity from the upstream, we must handle various build processes. Ideally, resource fetching and compilation should be separated into distinct stages. In practice, however, historical build scripts rarely follow this principle; build scripts commonly download resources during compilation, which makes it harder to fix "Incompatible Dependencies" and "Missing Resources." Given the diversity of targets and the complexity of real-world projects, this is particularly challenging, as even minor modifications to the build process can easily cause failures.

These challenges are not isolated; they often interact and amplify one another, making reproduction far more complex than it initially seems. This interdependence explains why prior datasets have largely ignored reproducibility, despite its importance for research utility. Addressing reproducibility in vulnerability datasets is therefore both novel and non-trivial. In the next section, we introduce ARVO, which systematically tackles these challenges through a scalable, automated design.

4. ARVO

We designed ARVO with the goal of producing a reproducible and scalable vulnerability dataset that addresses the challenges mentioned in Section 3.3. In detail, we aim to achieve:

Reproducibility. ARVO should provide all the reproducing resources (as Section 3.3 discusses) along with a reliable pipeline to recompile both vulnerable and fixed targets from source code.

Scalability. The dataset should contain a large number of vulnerabilities and automatically incorporate new vulnerabilities as they are found, to allow the dataset to expand and grow easily over time.

Quality and Diversity. Each vulnerability in the dataset should be validated to ensure it is actually a bug with security impact. The vulnerabilities should be distributed across a large number of different projects to ensure that evaluations using the dataset are representative.

Ease of Use. The dataset should be easy for researchers and practitioners to use, without requiring extensive security expertise or knowledge of how to build the projects.

In this section, we will describe the methods used in ARVO and how they mitigate the reproducibility challenges.

4.1. Overview

ARVO consists of two major components: (1) the reproducer and (2) the vulnerability patch locator. ARVO outputs the ARVO dataset.

ARVO is a framework to generate a reproducible and interactive research vulnerability dataset, designed to ingest source metadata from "bug"/project databases and augment this information with relevant source code, build steps, and binaries. Because we hope to support downstream uses such as analysis of security patches, evaluating vulnerability discovery systems, and automated vulnerability repair, the ARVO dataset also needs to include environments for re-compiling the code of each project so that modifications to the source code can be straightforwardly tested. To enable easy access, ARVO provides an online Dockerized dataset as well as infrastructure to build the dataset from scratch.

The reproducer takes the provided metadata from the upstream bug database(s), compiles the project at the specified (vulnerable) version, and verifies that the provided triggering input causes a crash. It also compiles the project at the fixed version listed in the upstream dataset and checks that the crash no longer occurs. If either of these steps fails, the vulnerability is marked unreproducible and excluded from the dataset.

However, as previously discussed, the upstream metadata often does not specify the exact fixing commit, but only a range of candidates. ARVO's vulnerability patch locator searches this commit range to find the earliest commit that resolves the issue. ARVO's reproducibility enables the locator to bisect the commit history and identify the exact changes that fix the vulnerability.

4.2. Upstream Dataset

To obtain a large number of vulnerabilities and allow the dataset to grow over time, ARVO is designed to draw

project and bug metadata from upstream sources (currently, OSS-Fuzz). We rely on some assumptions about the upstream data source (discussed in Section 3):

1) Version/Time Information: To reproduce and pinpoint its fix, we need version identifiers (e.g., git commit hashes) from the revision control system, which identifies the vulnerable and non-vulnerable versions of the project and its dependencies. When such identifiers are unavailable, ARVO falls back to using timestamps to approximate the correct versions, though this introduces some loss of precision.
2) Build Environment: A virtualized, interactive environment that can compile and execute the target programs and their dependencies.
3) Crash Information: At minimum, we need a triggering input and the command to execute the target program on that input. Additional information, such as sanitizer output, can also be used to validate that the crash we observe is the same one identified by the upstream source, though it is not strictly required.

We chose OSS-Fuzz as our initial upstream source because of its diversity and complexity. By introducing reproducibility into such a heterogeneous dataset, we can more effectively demonstrate the robustness and generality of ARVO's reproducibility solutions.

To identify security-relevant issues with the required metadata, we searched the issue tracker according to the labels OSS-Fuzz automatically applies to each issue: `Type=Bug-Security` (the crash is likely to be security-relevant, based on the sanitizer report and call stack), `label:Reproducible` (the crash occurs deterministically whenever the triggering input is provided), and `status:Verified` (OSS-Fuzz verified that the target no longer crashes).[2] Combining these query elements, we obtain 8,921 issues in over 300 projects after filtering the false positives (see Section 6.3), which serve as the starting point for our dataset.

4.3. Reproducer

The reproducer is the keystone of the patch locator: to reproduce an issue and identify its precise fix, ARVO rebuilds the project and its dependencies from source across different commits. This task is especially challenging for older vulnerabilities, where dependencies, resources, and toolchains may no longer be available.

By applying the techniques described in this section, the reproducer has successfully reproduced 69% of vulnerabilities (6,138 cases) with their corresponding patches identified. The difference between this number and the 81% success rate observed in our comparison experiment (see Section 6.1) is primarily due to time constraints. Some OSS-Fuzz projects (e.g., LibreOffice) require several hours to compile, and the repeated builds necessary for commit bisection make the process extremely time-consuming. As a result, not all reproductions have been completed; we expect the remaining gap to be closed in the coming months.

2. We will see in Section 6.3 that this label is not always accurate; we found over 300 cases where the provided test case still crashes the most recent version of the project.

We identify three key strategies that ARVO uses to improve the reproducibility of vulnerabilities: 1) minimally intrusive build instrumentation; 2) revision control; and 3) fixing missing resources. These strategies are implemented in the ARVO reproducer, as shown in Figure 1.

Build Instrumentation. Revision control during reproduction requires instrumentation of the build process. OSS-Fuzz implements an intrusive revision control mechanism that breaks the original compilation process. By contrast, ARVO applies minimally intrusive revision control, preserving the natural build flow and making reproduction more reliable.

OSS-Fuzz provides virtualized build environments by compiling projects inside Docker. As shown in Figure 2, the workflow has two stages: Docker Build, which prepares resources, and Docker Run, which performs compilation.

In OSS-Fuzz, revision control is enforced only after Docker Build completes. The project source is swapped outside the container and mounted back in before compilation. This approach ignores any resource modifications or initialization steps that occur during the build stage. Replaying these actions later is imprecise and risky, leading to broken builds.

By contrast, ARVO sidesteps these issues by instrumenting the build process with minimal change. ARVO separates actions into resource-download actions and other actions. The revision control is only needed for resource fetching actions, and ARVO only hooks these actions and keeps the modification *minimally intrusive*. The component initialization actions during the Docker Build stage will not be changed.

In practice, ARVO hooks only resource-download commands instead of mounting external sources. This minimally intrusive philosophy is also integral to our approach to revision control, ensuring that modifications are both effective and non-disruptive.

Revision Control. Correct dependency versions are essential for reproducing historical vulnerabilities, yet this requirement has long been overlooked. Existing efforts typically control only the main component's revision while ignoring its dependencies. However, successful reproduction depends on precise revision control of the entire build environment, including both the main project and all of its dependencies.

The build scripts used to compile the fuzz targets for each project are provided by the project developers in two parts: a Dockerfile that downloads dependencies and external resources, and a `build.sh` script that actually compiles the fuzz targets. The official reproducer provided by OSS-Fuzz rolls back the main project to the vulnerable commit, but it does not attempt to reset dependencies to their corresponding versions. This leads to compatibility issues and build failures since dependencies have changed their APIs or build procedures. For instance, `ImageMagick` relies on 15 separate components, each with frequently changing APIs and usage patterns, and attempting to reproduce a vulnerability in `ImageMagick` without adjusting the dependencies to match the vulnerable version will likely result in a failed build. Also, because the build script is usually attached to the main component and separate from the dependencies, the old

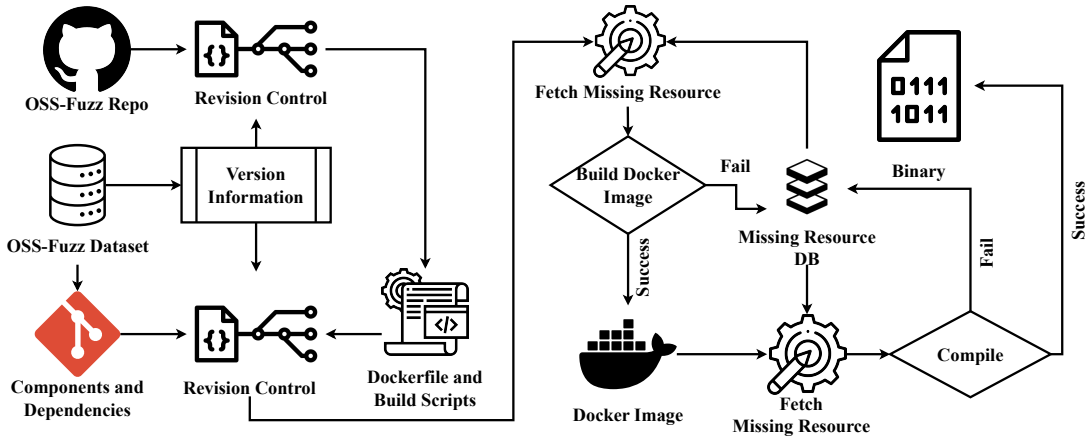

Figure 1: ARVO Reproducer Structure. After reproduction, ARVO performs a verification step using the corresponding PoC: the vulnerable version must reproduce the intended crash, and the fixed version must run without crashing on the same input. Only verified cases are then packaged together with their build environment as a Docker image to support reproduction.

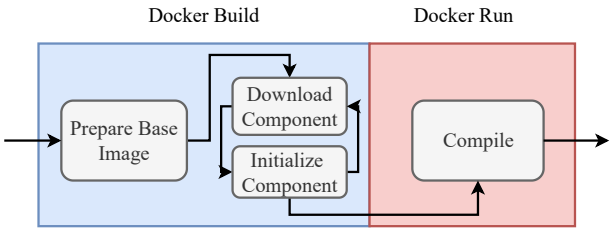

Figure 2: Simplified Compilation Procedure.

compiling commands in the build script usually fail to compile the latest version of dependencies.

To demonstrate the impact of incorrect dependency versions, our ablation study in Section 6.1 evaluates the effect of disabling component revision control.

Broken Resource Fixing. Similar to "bit rot" in software, reproducing old vulnerabilities often encounters missing or inaccessible dependencies, especially for projects from the 2017–2019 period. During this time, many projects migrated their repositories from Subversion to Git, breaking build scripts that reference the old repositories. ARVO mitigates this issue by maintaining a detection-fixing loop that continuously identifies and fixes broken resources.

ARVO divides missing resources into two categories: *core resources*, which are necessary to compile the fuzz target, and *non-core resources*, which are not necessary for compilation but are required for other parts of the build process. Core resources include software dependencies, such as libraries, as well as tools used by the build process that may be necessary to compile key components. Non-core resources include documentation generation tools, seed/corpora used for fuzzing, and other resources that are not directly related to the fuzz target.

To detect missing and broken resources, ARVO captures and logs the error messages generated during the build process and looks for errors related to failed URL downloads. It then classifies each missing resource as core or non-core based on whether it is required to compile the fuzz target; because ARVO performs reproduction rather than fuzzing, resources used only by the fuzzing workflow can be safely skipped. For non-core resources, such as corpora and seeds, ARVO modifies the related commands to prevent build crashes, with specific modifications depending on the command type to ensure minimal loss. For core resources, limited manual work (a few hours per season) is required to locate the missing resource and replace it with a working URL. [3] These resource fixups are stored as reusable rules that can be applied across multiple vulnerabilities and projects.

Although the manual effort might appear daunting, in practice, most missing resources are shared across many vulnerabilities and projects. Fixing a relatively small set of resources, therefore, resolves a large number of cases. For example, our dataset includes only 53 unique missing resources identified over the past eight years; keeping ARVO's fixes updated would thus require updating roughly seven records per year. This translates to just a few hours of work annually, yet it enables substantial gains in reproducibility, allowing us to successfully reproduce thousands of additional vulnerabilities

4.4. Patch Locator

Automatically locating vulnerability patches, as discussed in Section 2.2, is nontrivial even when a PoC is available because of the absence of a reliable reproducer. With its reproducer, ARVO can automatically locate the precise patches that resolve each vulnerability. This section introduces ARVO's patch locator, which applies commit bisection to identify fixing commits. We first describe how bisection is used to locate patches in the main component, and then extend the discussion to the more challenging case where dependencies must also be tracked during bisection.

Benefiting from the combination of continuous fuzzing and data collection, OSS-Fuzz stands out because it automatically provides, for each discovered bug, a range

3. An experimental version of ARVO uses LLMs to reduce manual effort, primarily by resolving compilation issues and filtering upstream false positives. The results reported in this paper do not depend on any LLM involvement.

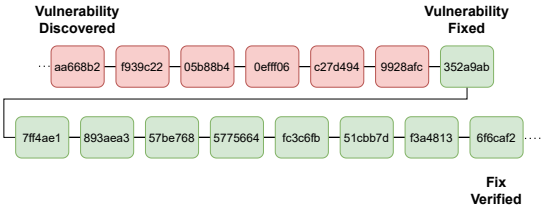

Figure 3: Vulnerability lifecycle for OSS-Fuzz issue #42508698 on `ImageMagick`.

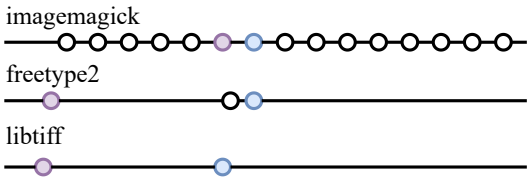

Figure 4: Revision control for the locator. To compile the blue/purple revision of `ImageMagick`, corresponding dependency revisions are selected using commit timestamps. Only 2 of 14 dependencies are shown for clarity.

of commits that includes the patch. The reported result is a range rather than a single commit because OSS-Fuzz checks for vulnerability fixes periodically rather than commit by commit. For most projects, this is typically a daily build. During their daily builds, OSS-Fuzz checks whether the latest version still crashes on the recorded PoC. If it does not, the issue is marked as fixed. For highly active projects, however, the delay between the maintainers' fix and OSS-Fuzz's verification can create a range of candidate commits for the actual patch. There are also cases that include larger ranges, where OSS-Fuzz did not verify whether the patch PoC was applied for a longer time.

Figure 3 illustrates the process through a concrete example: the vulnerability is first identified at `6f6caf`; the maintainers commit a fix the next day at `aa668b`; and OSS-Fuzz verifies the fix at `aa668b`. However, `aa668b` is not the actual fix but merely a minor update to the `ChangeLog` file. To identify the actual fix, we must search over the 14 commits and 83 files that were changed between the initial report and the verification.

ARVO's reproducer makes it possible to use commit bisect to locate the vulnerability patch. However, once dependencies are taken into account, the task becomes more complex than simply bisecting the main component's commits.

During bisection, every chosen commit of the main component must be compiled and tested with the triggering input. To keep these builds working, ARVO also needs to identify the correct dependency versions for each commit. If the dependencies are mismatched, the project may fail to build at a chosen midpoint. ARVO handles such cases by stepping to a neighboring commit so that bisection can proceed. In the extreme case where a high fraction of intermediate commits fail to build, bisection can no longer make progress and ARVO falls back to a *linear search*: it walks candidate commits one by one, which is significantly more expensive and may yield a range of commits rather than a single fix commit. Unlike in reproduction, the fix locator cannot rely on dependency versions recorded upstream, since those versions may no longer apply across the entire commit range.

To address this, ARVO uses commit timestamps. For each main component commit, it selects the most recent compatible dependency commits available at that time, ensuring that the project remains buildable throughout the bisection process.

Figure 4 illustrates this approach with the `ImageMagick` case. For each revision of the main component, we use the commit timestamp to select the appropriate dependency versions, which greatly improved build reliability. With this precise version control in place, ARVO can bisect across thousands of commits to accurately pinpoint the fixing commit.

4.5. Beyond OSS-Fuzz

Although this paper focuses on OSS-Fuzz, the methodology described above is not specific to it: any upstream that records vulnerable source revisions, build instructions, and triggering inputs can be plugged in. As a proof of generality, we ported the pipeline to a different upstream, syzbot [30], which reports kernel bugs found by syzkaller. Each syzbot report supplies a syz-language reproducer (and often a C reproducer) along with the affected kernel commit, which maps directly to ARVO's required inputs.

We sampled 144 syzbot reports at random; 44 referenced commits in orphan linux-next or subsystem trees that were no longer reachable from mainline and thus fell outside the methodology's scope. On the remaining 100 reachable reports, the prototype reproduced 78 (78%); 20 built and ran but did not crash within the timeout (typically race conditions), and 2 failed to build. The reproduced crashes span `KASAN` reports, `BUG`, `WARNING`, and general protection faults.

The Linux kernel is in some respects a friendlier target than OSS-Fuzz: it is a single, well-maintained codebase with a uniform build system, so most of ARVO's original challenges (heterogeneous build scripts, missing third-party resources) do not apply. On the other hand, kernel bugs include a substantial proportion of race conditions that resist PoC-based reproduction, which is the dominant cause of the 20 "built but did not crash" cases above. The OSS-Fuzz evaluation in Section 6 therefore remains the more demanding and informative one; the kernel result only confirms that ARVO's solutions transfer to a non-OSS-Fuzz upstream.

4.6. Dataset Access

A goal of our dataset is that it should be easy to use, even for researchers who do not have a security background; we hope that this will allow researchers in other fields (e.g., machine learning) to use it as an evaluation target. Based on ARVO, we have uploaded Docker images for each vulnerability to Docker Hub, enabling each issue to be reproduced and recompiled with a single command:

`docker run <repo name>:<localId>-<vul|fix> arvo [compile].`

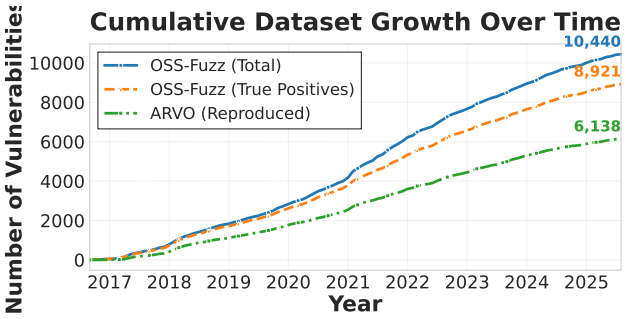

Figure 5: Database growth over time.

To support more advanced uses of the dataset (e.g., rebuilding the project with other instrumentation), we open-source ARVO so researchers can rebuild the ARVO dataset from scratch with their desired changes.

5. Dataset

This section presents the details of the ARVO dataset constructed using the methods described in Section 4.

5.1. Dataset Characteristics

Dataset Size and Growth. At the time of writing, ARVO has successfully reproduced 6,138 vulnerabilities across 311 projects, out of the 8,921 vulnerabilities initially obtained from OSS-Fuzz. For each reproduced case, we provide interactive environments supporting recompilation and instrumentation for deeper analysis. In addition, 221 of these vulnerabilities are also linked to their corresponding CVE identifiers.

We measured our reproduction success over time and found that ARVO has maintained a roughly constant success rate (Figure 5). Our design emphasizes a general approach that works across diverse open-source projects. As Figure 5 shows, the size of the ARVO dataset grows steadily in tandem with OSS-Fuzz, which continues to expand its project coverage and discover new bugs. By leveraging its upstream's ongoing efforts, ARVO is well positioned to scale into the future, ensuring that the ARVO dataset continues to grow and stay up to date.

Project and Language Distribution. To demonstrate the diversity of ARVO dataset, we computed the distribution of vulnerabilities among 311 projects. This distribution is relatively even; the top 10 projects collectively account for only 35.71% of all vulnerabilities in the dataset. This indicates the comprehensive diversity of the ARVO dataset: Rather than concentrating around a small set of projects, it spans a wide range of software.

Duplicates. In ARVO dataset, "duplicates" are kept. During upstream fuzzing, a vulnerability could crash on different harnesses/functions by different fuzzing engines (e.g., libfuzzer and AFL). These "duplicates" are crashes triggered by different PoCs but fixed on the same commit. Based on ARVO located patches, we can connect these crashes to the same cause. ARVO keeps them in the dataset but marks them as different crashes fixed by the same patch. Preserving and labelling duplicates in this way is preferred for downstream applications, such as auto-patch generation, since more PoCs could be used to verify the correctness of generated patches.

Patch Statistics. Of the 6,138 vulnerabilities in the ARVO dataset, we first filtered out duplicates (e.g., when a single patch fixed multiple vulnerabilities—1,550 cases).

The large size of this dataset allows us to collect some interesting statistics on the nature of vulnerability patches. Prior research has found that security-related fixes are typically small and self-contained [31]. Our data also supports this finding: In ARVO dataset, 90% of the patches modified 4 files or fewer; 2,895 patches (63.22%) affect just a single file. Looking at the number of lines added and removed by each patch, we found a median of 6 lines added and 2 lines removed; the mean of both is significantly larger (228.7 added and 115.1 removed) due to a small number of outliers. 90% of the patches in our dataset have fewer than 62 lines added and fewer than 32 lines removed.

We also found that 53.68% (2,458) of the patches directly modify the source file and function present in the sanitizer-reported call stack. Of these, 37.46% of the patches have modified the function on the 0^{th} index in the crashing call-stack, 32.88% of the patches modified functions at the 1^{st} index, and 25.49% of the patches modified functions on the 2^{nd} index.

5.2. Data Accuracy Evaluation

To evaluate the accuracy of ARVO dataset, we compare it directly against OSS-Fuzz-OSV. By manually analyzing random samples from both ARVO and OSS-Fuzz-OSV, we obtain an overview of patch correctness and the relative reliability of each dataset.

Overview. ARVO dataset has 6,138 successfully identified patches while OSS-Fuzz-OSV has documented 3,381 vulnerabilities with patches. To make the comparison, we focus on the overlapping cases between them, a total of 2,219 vulnerabilities. The overlapping set of vulnerabilities has 0.83 dependencies on average, while the average ARVO dataset vulnerability has 4.05 dependencies. Based on the located patch commits, we divided them into three groups: agree (66.61%), disagree (20.91%), and partially agree (12.48%).

We manually evaluated 100 random cases from each group. In agree and partially agree cases, 97% and 93% were confirmed as true positives, respectively. Most partially agreed cases involve a merging commit and an effective commit in another branch. We considered a case a true positive when the overlapping commit patched the vulnerability. There were a total of 7 false positives in these two groups, and all of them were due to changes in the harness or compilation settings rather than actual vulnerability fixes.

In the disagree group, where ARVO and OSS-Fuzz-OSV reported different patches, the accuracy drops. Specifically, 63% of ARVO's results were true positives (versus 32% for OSS-Fuzz-OSV).

Based on the group sizes and per-group accuracies, the overall accuracy is computed as a weighted average:

$$\text{Overall accuracy} = \frac{\sum_{g \in \{\text{agree, partial, disagree}\}} N_g \, p_g}{\sum_{g \in \{\text{agree, partial, disagree}\}} N_g},$$

where N_g is the size of group g and p_g is its accuracy. Applying this formula yields 89.4% for ARVO dataset, outperforming OSS-Fuzz-OSV's 82.9% on patch data accuracy.

Both ARVO and OSS-Fuzz-OSV show high correctness in the agree and partially agree groups. However, in the disagree group, ARVO achieves nearly double the accuracy rate compared to OSS-Fuzz-OSV. In the 100 randomly sampled cases, ARVO exclusively provided valid patches for 46 vulnerabilities, whereas OSS-Fuzz-OSV identified only 15 such cases. In addition, there were 17 cases where both systems located valid patches, and 22 cases where neither system provided a valid patch.

Analysis. Although OSS-Fuzz-OSV benefits from maintainer input when identifying fix commits, ARVO achieves higher accuracy. The key advantage lies in reproducibility.

Unlike ARVO, the OSS-Fuzz-OSV dataset includes not only patches confirmed by binary search but also maintainer-reported fixes, which can introduce false positives. For example, in issues #42529818 and #42486491, OSS-Fuzz-OSV linked the vulnerabilities to plausible but ultimately unrelated patches. After extensive investigation, we found that these patches did not address the target vulnerabilities, whereas ARVO correctly identified the functional fixes. Reproducibility also helps improve patch quality. In issue #42541392, ARVO's recursive binary search pinpointed the actual fix in a submodule, whereas OSS-Fuzz-OSV listed only a submodule update commit.

However, this bisection-based patch locating method also has limitations. A PoC failing to trigger a crash does not necessarily mean that the vulnerability is fixed. Any change that makes the bug unreachable, such as modifications in the fuzzing harness or compilation settings, may suppress the crash without addressing the root cause. Automated upstream datasets like OSS-Fuzz may accept such changes as valid fixes, which can also mislead ARVO. Our manual evaluation shows that ARVO's results are highly reliable (89.4%), with the remaining ~ 10% false positives, primarily due to patches that suppressed the crash without actually fixing the underlying vulnerability. Overcoming this challenge will require combining dynamic methods (bisection) with complementary techniques such as semantic understanding. By ensuring reproducibility, ARVO provides the foundation for such future advances.

6. Case Studies

This section presents three case studies that illustrate both the effectiveness of ARVO's reproduction solution and its broader value for security research. Together, they demonstrate how reproducibility benefits ARVO itself, its downstream applications, and even its upstream datasets.

The first case study examines ARVO directly, using an ablation study to measure the contribution of its core features to reproducibility. The second demonstrates a downstream use: leveraging ARVO to reintroduce known bugs into software and construct reliable fuzzing benchmarks. The third shifts upstream, analyzing cases where ARVO failed to reproduce vulnerabilities and showing that many of these failures stem from OSS-Fuzz false positives rather than limitations of ARVO. Collectively, these studies highlight not only ARVO's strengths but also the broader impact of reproducible datasets across software security research.

TABLE 2: Successful Reproduction Counts for 100 Random Issues.

# Components	# Total	# OSS-Fuzz	# ARVO
1	47	27	41
2–4	29	10	19
5–10	10	0	9
>10	14	0	12
Total	**100**	**37**	**81**

6.1. Reproducibility Comparison and Ablation Study

To demonstrate ARVO's improvement in reproducibility and to highlight the significance of its core contributions, we compare ARVO with OSS-Fuzz's official reproducer and conduct an ablation study on ARVO's key solutions.

Reproducibility Comparison. Due to recent upstream changes (OSS-Fuzz migrated its issue tracker), ARVO had to rebuild its entire dataset. Reproduction and patch localization are time-consuming because they require multiple rounds of compilation. Consequently, the 6,138 completed cases do not reflect ARVO's full capacity, since it has not finished reproducing all upstream issues. To provide a fair and controlled comparison, we randomly sampled 100 vulnerabilities from OSS-Fuzz. Each issue was reproduced using both the official OSS-Fuzz reproducer and the ARVO reproducer.

As shown in Table 2, ARVO achieves a success rate of 81%, substantially outperforming OSS-Fuzz's 37%. OSS-Fuzz succeeds mainly in projects with a single component and struggles in multi-component projects due to its lack of dependency version control. This indicates that the OSS-Fuzz-OSV dataset is biased toward simpler projects, overlooking more complex ones. In contrast, ARVO consistently improves reproducibility across both simple and complex projects, covering 67 distinct projects in total. The 81% success rate demonstrates that ARVO's solutions effectively mitigate the key challenges in vulnerability reproduction.

Ablation Study. To evaluate the contribution of ARVO's key features to reproducibility, we conducted an ablation study by selectively disabling (1) Revision Control for non-main components, (2) Resource Fixing, and (3) Base Environment Control (providing the corresponding base image environment for compilation). The results are shown in Table 3.

In the ablation study, disabling each feature reduced ARVO's success rate to varying degrees. Disabling only Resource Fixing has a smaller effect, while both Full Revision Control and Matching Base Environment caused larger drops. When features were disabled in combination, reproduction success decreased sharply, showing that all three components contribute in complementary ways.

Overall, ARVO improves reproducibility substantially compared to OSS-Fuzz (81% vs. 37%). The ablation results confirm that ARVO's features are not only individually useful but also mutually reinforcing. Together,

TABLE 3: Reproduction Counts with ARVO Features Disabled.

Disabled Feature	# Reproduced	# Lost	Success Rate
RF	54	27	66.7%
BE	50	31	61.7%
RC	46	35	56.8%
RF and BE	46	35	56.8%
RC and RF	42	39	51.9%
RC and BE	37	44	45.7%
RC, RF, and BE	34	47	42.0%

RF = Resource Fixing, BE = Base Environment, RC = Revision Control.

they enable ARVO to scale to more complex projects and provide a strong foundation for downstream vulnerability research.

6.2. Vulnerability Backporting

ARVO and Magma share a goal of building fuzzing benchmarks from real vulnerabilities, but pursue it through different strategies. This case study demonstrates how ARVO enables the creation of such benchmarks at scale, complementary to (rather than directly comparable with) Magma's approach. Recent benchmark efforts such as Magma highlighted the importance of realistic bugs for evaluating fuzzing performance. Magma forward-ports past vulnerabilities into a recent version of the software, which is valuable for evaluating mitigations and fuzzers against modern codebases, but requires manual patch adaptation; Magma originally introduced 118 bugs into 7 projects, later expanding to 9.

ARVO automates Magma's approach. Rather than relying on manual patch adaptation, ARVO searches among the commits related to each vulnerability (e.g., the commits where the bug was first detected and later fixed) for a target commit. By default, ARVO selects the commit on which most bugs can be triggered, maximizing benchmark size. If the user instead requires a specific recent commit (matching Magma's evaluation scenario), ARVO supports that target commit as well, at the cost of fewer triggerable bugs.

Once a target commit is chosen, ARVO compiles the program with varying combinations of inserted vulnerabilities and verifies each inserted bug by its PoC, keeping only the bugs whose PoC still triggers the intended crash. The resulting benchmark therefore contains only reachable bugs, together with their PoCs and corresponding patches.

To evaluate this idea, we applied it to 34 projects, excluding complex projects whose compilation takes hours and the ones with fewer than 40 historical vulnerabilities. For each project, we search for the code version suitable for porting most known vulnerabilities. As shown in Table 4, the project inserted most vulnerabilities in our evaluation contains 45 verified vulnerabilities, each confirmed by its corresponding PoC. The crash reports from these PoCs cover 9 distinct crash types, including `global-buffer-overflow`, `heap-buffer-overflow`, and `heap-use-after-free`. The automatically backported vulnerabilities are also not concentrated in a small set of files but are spread across the codebase: in the top 5 projects alone, 145 vulnerabilities land in 100 different files. This is achieved without any manual patch adaptation across versions (avoiding patch-application issues), at a scale beyond what manual approaches such as Magma can practically reach across many projects.

TABLE 4: Vulnerability Backporting Result (Top 5)

Project Name	# Verified Vulns[a]	# Affected Files
ghostscript	45	27
assimp	43	26
mupdf	30	26
selinux	14	11
opensc	13	10
Total	**145**	**100**

[a] The triggered vulnerabilities after deduplication

While this case study demonstrates the potential of automated vulnerability insertion, it also highlights interesting challenges that invite further exploration. Reverting a fix to reintroduce a bug onto a different commit faces the same core challenge as patch backporting: when the codebase has changed substantially between the bug's original location and the target commit, the patch (or its reverse) no longer applies cleanly. Even when a patch applies successfully, peripheral changes to surrounding code can leave the patched code referencing dependencies (functions, types, or fields) that no longer exist, causing compilation to fail. In our experiments on the `ndpi` project, most attempts failed for this reason. The similar challenge is discussed in patch-porting research [32], [33]. We did not integrate these techniques in our evaluation, and applying them to ARVO is left for future work. Another challenge lies in triggerability: not all reintroduced vulnerabilities can be triggered by their associated PoCs. Shallow and easy-to-trigger bugs (e.g., `Use-of-uninitialized-value`) can prevent PoCs from reaching deeper vulnerabilities, masking the intended crash. As a concrete example, we ran an exhaustive backporting attempt on the `ghostpdl` project, which contained 177 historical vulnerabilities at the time of the experiment. ARVO successfully backported 45 of them; among the 132 failures, 63 were deep vulnerabilities blocked by shallower ones reaching the crash first, 27 were patches that could not be cleanly reverse-applied, and 42 were cases in which the original PoC no longer triggered the vulnerability.

Despite these limitations, the case study shows that ARVO can insert historical vulnerabilities into diverse projects more efficiently than previous approaches, demonstrating both its scalability and effectiveness.

6.3. Correct False Positives on the Upstream

Accurate data is the foundation of reliable research. Conclusions drawn from noisy or incorrect vulnerability reports risk being misleading. While analyzing cases that ARVO failed to reproduce, we found that nearly half were not due to ARVO's limitations, but rather to incorrect upstream data. In these instances, ARVO could build the software, yet the provided PoCs failed to demonstrate the expected behavior: either the pre-patch commit did not crash, or the post-patch commit still did. This revealed two issues: numerous OSS-Fuzz reports correspond to false positives, and many patches are incorrectly labelled as fixes.

We identified **1,519 false positives** by combining ARVO's reproduction results with a confirmation step against the original binaries archived by OSS-Fuzz: when even running the upstream-archived binary on its recorded PoC failed to reproduce the expected crash, the vulnerability was treated as a false positive rather than as an ARVO reproduction failure. These cases fall into three categories: (1) crashes that cannot be triggered consistently (e.g., race conditions or OSS-Fuzz internal bugs), polluting the dataset; (2) crashes fixed outside the commit range identified by OSS-Fuzz, leading to mislabeled patches; and (3) crashes still reproducible on the latest upstream commit, posing a serious risk since attackers could exploit these unfixed vulnerabilities.

Handling category (1) is not complex: such crashes can be filtered out to avoid unreliable data. For category (2), ARVO leverages its unique reproducing ability to locate the true fix by bisecting the commits between the latest version and the commit where the crash was first observed. As an illustrative example, issue #42486945 [34] on OSS-Fuzz was claimed fixed. However, OSS-Fuzz pointed to a commit [35] that merely modified a README file as the "fix" of the heap buffer overflow, which is clearly incorrect. With ARVO, we located the actual patch [36] (Appendix A), which was applied nearly two years later. The commit message indicates that it fixes a wrong size passed to `memcpy`, and according to the crash call stack reproduced by ARVO, the modified function is indeed responsible for allocating the overflowed heap chunk. Furthermore, ARVO's deduplication revealed that issue #42502614 shares the same fix. Although different fuzz engines (`honggfuzz` and `afl`) were used, the crash reports were nearly identical. Thus, issue #42486945 remained an unfixed vulnerability for around **two years**, with OSS-Fuzz incorrectly labelling it as resolved while pointing attackers directly to the PoC.

Finally, for category (3), we confirmed and reported more than **300 cases** to the upstream, where the very latest commit remained vulnerable. By feeding these corrections back, ARVO strengthens the reliability of upstream datasets and avoids leaking information about unfixed vulnerabilities.

7. Discussion

ARVO presents a novel type of vulnerability dataset by introducing *reproducibility* into scalable upstreams, turning static records into interactive artifacts that can be rebuilt from source. The resulting ARVO dataset offers comprehensive vulnerability data to support security research. However, it primarily serves as a mitigation to recover lost reproducibility.

Upstream infrastructures typically archive the crashing binary but omit the *entire build environment* and other reproduction details. This missing context makes it increasingly difficult to reproduce vulnerabilities over time. A more sustainable long-term approach is for upstream sources to capture and preserve reproducibility information directly. If the build environment and related context were archived, datasets like ARVO would not need to reconstruct them later, and the broader open-source security community would benefit from easier, more reliable reuse of vulnerability data. Even if ARVO dataset itself becomes unnecessary once upstreams adopt reproducibility, the central contribution of this paper remains: introducing reproducibility into security datasets.

7.1. Limitations

Even though ARVO demonstrates significant improvement over prior datasets, the methodology does face certain limitations.

Source-Dependent. ARVO relies on upstream datasets and their metadata. While ARVO can detect some issues, broken metadata from the upstream dataset may lead to false positives.

Vulnerability Scope. ARVO is biased toward vulnerabilities that manifest as a crash on a PoC input. Bugs that are hard to reproduce, such as race conditions whose triggering window is narrow, are excluded; so are vulnerability classes without an observable crash signal (e.g., logic flaws). Extending the methodology to these classes is left for future work.

PoC-based Reproduction. While reproducing vulnerabilities, we did not enforce strict matches on the crash type and address, which may affect the accuracy of the ARVO dataset (i.e., it is possible that the crash we reproduce differs from the original vulnerability, despite sharing the same triggering input).

Patch Quality. ARVO's reliance on bisection for identifying vulnerability fixes has limitations. Due to the possibility of multiple related commits, this approach might not always accurately pinpoint the exact fix, particularly when fixes involve a series of modifications. Also, while a commit may lead us to a "correct" fix, an individual commit might encompass extensive modifications, complicating the identification of the precise change responsible for the fix.

Time-Consuming. Even with Docker images provided, re-compiling downstream applications remains costly. Although running the commands is straightforward, rebuilding the entire ARVO dataset from scratch is resource-intensive: our most recent rebuild, including patch locating, took roughly 4 weeks on a 192-core server with 256 GB of RAM. Patch locating further amplifies this cost, as failed commits can force linear search instead of bisection.

Duplicated Cases. The ARVO dataset contains cases where upstream reported multiple vulnerabilities that share the same underlying root cause. While these duplicates may appear redundant, they are valuable for patch verification, as their PoCs trigger crashes through different execution paths. We therefore preserve and label them rather than discarding them, to better test patch robustness. However, patch-commit-based labeling remains imprecise, since a single patch may simultaneously fix multiple vulnerabilities.

7.2. Future Work

The reproducer of ARVO has officially merged into OSS-Fuzz. In the near future, we will merge the patch locator into OSS-Fuzz and correct the corrupted metadata

on OSS-Fuzz to provide cleaner vulnerability information. In parallel, we will continue to maintain ARVO as an open-source project to fill the gap until upstream sources enforce reproducibility natively, at which point a separate ARVO would no longer be needed. To broaden its scope, we also plan to incorporate additional upstream sources, starting with the Linux kernel vulnerabilities, where more vulnerability information can be retrieved from the mailing history and commit messages.

Looking ahead, we will focus on making the reproduction process more reliable, improving the accuracy of ARVO dataset, and making ARVO easier to use as a dependable foundation for future security research until the day full reproducibility is introduced into upstream.

We also plan to extend ARVO's bug-insertion feature with prior patch-porting techniques [32], [33] to increase the number of bugs that can be successfully reintroduced onto a single target commit, and to explore LLM-assisted variants for cases where source-code changes between the bug's origin and the target commit defeat purely syntactic approaches.

8. Conclusion

In this paper, we present ARVO, an Atlas of Reproducible Vulnerabilities in Open-source software, establishing reproducibility as a core property for security datasets. We identified key challenges in vulnerability reproduction and proposed practical solutions, transforming document-centric reports into a reproducible database with interactive environments. Our open-source dataset and framework include more than 6,000 vulnerabilities across 311 projects, represented in over 12,000 interactive build images hosted on Docker Hub. Accuracy evaluation shows that 89.4% of located patches are correct, underscoring the reliability of our approach. By combining reproducibility, scale, and diversity, ARVO offers the most comprehensive dataset of its kind to date, with a framework designed to automatically incorporate new vulnerabilities and projects in the future.

Open Science

All artifacts necessary to reproduce the results in this paper are publicly available:

- **Source code:** https://github.com/n132/arvo
- **Dataset:** https://github.com/n132/ARVO-Meta
- **Evaluation data:** https://github.com/sefcom/ARVO

Ethics Considerations

We have found more than 300 potentially unfixed bugs and 2381 potential false positives, which we reported to OSS-Fuzz, and further responsible disclosure for the existing bugs is being processed. It is worth noting that throughout our experiments we only used publicly available data from the OSS-Fuzz issue tracker (fixes, PoCs, etc.), i.e. we have not found any new vulnerabilities.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback.

This material is based upon work supported by NSF 2247954, NSF 2146568, NSF 1663651, NSF 2442984, and NSF 2232915, by the Defense Advanced Research Projects Agency (DARPA) under Young Faculty Award D22AP00145-00, and by generous support from the US Department of Defense. This work is also sponsored by, and related to, Department of Navy award N00014-23-1-2563 issued by the Office of Naval Research, and supported in part by the Australian Government through the Australian Research Council's Discovery Projects funding scheme (project DP250101396). Hammond Pearce is supported in part by a Google Research Scholar award.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Defense, DARPA, or any other sponsor. The content of the information does not necessarily reflect the position or the policy of the Governments, and no official endorsement should be inferred.

References

[1] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[2] 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, 2016.

[3] Cyber Grand Challenge. CGC dataset. https://github.com/CyberGrandChallenge/samples, 2016. GitHub repository. Retrieved September 4, 2024.

[4] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, Baltimore, MD, August 2018. USENIX Association.

[5] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[6] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[7] Google. osv.dev. https://osv.dev/, 2024.

[8] DARPA. AI Cyber Challenge (AIxCC). https://aicyberchallenge.com, 2025.

[9] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI agents' cybersecurity capabilities with real-world vulnerabilities at scale, 2025.

[10] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. PATCHAGENT: A practical program repair agent mimicking human expertise. In *USENIX Security Symposium (USENIX Security '25)*, Seattle, WA, August 2025.

[11] CISA. CISA Open Source Software Security Roadmap. https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap, 2023.

[12] GitHub Staff. Octoverse: AI leads Python to top language as the number of global developers surges. https://github.blog/news-insights/octoverse/octoverse-2024/, 2024.

[13] Google. AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer, 2024.

[14] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. BScout: Direct whole patch presence test for java executables. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1147–1164. USENIX Association, August 2020.

[15] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. PDiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1149–1163, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 887–902, Baltimore, MD, August 2018. USENIX Association.

[17] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association.

[18] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1253–1270, Vancouver, BC, August 2017. USENIX Association.

[19] Guru Bhandari, Amara Naseer, and Leon Moonen. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, pages 30–39, New York, NY, USA, August 2021. Association for Computing Machinery.

[20] 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, page 426–437, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed OSS vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3282–3299, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology*, 30(2):1–27, March 2021.

[23] Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. The FormAI dataset: Generative AI in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2023, page 33–43, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2A: A dataset built for AI-based vulnerability detection methods using differential analysis. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 508–512, New York, NY, USA, September 2020. Association for Computing Machinery.

[26] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[27] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 654–668, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January 2012.

[29] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[30] syzbot. https://syzkaller.appspot.com/. Continuous fuzzing dashboard for the Linux kernel.

[31] 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*, CCS '17, pages 2201–2215, New York, NY, USA, October 2017. Association for Computing Machinery.

[32] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. Automated patch transplantation. *ACM Transactions on Software Engineering and Methodology*, 30(1), December 2021.

[33] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 633–645, New York, NY, USA, 2021. Association for Computing Machinery.

[34] OSS-Fuzz issue 42486945. https://issues.oss-fuzz.com/issues/42486945.

[35] libheif: Update readme. https://github.com/strukturag/libheif/commit/5f948947733b.

[36] libheif: Fix incorrect memcpy size. https://github.com/strukturag/libheif/commit/11ffeffadd98.

Appendix

Listing 1: Patch for Open Issue 42486945

```
commit 11
    ffeffadd980f9f96019fe180fc1e81827e3790
Author: Dirk Farin <dirk.farin@gmail.com>
Date:   Mon Apr 4 20:43:45 2022 +0200

    fix wrong memcpy size

diff --git a/libheif/heif_colorconversion.
    cc b/libheif/heif_colorconversion.cc
index 2b05068..5a07ebb 100644
--- a/libheif/heif_colorconversion.cc
+++ b/libheif/heif_colorconversion.cc
@@ -526,7 +526,8 @@ Op_YCbCr_to_RGB<Pixel
    >::convert_colorspace(const std::
    shared_ptr<const HeifPixel
      }

      if (has_alpha) {
-       memcpy(&out_a[y * out_a_stride], &
    in_a[y * in_a_stride], width * 2);
+       int copyWidth = (hdr ? width * 2 :
    width);
+       memcpy(&out_a[y * out_a_stride], &
    in_a[y * in_a_stride], copyWidth);
      }
    }
```