BinForge: Facilitating Generation of Diversified Binary Datasets

HyungJoon Yoon and Sang Kil Cha
KAIST
Daejeon, Republic of Korea
{topcue,sangkilc}@kaist.ac.kr

***Abstract*—Binary analysis increasingly depends on large and diverse binary datasets, yet constructing such datasets from open-source C/C++ projects remains difficult. Builds often fail across compiler versions, compiler flags, library versions, and target architectures, making existing datasets hard to reproduce or extend. We present BinForge, a system for automatically generating large binary datasets by managing toolchains and build configurations while using LLM-guided failure recovery to repair build errors without changing the intended build. Using BinForge, we construct 9,921,648 binaries from 151 open-source projects across 11 architectures, 19 compiler versions, and 6 optimization levels. To our knowledge, this is the largest and most diverse binary dataset for binary analysis to date. We demonstrate its value by evaluating TikNib, a state-of-the-art binary similarity analysis tool, and reveal previously unexplored challenges caused by compiler and architecture diversity. Both BinForge and the dataset are publicly available to support reproducible binary analysis research.**

***Index Terms*—Binary Analysis, Binary Dataset Generation, Build Automation, Build Issue Resolution**

I. Introduction

Binary analysis is central to understanding compiled programs when source code is unavailable. It supports tasks ranging from malware analysis [1]–[3], vulnerability discovery [4]–[8], and runtime security monitoring [9] to binary reassembly [10], [11] and decompilation [12], [13].

Progress in this area depends on binary datasets, i.e., collections of binary programs used to train, test, and compare analysis techniques. The rise of AI in binary analysis [14]–[17] has further increased the demand for large datasets.

Existing binary datasets are often difficult to reproduce or extend, either because key build details are missing or because complex build systems and dependencies make reproduction fragile. Yet reproducibility and extensibility matter in practice: one might want to recompile binaries with debugging information enabled (e.g., `-g`) to verify function entry points [18]–[21], or recompile projects with various combinations of compiler options to create a more diverse dataset for training robust machine learning models for malware variant detection [22], [23]. Achieving this reliably is difficult, however, since even small differences in library versions or system configurations can cause build failures, and the problem only worsens when using different compiler versions or cross-compiling for different architectures. Thus, the goal is not merely to compile individual projects, but to systematically generate many binaries under controlled variations of compilers, options, environments, and architectures.

Publicly available datasets such as BinKit [24], however, are often tied to specific environments, making it difficult to rebuild the same programs on a newer OS, with different compiler flags, or for additional architectures. Existing datasets therefore neither guarantee reproducibility nor provide the flexibility needed to extend them across environments. Package managers such as Gentoo's `Portage` [25] can help, but they remain environment-specific and still suffer from compilation challenges under diverse compiler options, as we discuss in §II.

A natural way to scale binary dataset construction is to automate the build failures that arise while compiling many project-configuration pairs. Prior work has repaired build failures in Maven, Gradle, and Android projects [26]–[28]; recent approaches based on large language models (LLMs) automate C/C++ project building [29]–[31]; and general-purpose AI agents can diagnose build errors. However, a successful project-level build alone is insufficient for large binary dataset generation across many target architectures and compilation options. Managed-ecosystem repair techniques do not address C/C++ cross-compilation and low-level configuration issues, while LLM-based C/C++ build approaches and general-purpose agents give LLMs broad control over the build process. The LLM-based approaches may change the environment or source code, build only part of a project, or use prebuilt packages. Such behavior conflicts with binary dataset generation, whose goal is to compile the same intended programs under controlled combinations of architectures and options, not merely to obtain any successful build. Consequently, resolving such failures still demands substantial manual effort: in our preliminary study (see §II), an experienced graduate student spent over two months rebuilding BinKit on a newer OS with updated compilers and project versions.

We introduce BinForge, a tool for constructing controlled, large-scale, and diverse binary datasets from open-source C/C++ projects. The key idea is to automate developers' trial-and-error process while limiting each LLM response to a structured build configuration update that can be reused across configuration variants—combinations of target architectures, compiler versions, optimization levels, and flag sets—and dependency builds. BinForge controls variants and build execution, provides a unified interface for compiler and linker

flags, integrates Crosstool-NG [32] for cross-compilation, and uses LLMs to refine configurations when errors occur.

Using BinForge, we construct the largest binary dataset to date: 9,921,648 binaries from 151 open-source projects, compiled for 11 architectures with 19 compiler versions and 6 optimization levels. Both BinForge and the dataset are publicly available at https://github.com/SoftSec-KAIST/BinForge. We demonstrate that this scale and diversity matter by evaluating TikNib, a state-of-the-art binary similarity analysis tool [33]: our dataset exposes previously unexplored challenges caused by compiler and architecture diversity.

Our contributions are summarized as follows:

- We characterize the challenges of producing diversified binary datasets by extending BinKit and evaluating existing LLM-based build approaches.
- We present BinForge, a tool that generates large binary datasets from C/C++ projects across compilers, flags, and target architectures.
- We create the largest binary dataset to date: 9,921,648 binaries from 151 projects across 11 architectures, 19 compiler versions, and 6 optimization levels.
- We use the dataset to evaluate TikNib [33] and reveal previously unexplored challenges in binary similarity analysis.

II. Motivating Study: Extending BinKit

To understand why build errors impede binary dataset reproduction and extension, we examine the manual effort required to extend BinKit [24] and why existing LLM-based build approaches are not yet sufficient to automate this process.

A. Challenges in Extending BinKit

BinKit (v2.0) [24], one of the few existing tools that releases both its build scripts and resulting binaries, contains 212K binaries compiled from 51 open-source projects. However, it remains tied to particular OS versions and compiler toolchains, making it difficult to extend the dataset to newer compilers, project versions, or additional architectures.

To confirm this, we attempted exactly that: running BinKit's build scripts on Ubuntu 24.04 instead of 16.04 while extending the dataset to newer compilers, project versions, and additional architectures. The attempt produced numerous build errors, and resolving them took an experienced graduate student more than *two months*, since even small fixes were time-consuming to identify and one fix could repeatedly expose another across configurations and dependencies.

Underlying this effort were five recurring categories of build errors. First, pre-built toolchains depended on unavailable system libraries; for example, Clang v13 required `libtinfo.so.5`, whereas Ubuntu 24.04 provides the incompatible `libtinfo.so.6`, forcing us to rebuild the entire toolchain. Second, newer compilers rejected legacy source patterns, requiring compatibility flags such as `-Wno-implicit-function-declaration`. Third, updated projects introduced dependencies that package managers did not always provide [34], [35]; for example, upgrading `a2ps` from v4.14 to v4.15.6 introduced a dependency on `bdw-gc`. Fourth, even after building a required library, target projects sometimes needed explicit linker flags such as `-ltinfo` to link against it. Finally, adding compiler flags to diversify the generated binaries revealed ordering constraints; for example, `-pie` had to precede `-shared`.

B. Limitations of LLM-Based Build Approaches

Given the manual effort these errors demanded, we ask whether existing LLM-based build solutions can reduce it, and find that they fall short for binary dataset generation.

Recent approaches [29]–[31] use LLMs and build feedback to automate individual C/C++ project builds. However, our evaluation of these approaches on 51 BinKit projects (§IV-D) shows that they may silently change the compiler, OS, or source code, or obtain prebuilt binaries instead of building the project from the provided source. Such behavior is unsuitable for binary dataset generation, which must build the provided source without modifying it or changing the compiler or OS.

General-purpose AI agents can directly inspect and modify the build environment, suggesting missing configuration options or retrying builds with different flags. However, our experience shows that directly applying them to binary dataset generation remains limited. They may bypass build issues by building only part of a project rather than resolving the underlying issues, and applying them across many builds can incur substantial cost, as discussed in §IV-D.

C. Takeaways

Our manual extension study and evaluation of LLM-based build approaches lead to four takeaways:

- Building C/C++ projects is highly sensitive to the environment: changes in OS, compiler, or library versions can introduce build errors.
- A single fix can expose new errors, creating a cascade that requires multiple rounds of troubleshooting.
- Many fixes require only small configuration changes, such as adjusting compiler or linker flags, but identifying the right change is time-consuming.
- Existing LLM-based C/C++ build approaches can introduce many uncontrolled changes, such as silently modifying source code or retrieving prebuilt packages from the Internet, undermining dataset validity.

III. BinForge Design

This section presents BinForge's architecture and explains how its components interact to respond to the findings in §II.

A. Architecture Overview

At a high level, BinForge takes as input a list of open-source C/C++ projects (`projs`), a list of target architectures (`archs`), a list of compilers and their versions (`compilers`), a list of optimization levels (`levels`), and a list of compiler flag sets (`flags`). It then produces (1) a binary dataset containing the compiled binaries for each project across all target architectures, compilers, optimization levels, and compiler flag

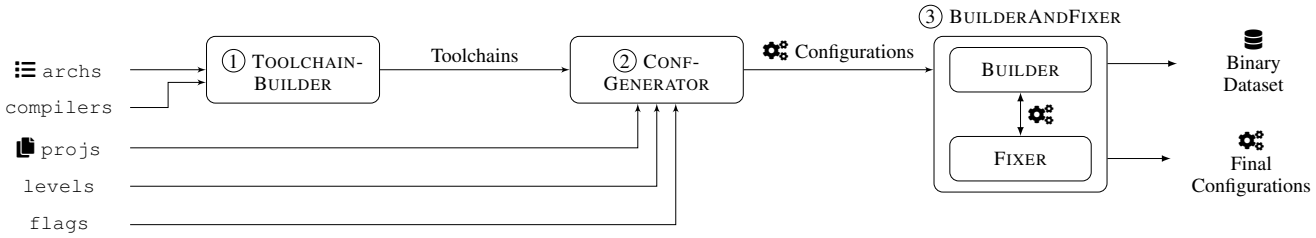


Fig. 1. Architecture of BinForge

sets, and (2) the final build configurations used to build them. The key innovation of BinForge is a structured interface that guides LLMs toward reusable configuration updates rather than one-off source edits or build-command changes.

Figure 1 shows the architecture of BinForge, which consists of three modules: ① ToolchainBuilder, ② ConfGenerator, and ③ BuilderAndFixer. ToolchainBuilder builds the compiler toolchains required for each target architecture. ConfGenerator generates an initial build configuration with default compiler and linker flags for each project and its dependencies. Finally, BuilderAndFixer compiles each project across all combinations of target architecture, compiler, optimization level, and compiler flags, producing the final binary dataset while iteratively fixing build errors.

As BuilderAndFixer resolves build errors, it updates the configuration files by adding missing library dependencies and adjusting compiler or linker flags, addressing the latter four categories observed in our BinKit extension study (§II-A). Thus, BinForge uses LLMs to produce reusable configuration updates rather than one-off source edits or build-command changes.

1) ToolchainBuilder*:* Reproducible binary dataset generation requires a matching compiler, linker, and related tools for each target architecture. ToolchainBuilder therefore prepares the required toolchains before building target projects. We use Crosstool-NG [32] as the basis for building these toolchains, but it requires two modifications. First, the latest Crosstool-NG lacks support for some targets we require, such as MIPS64 little-endian Linux. Second, cross-compiler builds may fail when they rely on the system's default compiler, for example due to incorrect library or tool search paths. We therefore add the missing target support to Crosstool-NG and bootstrap a host compiler toolchain before building cross-compilers for the target architectures.

2) ConfGenerator*:* ConfGenerator provides a structured interface for specifying how to build each project and its dependencies, including compiler and linker flags, and configure options that may need adjustment when resolving build issues. Initially, ConfGenerator generates configuration files (see §III-B) with default compiler and linker flags, along with user-supplied target architectures (`archs`), compilers (`compilers`), optimization levels (`levels`), and compiler flag sets (`flags`). As builds proceed, BuilderAndFixer updates these configurations to fix build errors.

3) BuilderAndFixer*:* BuilderAndFixer consists of a builder that attempts each configured build and a fixer that, on failure, analyzes the error and updates the configuration by adjusting compiler or linker flags or adding missing library dependencies. Because fixes are recorded in the configuration, they can be reused across configuration variants and dependency projects. This iterative process continues until all builds succeed or a maximum number of attempts is reached. BuilderAndFixer then outputs the final binary dataset and the exact configuration used for each build, enabling reproduction and exposing the adjustments made during repair. We further detail the design of BuilderAndFixer in §III-C.

B. Build Configuration

A build configuration (or simply configuration) specifies how to build a project and its dependencies. We represent it in YAML for human readability and ease of parsing. The configuration records each project's source URL, version, dependencies, configure options, and compiler and linker flags.

Given a configuration, `EnumerateAllVariants` enumerates all combinations of target architecture, compiler, optimization level, and compiler flag for the project (Algorithm 1). We refer to each unique combination of target architecture, compiler, optimization level, and compiler flag as a *configuration variant* throughout this paper. Unlike traditional build systems that target a fixed set of build configurations, BinForge compiles all requested configuration variants for supported open-source C/C++ projects with minimal manual effort.

The structured configuration provides a clear interface for specifying project and dependency builds, resolving failures by adding missing dependencies or adjusting compiler and linker flags, and reusing successful fixes across configuration variants and dependency projects.

1) Supported Configuration Variants: Beyond project-level build parameters, BinForge defines configuration variants that capture the diversity of build environments commonly observed in real-world software. These built-in variants cover a wide range of compiler and platform combinations without manual intervention.

Target architectures. BinForge currently supports 11 architectures spanning x86, ARM, MIPS, and PowerPC families, including both little- and big-endian variants: x86, x86_64, arm_32, aarch64, mips_32el, mips_64el, mips_32be, mips_64be, ppc_32be, ppc_64be, and ppc_64le.

Algorithm 1: Pseudocode of BUILDERANDFIXER.

```
1  function BuilderAndFixer(conf)
2      n ← 0; C ← {·}; B ← ∅
3      foreach v ∈ EnumerateAllVariants(conf) do
4          C[v] ← false
5      while (∃v ∈ C : C[v] = false) ∧ (n < max_iter) do
6          foreach v ∈ EnumerateAllVariants(conf) do
7              if C[v] then
8                  continue
9              (succeed, errlog, bins) ← Build(conf, v)
10             if succeed then
11                 C[v] ← true
12                 B ← B ∪ bins
13             else
14                 conf ← QueryLLM(errlog, conf)
15                 break
16         n ← n + 1
17     return (B, conf)
```

Compilers. BinForge integrates 13 versions of Clang (v7.1.0-v19.1.7) and 6 versions of GCC (v8.5.0-v13.3.0), all of which are supported by the toolchain builder (§III-A1). This range of compiler versions allows BinForge to generate binaries under diverse compiler behaviors, including deprecated flags and evolving language semantics.

Optimization levels. BinForge supports the six standard optimization levels widely used across compilation workflows: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast`.

Compiler-flag variants. To further increase diversity, BinForge defines eight representative combinations of orthogonal compiler flags that affect binary structure and runtime properties. These flags govern position-independent executable (PIE) generation, function inlining, and stack protection. The combinations range from the baseline (`-no-pie`, compiler-default inlining, and `-fno-stack-protector`) to configurations that enable any subset of PIE (`-pie`), forced no-inlining (`-fno-inline`), and stack protection (`-fstack-protector-all`).

By combining all supported architectures, compilers, optimization levels, and flag sets, BinForge expands each original project binary into 10,032 distinct variants ($10{,}032 = 11$ architectures $\times$ 19 compilers $\times$ 6 optimization levels $\times$ 8 compiler flag sets). This expansion yields a highly diverse set of binaries that reflects a wide range of compilation conditions.

C. Automatic Build Fixing

The BUILDERANDFIXER module uses LLMs to automate developers' trial-and-error process by searching for solutions and updating build configurations until the build succeeds or the maximum number of attempts is reached.

1) The Algorithm: Algorithm 1 shows the overall procedure of BUILDERANDFIXER. It takes a configuration for a single project as input, and returns a set of binaries successfully built for the project across all target architectures, compilers, optimization levels, and compiler flag sets, along with the final configuration used to build them. The same algorithm runs for each project to produce the final binary dataset.

Lines 2–4 initialize the iteration count n, a map C that marks each configuration variant as not yet built, and a set B that stores successfully built binaries. The `while` loop in Line 5 continues while a configuration variant remains unbuilt and n is less than `max_iter`. In our implementation, `max_iter` is set to 20, based on the BinKit calibration run described in §IV-A. Line 6 iterates over each unbuilt variant v, and Line 9 attempts the build and returns its status (`succeed`), error log (`errlog`), and binaries (`bins`). If the build succeeds, it marks v as successfully built and adds the built binaries to B in Line 10–12. If the build fails, Line 14 queries an LLM with the error log and current configuration, then restarts the process with the updated configuration. Each configuration update is scoped to the failed variant's build context, so it does not affect variants already built successfully. We detail the prompt design in §III-C2.

2) LLM-based Query Design: The `QueryLLM` function in Line 14 of Algorithm 1 uses a structured prompt that guides the LLM to analyze the error message and suggest configuration modifications in a consistent YAML format. The Markdown prompt has seven sections, described below.

The Overview section summarizes the task. The Role section defines the LLM as an expert software engineer in C/C++ build systems responsible for resolving build errors. The Error Categories section lists common build error categories, including missing dependencies, incompatible compiler flags, and incorrect linker flags, to focus the LLM on relevant issues. The Solution Generation Rules section requires addressing one error category at a time and specifies which configuration field to modify for each error type. These rules keep the LLM output within configuration changes, such as dependencies, compiler flags, linker flags, and configure options. The Output Format section specifies the YAML structure so that BinForge can directly parse and apply the LLM output. The Error Logs section contains the error message from the failed build. The Build Context section provides the target architecture, compiler, optimization level, and compiler flag set for the current build.

D. Scope of BinForge

BinForge currently assumes that target projects are written in C or C++ and use either the GNU Autoconf or CMake build system. Also, those target projects should allow the user to configure (1) the target architecture, (2) the compiler flags (CFLAGS), and (3) the linker flags (LDFLAGS). Specifying the target architecture is crucial for cross-compilation, as it allows the compiler to generate code for a different architecture than the one it is running on [36]. Both Autoconf and CMake allow the user to specify these options either through the command line (e.g., `--host` option in `configure` scripts) or through the configuration file (`CMAKE_SYSTEM_PROCESSOR`). It is noteworthy that not every project supports these options. For example, `ed` (v1.21), although it was released recently in Jan. 2025, uses a hard-

coded `configure` script (i.e., it does not use GNU Autoconf) and does not support the `--host` option. Such projects are not our target for now as there is no straightforward way to cross-compile them. Please refer to §V for more discussion on the limitations of BinForge.

E. Implementation

We implemented the core of BinForge in about 5K SLoC of Python. BinForge uses Crosstool-NG [32] (v1.27.0) to build GCC toolchains, and supports 13 versions of Clang (v7.1.0-v19.1.7) and 6 versions of GCC (v8.5.0-v13.3.0). We used `gpt-5.3-chat-latest` through the OpenAI API for automatic build fixing.

IV. Evaluation

This section evaluates BinForge to answer the following research questions.

RQ1. How reliably can BinForge compile a curated benchmark of open-source software projects? (§IV-B)
RQ2. How reliably can BinForge compile diverse and complex real-world software projects? (§IV-C)
RQ3. Can existing LLM-based build approaches replace BinForge for binary dataset generation? (§IV-D)
RQ4. How large and diverse is the dataset generated by BinForge compared to state-of-the-art benchmarks? (§IV-E)
RQ5. Does the expanded dataset generated by BinForge affect the performance of binary analysis tools? (§IV-F)
RQ6. What can we learn from the expanded dataset? (§IV-G)

A. Evaluation Setup

In the BinForge build experiments, a project is considered successfully built only if its configuration, compilation, and installation phases complete without errors for all requested configuration variants. We further use B2R2 [37] to validate the file format and target ISA of the generated binaries. If any phase fails, BinForge attempts to repair the issue and rebuild the project using its automatic build-issue resolution algorithm (Algorithm 1). In the experiments in §IV-B and §IV-C, when BinForge needs the source code for a dependency, it helps the LLM find a source package or source URL by querying `apt-cache` through `APT`; we call this support dependency source lookup. We evaluate its impact through an ablation on projects with at least one dependency, where BinForge disables this support and the LLM must infer source URLs from its parametric knowledge. For the automatic building experiments in §IV-B and §IV-C, we used a server with 88 Intel Xeon E5 cores and 512 GB of RAM. For running binary analysis tools in §IV-F and §IV-G, we used a desktop with an Intel i9-11900K 3.50GHz CPU and 64 GB of RAM.

We used the BinKit evaluation to calibrate `max_iter` in Algorithm 1: among the projects with build issues, `inetutils` required the most repair attempts, reaching 10, so we set `max_iter` to 20 for the remaining experiments, twice the maximum observed in the BinKit run.

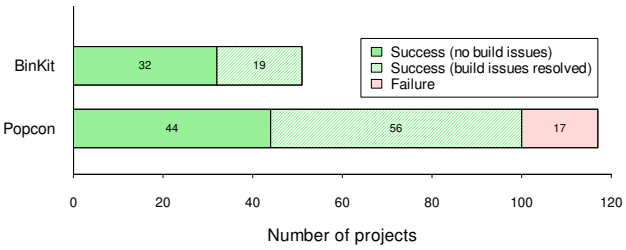


Fig. 2. Breakdown of build outcomes for BinKit (§IV-B) and Debian Popularity Contest (§IV-C) projects.

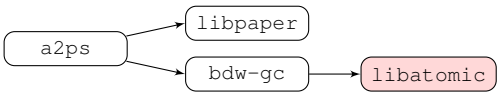


Fig. 3. Dependency graph constructed for `a2ps` in the dependency source lookup ablation. BinForge inferred the chain but failed to find the source URL for `libatomic`.

B. Building Curated Benchmarks

We evaluated BinForge by modernizing and expanding the BinKit benchmark [24]. BinForge rebuilt the pinned versions of the original BinKit projects in a clean Docker container running Ubuntu 24.04.4 LTS; the original BinKit dataset was built on Ubuntu 16.04. We also expanded the configuration space from 2,080 variants in the original BinKit to 10,032 variants in our setting, covering all supported combinations of target architectures, compilers, optimization levels, and compiler-flag sets defined in §III-B1.

1) BinKit Build Results: On this curated benchmark, BinForge successfully built all 51 BinKit projects without manual intervention: 32 projects (62.7%) succeeded on the first attempt, while the remaining 19 (37.3%) required build-error repairs that BinForge applied automatically. The first bar in Figure 2 shows the build outcomes of the 51 BinKit projects.

Thus, BinForge modernized BinKit and expanded it to all supported configuration variants, automatically resolving the resulting build failures, in contrast to the two-month manual effort described in §II-A.

2) Ablation of Dependency Source Lookup: We disabled dependency source lookup through `APT` and reran the experiment on the six BinKit projects with dependencies, which together require seven unique dependencies. BinForge still built 5 of the 6, failing only on `a2ps`. As Figure 3 shows, BinForge correctly inferred the `a2ps` dependency chain (`libpaper` and `bdw-gc` as direct dependencies, `libatomic` as a dependency of `bdw-gc`), but could not find the correct source URL for `libatomic`, causing the build to fail. This shows that dependency source lookup complements dependency-chain inference by reliably obtaining the corresponding source code.

3) Takeaways: The BinKit experiment answers RQ1: BinForge rebuilt all 51 BinKit projects on a modern environment across the expanded configuration space without manual intervention. The ablation further shows that dependency source lookup mitigates the limitations of relying on the LLM's memorized source URLs.

C. Building Real-World Software

Although BinKit contains a diverse set of open-source projects used in prior work, it remains a curated benchmark and may not capture the complexity of real-world software at scale. We therefore ask whether BinForge generalizes to larger and more complex real-world projects. To answer this, we apply BinForge to a set of C/C++ projects from Debian using all the configuration variants described in §III-B1.

1) Selection of Target Projects: To select representative real-world projects, we used the Debian Popularity Contest (Popcon) [38], which reports package installation frequency among Debian users. We selected the 220 target projects as follows.

1) Sort packages by popularity and select the top 1,000.
2) Download the corresponding source packages with `APT`, skipping 118 packages whose source code was unavailable or inaccessible. This left 882 packages with accessible source code.
3) Remove 311 packages that share a source package with another (e.g., `python3` and `libpython3-stdlib`) and 12 packages overlapping with BinKit, leaving 559 distinct source packages.
4) Filter out 339 projects outside the scope of BinForge (§III-D), such as non-C/C++ projects, resulting in 220 in-scope source packages.

2) Popcon Build Results: We randomly sampled projects from the 220 candidates and ran BinForge until it produced 100 successful builds. Reaching this stopping point required attempting 117 projects and 17.45 days of continuous execution on our server machine. The runtime was dominated by compiling each project across all configuration variants, which is necessary for binary dataset generation. Thus, even if all successful build fixes were known in advance, the total runtime would remain largely unchanged. To keep the experiment tractable, we stopped after obtaining 100 successful builds.

Across the 117 attempted projects, BinForge successfully built 100 projects (85.5%) without any manual intervention, while the remaining 17 projects failed after exhausting the repair attempts. Of the 100 successful builds, 44 completed on the first attempt without any build issues and therefore required no LLM intervention. The remaining 56 successful builds required at least one build-error repair, which BinForge applied automatically. The second bar in Figure 2 summarizes these outcomes. Although this success rate is lower than the 100% success rate on BinKit, it remains strong given the greater diversity and complexity of the Popcon projects, as discussed in §IV-C3.

Table I summarizes our manual classification of the 17 failed Popcon projects. Two failure categories involve dependencies. *Required dependency build failure* covers cases where a target project failed because a required dependency failed to build; for example, `flpsed` depends on `fltk`, whose dependency `libx11` failed to build. *Out-of-scope dependency* covers cases where a required dependency falls outside the scope of BinForge; for instance, `heros` depends on `allegro`, which requires `jack`, but `jack` only supports the `waf` build system. The remaining three categories involve missing or incorrect build-configuration values: configure options, compiler flags, and linker flags. For example, BinForge correctly identified and compiled `sdl` for `bumprace`, but failed to pass the required configure option `--with-sdl-prefix`.

TABLE I
FAILURE CAUSES FOR THE 17 FAILED POPCON PROJECTS.

Failure Cause	#Projects	Ratio
Required dependency build failure	6	35.3%
Missing or incorrect configure options	5	29.4%
Out-of-scope dependency	3	17.6%
Missing or incorrect compiler flags	2	11.8%
Missing or incorrect linker flags	1	5.9%
Total	17	100.0%

3) Why Popcon Projects Are Harder to Build: Is the lower success rate on Popcon due to the projects being inherently harder to build than those in BinKit? We answer this by comparing the repair rates and dependency structures of the two datasets.

First, Popcon projects required repairs more often: 56 out of 100 successful builds needed repair, compared with 19 out of 51 BinKit builds. This suggests that Popcon projects more often require fixes such as adding missing dependencies or adjusting compiler and linker flags, as observed in our BinKit extension study (§II-A), leading to more frequent LLM interventions.

Second, Popcon projects also have more complex dependency structures than BinKit projects. We measure this by extracting dependency graphs from BinForge's configuration files for the successfully built projects in both datasets. On average, Popcon projects have a dependency graph size of 1.59 nodes and depth of 0.34, compared to 1.18 nodes and depth of 0.14 for BinKit projects. This structural complexity—deeper and more heterogeneous dependency chains in Popcon—naturally leads to more frequent build issues and explains the higher repair rate observed.

4) Ablation of Dependency Source Lookup: We disabled dependency source lookup through `APT` and reran the experiment on the 32 successful Popcon projects that require at least one dependency. Under this setting, BinForge successfully built 25 of these 32 projects (78.1%). The remaining 7 projects (21.9%) failed because BinForge could not find the correct source URL for at least one dependency. This result mirrors the limitation observed in §IV-B2 at a larger scale.

5) Takeaways: The Popcon experiment shows that BinForge generalizes to real-world software, automating 100 of 117 builds. The remaining failures reflect the diversity of Popcon packages, including heterogeneous build systems, deeper dependency chains, and distribution-specific toolchain assumptions.

TABLE II
PROJECT-LEVEL BUILD RESULTS ON THE 51 BINKIT PROJECTS, AGGREGATED OVER 10 RUNS. EACH CELL SHOWS #PROJECTS BUILT (AND SUCCESS RATE).

Approach	Avg.	Min.	Max.
CXXCrafter [29]	36.6 (71.8%)	31 (60.8%)	40 (78.4%)
CompileAgent [30]	11.8 (23.1%)	10 (19.6%)	13 (25.5%)
ReflectBuild [31]	45 (88.2%)	45 (88.2%)	45 (88.2%)
BinForge	51 (100.0%)	51 (100.0%)	51 (100.0%)

D. Comparison with LLM-Based Build Approaches

We now compare BinForge against existing LLM-based build approaches to answer RQ3. Since BinForge has already built both the BinKit (§IV-B) and Popcon (§IV-C) datasets, we can draw each comparison from whichever dataset most sharply contrasts BinForge with the approach at hand, rather than rerunning every baseline on every dataset. We therefore make two comparisons, each matched to what the compared approach is capable of.

The first comparison targets three specialized C/C++ build approaches: CXXCrafter [29], CompileAgent [30], and ReflectBuild [31]. These approaches build a project only under a single default configuration and do not support building across multiple configuration variants, so it suffices to test them on the 51 BinKit projects from §IV-B, checking whether they can produce even one working build per project. This isolates project-level build capability, whereas BinForge additionally builds each project across all configuration variants, which is necessary for binary dataset generation.

The second comparison targets a general-purpose AI agent, which, unlike the specialized approaches, can be directed at multiple configuration variants and thus be held to the same goal as BinForge. Because this comparison hinges on build repair rather than default builds, we run it on the 73 Popcon projects that required at least one repair from BinForge in §IV-C3, the cases that most directly exercise this capability.

1) Specialized C/C++ Build Approaches: To reduce differences due to model capability, we updated CXXCrafter and ReflectBuild to use `gpt-5.3-chat-latest`, the same model used by BinForge. Because CompileAgent is designed to use models from different families, we preserved the DeepSeek, GPT, and Claude families and used their available releases closest to GPT-5.3: DeepSeek v4 Pro, GPT-5.3, and Claude Sonnet 4.5. We repeated each approach 10 times, with every run covering all 51 projects, and counted a build as successful only if it built the provided source and produced the expected binaries after manual verification.

As shown in Table II, ReflectBuild ranked highest, building 45 of 51 projects in every run, while CompileAgent averaged only 11.8. Analyzing the failures, we found that all three approaches often bypassed a genuine source build rather than resolving the underlying issue. CompileAgent declared success even when no binaries were produced (e.g., `cpio`, `direvent`). ReflectBuild installed a target package, copied a system binary, or replaced the source—for instance, installing `libtool` and `texinfo` through `APT`, copying `/bin/tar` for `tar`, and replacing the `gcal` source. CXXCrafter modified source files or altered the build environment—for example, installing `sharutils` and `gcal` through `APT`, removing gnulib from `gnu-pw-mgr`'s `bootstrap.conf`, and downgrading from Ubuntu 22.04 to 20.04 and GCC 11 to GCC 9 to build `plotutils`.

Even on this lenient project-level criterion, none of the approaches matched BinForge, which built all 51 projects (100%) while the best competitor, ReflectBuild, topped out at 45 (88.2%). Moreover, BinForge reached this perfect success rate under a far stricter criterion: it built all 51 projects not for a single project-level build but across all 10,032 configuration variants of each project, and it did so without changing the provided source code, selected compiler, or OS (recall §IV-B).

2) General-Purpose AI Agent: As the general-purpose agent, we used the OpenAI API with `gpt-5.3-codex`, which could inspect build logs, run shell commands, install or build missing dependencies, change build options, and retry builds. We allowed the agent to cache its own repairs and reuse them across later variants of the same project. Unlike BinForge, however, the agent manages these cache files on its own rather than recording repairs as structured configuration changes.

As Table III shows, the AI agent built 43 of 73 projects (58.9%), below BinForge's 56 (76.7%); both fall below the 85.5% of §IV-C because we consider only 73 projects that required repairs. Analyzing the 30 failed projects, we found that the agent often bypassed build issues through configuration reductions or partial builds rather than resolving the underlying causes. For `libblkid1`, it disabled most components and produced only two binaries, whereas BinForge produced 101; for `libnetfilter-conntrack3`, it built only selected subdirectories instead of fixing the failure. These behaviors make direct agent use unreliable for binary dataset generation.

Directly using an AI agent is also substantially more expensive than using BinForge. As shown in the same table, the agent used 61.9× more tokens and incurred 17.7× higher cost than BinForge, while achieving a lower success rate. Thus, direct agent use is both less effective and more costly.

Together, these results answer RQ3: the evaluated approaches do not replace BinForge; the project-level approaches lack controls for configuration variants, while direct agent use is less effective and more costly.

E. Dataset Scale and Diversity Comparison

The successfully built BinKit and Popcon projects (§IV-B and §IV-C, respectively) collectively form the BinForge

TABLE III
COMPARISON OF BINFORGE AND AN AI AGENT ON 73 POPCON PROJECTS THAT REQUIRED AT LEAST ONE BUILD-ISSUE REPAIR.

Experiment	#Projects	#Success (Rate)	Tokens Used	Cost
BinForge	73	56 (76.7%)	12.97M	$26.8
AI agent	73	43 (58.9%)	803.68M	$474.3

TABLE IV
COMPARISON WITH PRIOR WORK ON BINARY DATASET GENERATION.

Work	#Bins.	#Projs.	#Bin/#Proj†	#Comp.	#Opts.	#Arch.	#Flags	Automated Build Repair
BinKit 1.0 [33]	243,128	51	4,767.2	13	5	8	4	✗
BinKit 2.0 [24]	212,770	51	4,172.0	18	**6**	8	1	✗
Assemblage [39]	1,536,171	**220,792**	7.0	6	4	2	1	✗
Decompile-Bench [40]	85,233	3,961	21.5	1	4	1	1	✗
R^3-Bench [41]	24,941	4,785‡	5.2	–§	2	2	1	✗
BinForge ✱	**9,921,648**	151	**65,706.3**	**19**	**6**	**11**	**8**	✓

† This is the ratio between the number of binaries and the number of projects in the dataset, which indicates the diversity of binaries we can generate for each project. Higher the ratio, the binary dataset is more diverse, hence, more suitable for binary analysis tasks.
‡ Of the 7,462 projects processed for R^3-Bench, 4,785 contribute 24,941 binaries to the final corpus. We use the latter project count in this comparison.
§ R^3-Bench uses the toolchain selected by each Nix package and does not report a fixed set of compiler variants.

dataset. The dataset includes only projects that completed fully automated builds, including automated build failure resolution where needed, ensuring that every binary is reproducible from source without human intervention and providing a practical foundation for large-scale binary analysis and learning-based studies.

1) Baseline Binaries in the Dataset: We call each distinct binary output of a project a baseline binary. Their number varies widely across projects. For example, `a2ps` produces one baseline binary, whereas `coreutils` produces 103. In total, the successfully built BinKit and Popcon projects produce 267 and 722 baseline binaries, respectively.

These baseline binaries form the starting point for expansion into the full set of configuration variants. BinForge generates 10,032 variants for each baseline binary, covering all architecture-compiler-optimization-flag combinations. Thus, the two groups expand into 267 $\times$ 10,032 and 722 $\times$ 10,032 unique binaries, respectively, contributing substantially to the overall scale and diversity of the dataset.

2) Comparison with Existing Datasets: Table IV compares BinForge with prior work on binary dataset generation: BinKit v1.0 [33], BinKit v2.0 [24], Assemblage [39], [42], Decompile-Bench [40], and R^3-Bench [41].

It is noteworthy that the dataset generated by BinForge contains 6.5$\times$ as many binaries as Assemblage. In contrast, Assemblage includes 1,462$\times$ as many projects, reflecting the study's focus on large-scale repository crawling. We define *diversity* as the number of distinct binaries that can be produced from the same source code, which is crucial for training and evaluating binary analysis models [23]. To quantify this, we calculated the ratio of binaries to projects in each dataset. The dataset generated by BinForge has the highest ratio among the compared datasets, indicating that it provides the most diverse set of binaries per project. This strength stems from BinForge reliably supporting multiple compilers, architectures, optimization levels, and flag combinations shown in the fifth through eighth columns of Table IV. Overall, BinForge is the only work to provide a dataset that is both significantly larger and more diverse than existing benchmarks.

F. Impact on Binary Analysis Tools

To assess the impact of the BinForge dataset, we ran TikNib [33], a state-of-the-art binary code similarity analysis tool, and compared its results on BinForge and BinKit [24]. TikNib uses a simple linear model with a well-designed feature set for accurate function-level binary code similarity analysis.

Because TikNib assumes a Linux environment, we ran it with IDA Pro v7.7 under Windows Subsystem for Linux (WSL) and made minor changes to path variables.

BinForge$^-$ is a subset of BinForge that consists only of the projects successfully built from the Debian Popcon selection (§IV-C), excluding all BinKit targets. Following the original paper, we used 10-fold cross-validation and reported accuracy using ROC AUC. We partitioned each dataset by configuration, including optimization level, target architecture, and compiler version, and evaluated selected subset pairs. For each pair, TikNib finds the most similar function in one subset for every function in the other and reports the average accuracy.

Table V summarizes the results obtained from various pairs of configurations. For example, the "O0 vs. O3" column shows the average accuracy of TikNib when it tries to find similar functions from binaries compiled with `-O0` and `-O3`. The "Opti All," "Arch All," and "Comp All" columns report the average accuracy across all pairs of optimization levels, target architectures, and compiler versions, respectively, while "All" reports the average across all configuration pairs.

Overall, we observe that the accuracy of TikNib is significantly lower when using the expanded datasets generated by BinForge compared to when using BinKit. For example, the ROC AUC values for BinKit, BinForge, and BinForge$^-$ were 0.95, 0.84, and 0.86, respectively.

The same trend appears across all configuration pairs. We believe the lower accuracy stems from the broader compiler and flag diversity and the additional PPC32 and PPC64 architectures in BinForge. For example, considering PowerPC (PPC) binaries significantly decreases the accuracy of TikNib because they include traceback tables [43] within the `.text` section that TikNib can decode as false instructions.

All these results suggest that the performance of binary analysis tools can be significantly affected by considering more diverse binaries. It also confirms our motivation that expanding binary datasets with diverse configurations is crucial for training better binary analysis models.

TABLE V
BINARY CODE SIMILARITY ANALYSIS RESULTS ON BINKIT AND BINFORGE DATASETS IN TERMS OF ROC AUCS.

		Optimization						Architecture						Compiler			
Dataset	All	Opti All	O0 vs. O3	O0 vs. Os	O1 vs. Os	O2 vs. O3	O3 vs. Os	Arch All	Bits (32-bit vs. 64-bit)	Endian	ARM vs. MIPS	x64 vs. ARM	x64 vs. MIPS	Comp All	GCC vs. Clang	Clang (oldest vs. latest)‡	GCC (oldest vs. latest)‡
BinKit [33]	0.95	0.96	0.93	0.96	0.99	0.99	0.97	0.98	0.99	1.00	0.98	1.00	0.98	0.98	0.97	1.00	1.00
BinForge	0.84	0.91	0.83	0.85	0.93	0.97	0.94	0.89	0.94	0.91	0.96	0.98	0.96	0.93	0.89	0.94	0.97
†**BinForge**$^{-}$	0.86	0.93	0.85	0.87	0.95	0.98	0.95	0.91	0.94	0.94	0.96	0.98	0.96	0.95	0.90	0.96	0.98

† This dataset consists of the projects that BinForge automatically built from the Debian Popcon selection (§IV-C), excluding all BinKit projects.
‡ These columns show the performance differences between the oldest and the latest compiler versions used in each dataset. BinKit used Clang (v4 vs. v7) and GCC (v4 vs. v8), while BinForge used Clang (v7 vs. v19) and GCC (v8 vs. v13).

G. Lessons Learned

To answer RQ6, we compare the original and expanded datasets and revisit whether the conclusions of the original TikNib paper [33] still hold.

First, we note that TikNib's accuracy does not change significantly when more projects are added: Table V shows that BinForge and BinForge^{-} have similar ROC AUCs for every configuration pair. On the other hand, the accuracy drops significantly when adding new compiler versions, optimization levels, and target architectures to the dataset as we can see from the comparison between BinKit and BinForge.

> The quality of a binary dataset can be more important for binary analysis than the quantity of it.

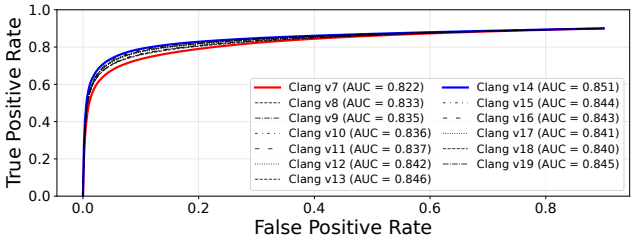
Second, the expanded dataset challenges TikNib's conclusion that compiler version has little impact on its performance. The last two columns of Table V show ROC AUC = 1.00 across compiler versions in BinKit, whereas TikNib's accuracy decreases across GCC and Clang versions in BinForge.

To investigate this effect, we evaluated TikNib on each compiler version separately. Figure 4 reports the resulting ROC AUCs for Clang and GCC, marking the best and worst cases in blue and red. From the figure, we observe significantly larger variations in the ROC AUC values when considering different versions of Clang compared to GCC. This means that Clang changes the way it emits binary code more frequently than GCC, which can substantially affect the performance of binary analysis tools.

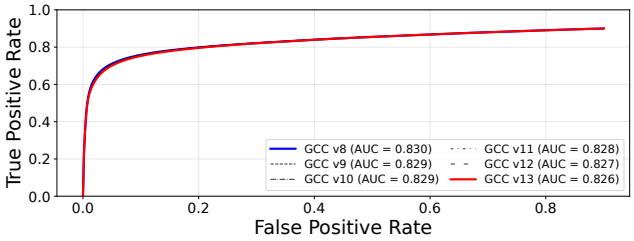
> Unlike the conclusion drawn from the previous work, we observe that the impact of compiler versions on the performance of binary analysis can be significant, and hence, accommodating new compiler versions is crucial to maintaining the performance of binary analysis tools.

V. LIMITATION

BinForge currently supports projects that use Autoconf or CMake, as discussed in §III-D. This scope allows BinForge to configure projects through a common interface for setting compiler and linker flags, which is necessary for generating diverse binary datasets. Although this requirement excludes some projects, many C/C++ projects satisfy it in practice. Supporting arbitrary build systems is beyond the scope of this paper because projects that do not follow conventions from well-known build systems often lack a standard way to expose the build options that BinForge needs to adjust.



(a) Clang



(b) GCC

Fig. 4. ROC AUCs for different versions of Clang and GCC.

VI. RELATED WORK

Empirical Studies of Build Failures. Huang *et al.* [34] analyzed 303 issues from build experiments on 12 open-source projects and found that environment- and test-related problems were frequent and difficult for non-contributors to resolve. A follow-up study [35] showed that providing essential information improved build success, highlighting the importance of project knowledge and effective troubleshooting.

Build Repair in Other Ecosystems. BuildMedic [26] repairs Maven dependency failures, HoBuFF [27] repairs Gradle

scripts, GradleFixer [28] repairs Android builds using an LLM agent, and EvidenT [44] repairs Open Build Service packages. These primarily target ecosystems with managed build environments rather than C/C++ projects, making them orthogonal to BinForge.

Automated C/C++ Project Building. CXXCrafter [29] extracts build information and iteratively generates and updates a Dockerfile to build C/C++ projects. CompileAgent [30] uses tool-integrated agents to inspect repositories, follow build instructions, and resolve errors. ReflectBuild [31] uses feedback between researcher and builder agents to automate C++ builds. These approaches target a single working build of a project, whereas BinForge builds each project across architectures, compilers, optimization levels, and flags using reusable build configurations.

Binary Dataset Generation. BinKit [33] is a large-scale binary similarity benchmark covering diverse compilation options and architectures, while Assemblage [39] crawls public repositories to construct a large Windows and Linux binary corpus. More recently, Decompile-Bench [40] provides binary-source function pairs for decompilation, while R^3-Bench [41] provides C/C++/Rust binaries for symbol recovery. None of these studies integrates automated build failure resolution into dataset generation.

VII. Conclusion

In this paper, we presented BinForge, the first system to generate binary datasets for binary analysis research while automatically resolving build failures across configuration variants. Using BinForge, we created a dataset of 9,921,648 binaries from 151 real-world open-source projects across 11 architectures, 19 compiler versions, and 6 optimization levels, automatically resolving 111 build errors across 56 projects. Using this dataset, we uncovered additional challenges in binary analysis, including the impact of compiler versions on binary code similarity analysis. Both our tool and dataset are publicly available at https://github.com/SoftSec-KAIST/BinForge.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02263143, Development of Cybersecurity Threat Response Technologies for Satellite Ground Stations, and No. 2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] J. Caballero, P. Poosankam, S. McCamant, D. Babić, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2010, pp. 413–425.

[2] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable malware clustering based on static features," in *Proceedings of the USENIX Annual Technical Conference*, 2013, pp. 187–198.

[3] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 81–96.

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[5] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: cross-architecture cross-os binary search," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 678–689.

[6] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2016, pp. 266–280.

[7] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2017, pp. 346–359.

[8] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[10] H. Kim, S. Kim, J. Lee, K. Jee, and S. K. Cha, "Reassembly is hard: A reflection on challenges and strategies," in *Proceedings of the USENIX Security Symposium*, 2023, pp. 1469–1486.

[11] H. Kim, S. Kim, and S. K. Cha, "Towards sound reassembly of modern x86-64 binaries," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025, pp. 1317–1333.

[12] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 353–368.

[13] J. Wiedemeier, E. Tarbet, M. Zheng, S. Ko, J. Ouyang, S. K. Cha, and K. Jee, "PyLingual: Toward perfect decompilation of evolving high-level languages," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2025, pp. 2976–2994.

[14] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 99–116.

[15] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proceedings of the USENIX Security Symposium*, 2022, pp. 2099–2116.

[16] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *Proceedings of the USENIX Security Symposium*, 2022, pp. 4327–4343.

[17] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O'Boyle, "SLaDe: A portable small language model decompiler for optimized assembly," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2024, pp. 67–80.

[18] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 611–624.

[19] C. Pang, R. Yu, D. Xu, E. Koskinen, G. Portokalidis, and J. Xu, "Towards optimal use of exception handling information for function detection," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2021, pp. 338–349.

[20] H. Kim, J. Lee, S. Kim, S. Jung, and S. K. Cha, "How'd security benefit reverse engineers? the implication of intel CET on function identification," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2022, pp. 559–566.

[21] S. Kim, H. Kim, and S. K. Cha, "FunProbe: Probing functions from binary code through probabilistic analysis," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2023, pp. 1419–1430.

[22] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2021, pp. 142–157.

[23] V. Singhal, A. A. Pillai, C. Saumya, M. Kulkarni, and A. Machiry, "Cornucopia: A framework for feedback guided generation of binaries,"

in *Proceedings of the International Conference on Automated Software Engineering*, 2023, pp. 1–13.
[24] "BinKit," https://github.com/SoftSec-KAIST/BinKit.
[25] "Portage," https://wiki.gentoo.org/wiki/Portage.
[26] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 106–117.
[27] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: How far are we?" in *Proceedings of the International Symposium on Software Testing and Analysis*, 2019, pp. 43–54.
[28] H. M. Son, H. Ren, X. Liu, and Z. Zhao, "Automating Android build repair: Bridging the reasoning–execution gap in LLM agents with domain-specific tools," in *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics*. Rabat, Morocco: Association for Computational Linguistics, 2026, pp. 4169–4189. [Online]. Available: https://aclanthology.org/2026.eacl-long.195/
[29] Z. Yu, Y. Zhang, M. Wen, Y. Nie, W. Zhang, and M. Yang, "CXXCrafter: An LLM-based agent for automated C/C++ open source software building," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2618–2640, 2025.
[30] L. Hu, G. Chen, X. Shang, S. Cheng, B. Wu, G. Li, X. Zhu, W. Zhang, and N. Yu, "CompileAgent: Automated real-world repo-level compilation with tool-integrated LLM-based agent system," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vienna, Austria: Association for Computational Linguistics, 2025, pp. 2078–2091. [Online]. Available: https://aclanthology.org/2025.acl-long.103/
[31] N. Wei, B. Jiang, W. Wei, and M. Duan, "Agentic LLM-driven C++ build automation: An empirical study," in *Proceedings of the 2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2026, pp. 125–135. [Online]. Available: https://doi.org/10.1109/SANER67736.2026.00023
[32] "Crosstool-NG," https://github.com/crosstool-ng/crosstool-ng.
[33] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1661–1682, 2023.
[34] S. Huang and X. Wang, "Build issue resolution from the perspective of non-contributors," in *Proceedings of the International Conference on Automated Software Engineering*, 2024, pp. 2304–2308.
[35] S. Huang, N. Meng, X. Liu, and X. Wang, "An exploratory study on build issue resolution among computer science students," in *Proceedings of the IEEE/ACM International Conference on Software Engineering Education and Training*, 2025, pp. 182–192.
[36] "Configure terms and history," https://gcc.gnu.org/onlinedocs/gccint/Configure-Terms.html.
[37] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha, "B2R2: Building an efficient front-end for binary analysis," in *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.
[38] "Debian popularity contest," https://popcon.debian.org/.
[39] C. Liu, R. Saul, Y. Sun, E. Raff, M. Fuchs, T. S. Pantano, J. Holt, and K. Micinski, "ASSEMBLAGE: Automatic binary dataset construction for machine learning," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2024. [Online]. Available: https://dl.acm.org/doi/10.5555/3737916.3739787
[40] H. Tan, X. Tian, H. Qi, J. Liu, Z. Gao, S. Wang, Q. Luo, J. Li, and Y. Zhang, "Decompile-Bench: Million-scale binary-source function pairs for real-world binary decompilation," in *Advances in Neural Information Processing Systems*, vol. 38, 2025.
[41] M. Yu, Z. Zeng, W. Ye, J. Sun, X. Bai, and S. Zhang, "R^3-Bench: Reproducible real-world reverse engineering dataset for symbol recovery," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025, pp. 217–227.
[42] "Assemblage," https://github.com/Assemblage-Dataset/Assemblage.
[43] "64-bit PowerPC ELF application binary interface supplement 1.9," https://refspecs.linuxfoundation.org/ELF/ppc64/PPC-elf64abi-1.9.
[44] C. Zhao, M. Ma, S. Zhang, Z. Huang, Y. Sun, C. Bansal, S. Rajmohan, and D. Pei, "EvidenT: An evidence-preserving framework for iterative system-level package repair," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2026.