CROSS2OH: Enabling Seamless Porting of C/C++ Software Libraries to OpenHarmony

Qian Zhang[a†], Tsz-On Li[b,c†], Ying Wang[a*], Li Li[d], Shing-Chi Cheung[b,c]
[a]Northeastern University, Shenyang, China
[b]The Hong Kong University of Science and Technology, Hong Kong, China
[c]Guangzhou HKUST Fok Ying Tung Research Institute, Guangzhou, China
[d]Beihang University, Beijing, China
2371418@stu.neu.edu.cn, toli@connect.ust.hk, wangying@swc.neu.edu.cn, lilicoding@ieee.org, scc@cse.ust.hk

***Abstract*—OpenHarmony is a new mobile operating system that offers a popular alternative to Android and iOS. To support its adoption, significant efforts have been devoted to porting C/C++ libraries from Linux to OpenHarmony. However, this porting process presents unique challenges due to the fundamental architectural differences in system libraries, runtime environments, and build systems between the two platforms. These discrepancies manifest as Cross-platform Incompatibility (CPI) issues during cross-compilation, which are particularly difficult to resolve for two key reasons. First, conventional cross-compilation toolchains provide only brief error messages that offer inadequate diagnostic information for CPI issues. Second, resolving these issues requires a deep understanding of cross-platform discrepancies, yet comprehensive documentation or systematic guidelines about such Linux-to-OpenHarmony differences remain largely unavailable.**

In this *experience paper*, to assist developers in addressing these challenges, we conducted an empirical study on 92 C/C++ libraries successfully ported to OpenHarmony. Through manual step-by-step reproduction of all CPI issues, our study reveals that discrepancies between Linux and OpenHarmony can be divided into three categories, and CPI issues can manifest through eight dimensions. Furthermore, we identified eight common adaptation strategies for resolving CPI issues. Based on these findings, we present CROSS2OH, an automated technique for porting Linux-based software to OpenHarmony. Our approach combines: (1) an adaptation knowledge base (derived from RQ1 and RQ2 findings) and (2) a static analysis approach to detect and patch eight types of CPI issues. Evaluation using real developer patches shows CROSS2OH achieves 0.94 recall and 0.91 precision in resolving CPI issues. Notably, CROSS2OH enables successful cross-compilation for 40 critical libraries (including dependencies for popular Android apps such as *WeChat*, *Microsoft Excel*, *Bilibili*), with 29 of them passed official OpenHarmony review. The evaluation results demonstrate CROSS2OH's potential to streamline the porting process and foster the growth of the OpenHarmony software ecosystem.

***Index Terms*—OpenHarmony; Software Porting**

I. INTRODUCTION

OpenHarmony, the open-source core of `HarmonyOS`, originated from `Huawei`'s contributions to the `OpenAtom Foundation`. Since `HarmonyOS NEXT`'s 2024 launch, it has been deployed at nearly one billion devices, emerging as a widely used mobile operating system alternative to Android and iOS [1], [2]. However, `HarmonyOS NEXT` and Android are not fully compatible, introducing difficulties in porting software across them [2].

The OpenHarmony platform natively supports C/C++ programming languages, while Android mobile applications extensively utilize Linux-based C/C++ software libraries. To foster ecosystem growth, the community advocates for cross-platform migration of core C/C++ software supply chains from Linux to OpenHarmony. The official ***openharmonysig/tpc_c_cplusplus*** [3] repository serves as a centralized platform for hosting compatible C/C++ software assets. As of May 2025, this repository contained only 396 C/C++ libraries, prescribing a premature ecosystem.

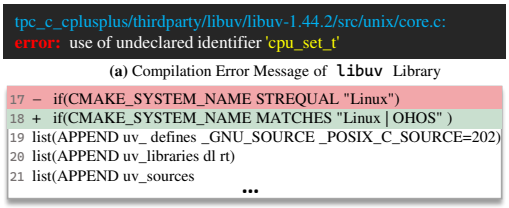
Cross-platform porting of C/C++ software is challenging. At its core, OpenHarmony employs a multi-kernel architecture: a highly tailored and optimized Linux kernel, alongside the LiteOS kernel and Hongmeng kernel [1]. There are two key differences between standard Linux and the OpenHarmony kernel: (1) ***System Libraries and Runtime Environment:*** Standard Linux relies on the *GNU C Library (glibc)* and *Linux kernel syscalls*, while OpenHarmony uses the *Musl C Library (musl)* and *customized Linux kernel syscalls*. (2) ***Build System:*** most existing C/C++ libraries are designed for *CMake/Make*-based systems, whereas OpenHarmony adopts *GN + Ninja* as its default build framework. These differences necessitate cross-compilation with specific code adaptations for OpenHarmony compatibility [4].

The OpenHarmony community provides LYCIUM [5], an official cross-compilation framework designed to simplify C/C++ library porting. Although LYCIUM integrates comprehensive toolchains (e.g., *aarch64-linux-ohos-clang*) and automates build system conversion (*CMake→GN/Ninja*), directly using LYCIUM for software porting can still introduce various Cross-platform Incompatibility (**CPI**) issues. **Due to lack of code adaptation toolchain support from OpenHarmony**, developers face two challenges during the porting process:

(1) ***Challenges in Locating and Diagnosing CPI Issues:*** Software being ported could exhibit multiple CPI issues. However, the cross-compilation toolchain only provides brief error messages describing the symptom of CPI issues, without adequate information about the root causes (e.g., discrepancies between platforms) of the issues.

†Both authors equally contribute to the paper.
*Ying Wang is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00146



(a) Compilation Error Message of `libuv` Library

(b) Adaptation Patch for Porting `libuv` Library

Fig. 1: An illustrative example of Software Porting Challenges

(2) ***Challenges in Developing CPI Adaptation Solutions:*** Due to lack of comprehensive documentation or systematic guidelines about Linux-to-OpenHarmony differences, developers could have difficulties in systematically understanding: (a) cross-compilation-impacting differences between standard Linux and OpenHarmony kernels, and (b) appropriate code adaptation approaches for resolving CPI issues.

Figure 1 illustrates an example. Using LYCIUM, we reproduced the porting process of the `libuv` library [6]. A compilation error occurred in the source file `src/unix/core.c`: `use of undeclared identifier 'cpu_set_t'`. The error log indicated the issue occurred at the statement invoking `cpu_set_t`. By analyzing the actual adaptation patches in the ***openharmony-sig/tpc_c_cplusplus*** repository, we found that `cpu_set_t` is part of GNU extensions, defined in `<sched.h>`. Its declaration depends on the `_GNU_SOURCE` macro being enabled. Without this macro definition, the type remains inaccessible even with correct header inclusion. Thus, explicitly declaring `cpu_set_t` at the error location cannot resolve the issue. The fundamental solution requires including `_GNU_SOURCE` macro definition in the OpenHarmony-specific conditional compilation branch (`Line 18`). In conclusion, compilation error messages offer inadequate guidance for both diagnosing CPI issues and deriving adaptation strategies. Moreover, the build system's first-error termination mechanism inherently limits complete CPI issue discovery during initial compilation phases.

To facilitate developers in addressing C/C++ porting challenges and promote the sustainable growth of OpenHarmony's ecosystem, we first conduct an empirical study of CPI issues, followed by the development of an automated resolution approach. In the study, we collected and studied 92 successfully ported libraries with adaptation patches from the ***openharmonysig/tpc_c_cplusplus*** repository. Through manual step-by-step reproduction of all CPI issues and analysis of compilation error logs, error contexts, adaptation patches, and documentation, we systematically investigated: (**RQ1**) CPI issue types and their root causes; (**RQ2**) Common adaptation strategies. Our study reveals that OpenHarmony and standard Linux differ in three core aspects: (1) *system libraries/runtime environments*, (2) *build systems*, and (3) *third-party library version requirements*, which can trigger CPI issues. Besides, our study shows that CPI issues manifest through eight dimensions. We further distilled eight common adaptation strategies for these CPI issue types.

Based on empirical study findings, we report our experience designing and implementing a technique CROSS2OH, which builds an adaptation knowledge base containing Linux-to-OpenHarmony incompatibility code features (**RQ1** findings) and compatible implementation alternatives (**RQ2** findings). It then combines static analysis with this knowledge base to detect eight types of platform-specific issues and automatically generate adaptation patches. By integrating these patches with the LYCIUM framework, CROSS2OH fully automates the cross-compilation process, enabling seamless cross-platform porting. We established a benchmark using authentic developer adaptation patches to validate tool effectiveness. Experimental results demonstrate that CROSS2OH achieves 0.94 recall and 0.91 precision in resolving CPI issues. Significantly, our tool enables successful cross-compilation for 40 critical software libraries - including dependencies for popular Android applications like ***WeChat***, ***Microsoft Excel***, and ***Bilibili*** - on OpenHarmony, with 29 of them passed official developers' review and testing procedures.

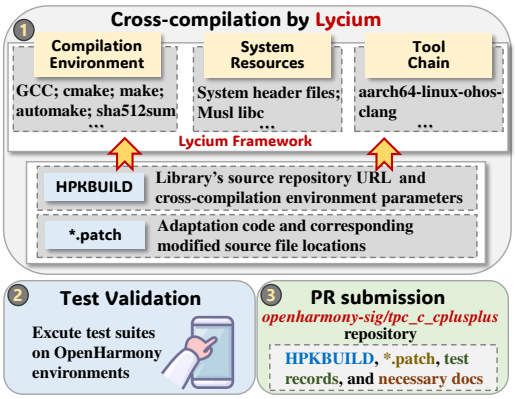


Fig. 2: Specifications for Library Porting in OpenHarmony

This paper makes the following contributions:

- ***Originality:*** First empirical study of CPI issues in OpenHarmony, including: (1) root cause analysis, and (2) adaptation strategies. Our empirical study provides actionable insights and good practices for porting C/C++ software to OpenHarmony.
- ***Reproduction package:*** Publicly released resources on website (***https://cross2oh.github.io/***): (1) Dataset (201 CPI issues and corresponding patches), (2) Benchmark (103 CPI issues and corresponding patches).
- ***Technique:*** CROSS2OH, a LYCIUM-integrated tool automating C/C++ porting with 0.94 recall and 0.91 precision. To date, CROSS2OH has enabled successful cross-compilation for 40 core mobile app libraries.

II. BACKGROUND

A. Principles and Specifications for Software Porting

The community encourages developers to port C/C++ software libraries from Linux to OpenHarmony and contribute them to the ***openharmony-sig/tpc_c_cplusplus*** repository. As shown in Figure 2, the porting process requires developers to utilize LYCIUM, the cross-compilation framework released by OpenHarmony, and comply with the following three porting specifications [7]:

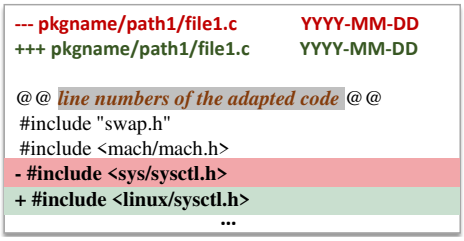


Fig. 3: A illustrative example of a `*.patch` file

- ***Specification 1:*** To facilitate tracing and maintenance of C/C++ libraries ported from Linux, **OpenHarmony community prohibits developers from making direct modifications to the libraries' source code. If developers need to adapt the libraries' source code for addressing CPI issues, the community mandates the developers to**: (1) Record adaptation code and corresponding modified source file locations in a `*.patch` file to resolve CPI issues. (2) Specify the library's source repository URL and cross-compilation environment parameters in the build file to be processed by LYCIUM. **LYCIUM cross-compiles the library based on the information in `*.patch` file and the build file, in order to generate OpenHarmony-compatible binaries.**
- ***Specification 2:*** Execute builtin test suites of the cross-compiled library binaries in OpenHarmony environments (preferably on OpenHarmony development boards), in order to validate full compliance of the ported libraries on OpenHarmony devices.
- ***Specification 3:*** The ***openharmony-sig/tpc_c_cplusplus*** repository does not accept ported source code or cross-compiled binaries of C/C++ libraries. Contributors must submit a pull request comprising cross-compilation configuration files (including a `*.patch` file and the build file for LYCIUM), test records, and necessary documentation to the repository. Upon the submission of the pull request, the configuration files will be verified by CI tools (i.e., cloud-based LYCIUM framework) running in the backend of the repository. If the cross-compilation by the CI tools passes, community maintainers will verify the test records on OpenHarmony devices and review the format of the submitted documentation. Pull requests that meet these specifications will be merged into the repository.

B. Configurations of LYCIUM Cross-Compilation Framework

The LYCIUM Framework currently supports cross-compilation with either *CMake*, *Make* or *Configure* [5]. The cross-compilation requires developers to manually create a build file for one of these three build methods, and also a configuration file called *.patch file.

`*.patch` file records the adaptation needed to be made to a library's source code, so that the library can be cross-compiled on OpenHarmony. The community recommends generating a `*.patch` file using the *Linux diff* tool as follows: diff -Naur *file1 file2* ≫ .patch . Here, *file1* refers to the original source file, while *file2* is the adapted source file. As illustrated in Figure 3, a `*.patch` file contains three critical pieces of information: (1) The name and path of the source file requiring adaptation. (2) The line numbers of the adapted code. (3) The adaptation code for resolving CPI issues.

However, the community lacks automated tools to help developers locate CPI issues or recommend adaptation strategies. Manually creating a `*.patch` file requires: (1) a thorough understanding of the differences between OpenHarmony and Linux, and (2) expert knowledge in resolving the compatibility issues caused by the differences. This remains a highly challenging task for developers.

III. EMPIRICAL STUDY

To systematically identify key differences between Linux and OpenHarmony that require adaptations, we conduct an empirical study by analyzing the `*.patch` files available in the ***openharmony-sig/tpc_c_cplusplus*** repository. In this section, we aim to answer the following two research questions:

- **RQ1 (Issue Types and Root Causes):** *What are common types of CPI issues? What are their root causes?*
- **RQ2 (Adaptation Strategies):** *How do developers resolve CPI issues in the `*.patch` files? Are there any common adaptation strategies?*

To address RQ1, we collect libraries from the ***openharmony-sig/tpc_c_cplusplus*** repository whose `*.patch` files are available in the repository. We reproduced all CPI issues of a library by iteratively removing each adaptation code segment from the `*.patch` file and then re-cross-compile the library with LYCIUM, so that compilation errors related to the CPI issues can be triggered. Then, we conduct root cause analysis by thoroughly examining compilation error logs and adaptation code.

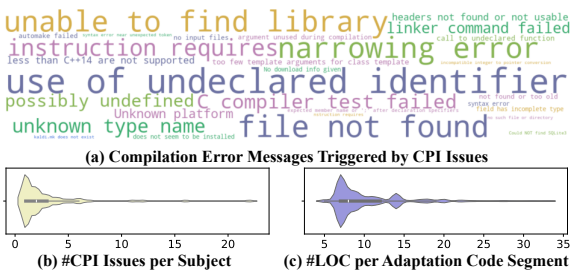
To answer RQ2, we derive adaptation strategies for CPI issues by analyzing developers' docs regarding `*.patch` files in the ***openharmony-sig/tpc_c_cplusplus*** repository.

A. Data Collection

We adopt a three-step approach to construct a dataset:

Step 1. Collecting `*.patch` files. As of May 2025, the ***openharmony-sig/tpc_c_cplusplus*** repository has archived 396 successfully ported C/C++ libraries. For our empirical study, we collected 257 libraries ported before April 2024 as the dataset $Dataset_1$, while reserving the 110 libraries ported in the most recent year (April 2024-May 2025) as the dataset $Dataset_2$ for experimental evaluation (Section V). Among the 257 collected libraries in $Dataset_1$, 92 (35.7%) contain `*.patch` files, indicating they encountered CPI issues during cross-platform porting. These constitute our primary investigation subjects.

Step 2. Reproducing compilation failures induced by CPI issues. Each `*.patch` file may contain multiple code segments. Each code segment specifies modified source file path and corresponding line numbers to address a CPI issue (Figure 3 shows an illustrative example). For each C/C++ library in $Dataset_1$, we systematically reproduced all related CPI issues using the LYCIUM 1.0 cross-compilation framework within the `ohos_sdk 4.1.3.5` environment. The reproduction process involved two key tasks: (1) For each



(a) Compilation Error Messages Triggered by CPI Issues

(b) #CPI Issues per Subject

(c) #LOC per Adaptation Code Segment

Fig. 4: Statistics of Empirical Study Dataset

`*.patch` file, we iteratively removed individual adaptation code segments and performed re-cross-compilation of the corresponding library. (2) Capturing compilation error logs and pinpointing the incompatible code context.

Step 3. Identifying adaptation strategies that resolve CPI issues. For each adaptation code segment in `*.patch` files, we collected and analyzed developers' commit logs and documents. These serve as critical evidence for identifying adaptation strategies.

Through the aforementioned data collection process, we identified 201 CPI issues. Each CPI issue can be formally represented as a quadruple ***<error, code_context, adaptation_code, doc>***, where: (1) *error*: The compilation error message triggered by the CPI issue; (2) *code_context*: The source code location (with surrounding context) exhibiting platform incompatibility as indicated by the compiler error; (3) *adaptation_code*: The modified code segment resolving the CPI issue; (4) *doc*: Developer documentation explaining the adaptation rationale. As illustrated in Figure 4, reproducing these CPI issues triggered 30 distinct compilation error types, with an average of 10.5 lines per adaptation code segment. These issues originate from widely-used, large-scale C/C++ libraries, ensuring representative analysis subjects.

B. Data Analysis

We performed an in-depth analysis of the 201 CPI issues. To address RQ1, we analyzed and categorized issue types and root causes based on each CPI issue's *error*, *code_context*, and *adaptation_code*. For RQ2, we derived adaptation strategies through examining *adaptation_code* and their accompanying documentation *doc*.

Our data analysis followed an open-coding procedure [8], a well-established qualitative research approach, to systematically categorize the issues. Three authors of this paper, each with over three years of C/C++ development experience, first independently analyzed each CPI issue with reference to OpenHarmony's technical documentation. Through four rounds of structured discussions, they then established consensus to refine the issue taxonomy and adaptation strategy classifications. This rigorous process involved aligning category definitions, resolving labeling conflicts through deliberative debates among all authors, and progressively optimizing the hierarchical taxonomy until full consensus was achieved on both issue types and adaptation strategies.

C. RQ1: Issue Types and Root Causes

The discrepancies between OpenHarmony and Linux led us to identify three common CPI issue types. These categories further decompose into eight distinct subcategories based on their root causes.

Type A. *CPI issues caused by system library and runtime environment discrepancies between OpenHarmony and Linux (108/201 = 53.7%).* Our analysis reveals that *Type A* CPI issues can be subdivided into three cases:

Type A.1. *OpenHarmony and Linux support discrepant sets of native APIs (51/201 = 25.4%).* This discrepancy stems from two main reasons: (1) the system APIs of Linux are provided by *glibc* while the system APIs of OpenHarmony are provided by *musl*. *glibc* and *musl* support discrepant sets of system APIs [9]. (2) OpenHarmony extensively customizes the Linux kernel [10], which results in partial incompatibility in standard syscall ABI (Application Binary Interface).

For example, the cross-compilation of `OpenLDAP` library on OpenHarmony fails due to an unresolved dependency *pthread_mutexattr_setrobust()*. This POSIX thread function, while implemented in Linux's *glibc*, is omitted in OpenHarmony's *musl*.

Type A.2. *glibc and musl use different include path structures (19/201 = 9.4%).* To include header files in source code, *glibc* requires using logical directories (e.g., `sys/` and `bits/`) in include statements, whereas *musl* requires directly referencing physical paths. This divergence stems from *musl* 's design goal to simplify glibc's implementation [11], [12], including path resolution and file discovery.

For example, the library `chipmunk2D` includes the system header `sysctl.h` via `#include <sys/sysctl.h>`, where `sys/sysctl.h` is a logical path used by glibc that maps to the physical path `linux/sysctl.h`. However, OpenHarmony (*musl*) requires using the physical path, so the include statement should be `#include <linux/sysctl.h>`. Using the glibc-style logical path results in compilation errors on OpenHarmony.

Type A.3. *OpenHarmony and Linux have different file system hierarchies (38/201 = 18.9%).* Linux strictly adheres to the Filesystem Hierarchy Standard (FHS), utilizing unified directories such as `/usr` and `/lib` for system resources, whereas OpenHarmony employs a customized file system hierarchy. For instance, OpenHarmony does not have the directory `/usr` by default. Alternatively, it introduces customized directories such as `/system` and `/chipset`. The discrepancy could result in compilation errors.

For example, the `pulseaudio` library hardcodes its configuration directory as `/usr/etc/pulse` in the Linux environment. However, OpenHarmony does not have the `/usr` directory by default, resulting in cross-compilation error.

Type B. *OpenHarmony and Linux have discrepant build systems (87/201 = 43.3%).* This category of CPI issues can be further divided into four cases:

Type B.1. *The compilers of Linux and OpenHarmony configure the signedness of* `char` *type differently (13/201 = 6.5%).* Specifically, `char` type on Linux is configured to

store `signed` value ranging from -128 to 127, whereas `char` type on OpenHarmony is configured to store `unsigned` value ranging from 0 to 255. This discrepancy could lead to errors in *type promotion rules*, *arithmetic operations*, and *signedness comparisons*.

For example, the compilation failure of `json-schema-validator` occurred because negative integer constants were assigned to `unsigned char` on OpenHarmony. This exceeded the valid range of the `unsigned char` data type, triggering an illegal narrowing conversion when initializing a char array.

Type B.2. *OpenHarmony and Linux differ in their compilation configurations for underlying instruction set architectures (ISA) (10/201 = 5%).* Linux primarily targets `x86_64` architecture, while OpenHarmony runs mainly on `ARM`, requiring explicit instruction set specifications to ensure binary compatibility. To cross-compile software in OpenHarmony running on an `ARM` platform, developers must specify the instruction set version using options like `-march=armv7-a` or `-march=armv8-a`, along with processor-specific extensions through parameters such as `-mcpu` (e.g., `cortex-a7`) and `-mfpu` (e.g., `neon-vfpv4`). In contrast, `x86_64` Linux compilation typically requires only generic optimization flags like `-march=x86-64`. Inappropriate configuration of these architecture-specific compilation settings may lead to compilation failures.

For example, the cross-compilation of `cpp-httplib` library from `x86_64` Linux to `ARM64` OpenHarmony failed due to missing `ARMv7-A` architecture specification (`-march=armv7-a`) in the `Makefile`'s compiler options.

Type B.3. *OpenHarmony and Linux have discrepant OS-detection macros for conditional compilation (60/201 = 29.8%).* Cross-platform C/C++ applications typically use OS-detection macros (e.g., `__linux__`, `__GLIBC__`) for conditional compilation (`#ifdef`, `#if`, etc.) of OS-specific components. However, OpenHarmony introduces its own OS-detection macros (`__OHOS__`, `__MUSL__`) to control conditional compilation on OpenHarmony. Inappropriate use of these macros could cause essential conditional branches to be incorrectly skipped during cross-compilation.

For example, the `Chrono` library employs conditional compilation (e.g., `#ifdef _WIN32`, `#elif __gnu_linux__`) to define OS-specific data types. However, the absence of OpenHarmony-specific macros causes this conditional branch to be completely skipped, resulting in undefined type errors during cross-compilation.

Type B.4. *The compiler toolchains of OpenHarmony and Linux have different requirements on C++ versions (4/201 = 2%).* OpenHarmony's `Clang++` mandates C++14 as the minimum supported version. Compilation errors will arise when the default C++ version set in a software's configuration file is lower than C++14.

For example, the `CMakeLists.txt` file in the `protobuf` library explicitly sets the C++ standard to version 11 via the `set(CMAKE_CXX_STANDARD 11)` directive. However, this configuration conflicts with the compilation toolchain's requirement that only C++14 or later versions are supported.

Type C. *API breaking changes of third-party libraries pre-installed in OpenHarmony and Linux* (4/201 = 2%). For example, the library `zbar` fails to be cross-compiled in OpenHarmony, with an unresolved reference error stating that *png_set_gray_1_2_4_to_8()* cannot be found in the `libpng` library, which is pre-installed in OpenHarmony. In fact, OpenHarmony installs a newer version of the `libpng` library, and the signature of the API changes to *png_set_**expand**_gray_1_2_4_to_8()*.

Others. Two issues do not fall into the above categories. They occurred due to discrepant warning messages thrown by the compiler toolchains of OpenHarmony and Linux. Thus, libraries that have the compilation flag "`-Werror`" enabled may fail the cross-compilation on OpenHarmony but not on Linux. Developers addressed these cases by removing the "`-Werror`" flag from the libraries' configuration files.

Answer to RQ1: Discrepancies between OpenHarmony and Linux across three core aspects - (1) ***system libraries/runtime environments***, (2) ***build systems***, and (3) ***third-party library version requirements*** - can trigger CPI issues, which typically manifest through eight dimensions.

D. RQ2: Adaptation Strategies

By studying 201 *adaptation_code* implementations in the `*.patch` files and their accompanying *doc*, we observed eight common adaptation strategies.

Strategy 1. For Native APIs not supported by OpenHarmony (***Type A.1*** CPI issues), developers can adapt them by either: (1) supplementing equivalent alternative APIs from *musl*/OpenHarmony kernel, or (2) implementing the APIs within `*.patch` files.

For example, the `HDiffPatch` library calls the system API *pthread_yield()*, which is unavailable in OpenHarmony's *musl* implementation. Developers addressed this compatibility issue by replacing it with an alternative API *sched_yield()* supported by OpenHarmony.

Strategy 2. To resolve *musl*'s discrepant include path structure from *glibc*'s (***Type A.2*** CPI issues), developers replace logical paths which reference header files with physical paths.

For example, *Chipmunk2D* library references the header file `sysctl.h` using logical path `sys/sysctl.h` in Linux environment. Developers addressed this by replacing the logical path with physical path `linux/sysctl.h`.

Strategy 3. To resolve OpenHarmony's non-compliance with Filesystem Hierarchy Standard (FHS) (***Type A.3*** CPI issues), developers employed path remapping by converting the standard FHS hierarchy to OpenHarmony-compatible filesystem hierarchy.

For example, The `cpp-httplib` library is typically installed in `/usr/local`, but `/usr` does not exist in OpenHarmony. Developers therefore remapped the path to `$LYCIUM_ROOT/usr`, which aligns with OpenHarmony's filesystem hierarchy.

Strategy 4. To resolve the inconsistency in default signedness of the `Char` type between OpenHarmony and Linux (***Type B.1*** CPI issues), developers explicitly enforced the signedness of `Char` type.

For example, `json-schema-validator` library's source code initializes `char[]` arrays using negative integers, which triggers type conversion error on OpenHarmony. To resolve this, developers explicitly enforced `Char` signedness by adding the `-fsigned-char` compilation option in `CMakeLists.txt`

Strategy 5. To resolve instruction set architecture (ISA) compilation incompatibilities between OpenHarmony and Linux (***Type B.2*** CPI issues), developers explicitly incorporated ARM-specific compilation options in build configurations to ensure proper architectural support.

For example, the `cpp-httplib` library's ISA incompatibilities were resolved through inclusion of `ARMv7-A`-specific compilation options (including `-march=armv7-a` and `-mfpu=neon`) in the `Makefile` configuration.

Strategy 6. To address unsupported conditional compilation macros for OpenHarmony (***Type B.3*** CPI issues), developers extended the original OS-detection logic by adding: (1) OpenHarmony-specific macro checks (e.g., `__OHOS__`); (2) Build system identification (e.g., `OHOS` in `CMake`). This ensures proper OS-specific dependency inclusion and proper variable definition.

For example, the `curl` library fails to cross-compile on OpenHarmony due to incomplete OS-detection in its *CMake* configuration. This adaptation requires patching the build configuration to include a condition check for OpenHarmony (`CMAKE_SYSTEM_NAME STREQUAL "OHOS"`) along with corresponding OpenHarmony-specific settings.

Strategy 7. To address C/C++ version incompatibilities between a library and OpenHarmony's compilation toolchain (***Type B.4*** CPI issues), developers explicitly specify OpenHarmony-supported version (e.g., C++14/17) in the library's build configuration files.

For example, the `MXNet` library originally supported C++11. To ensure successful cross-compilation on OpenHarmony, developers modified its *CMakeLists.txt* configuration to enforce C++17 compliance via set(`CMAKE_CXX_STANDARD 17`).

Strategy 8. To address API breaking changes of third-party libraries on Linux and OpenHarmony (***Type C*** CPI issues), developers ensure compatibility by either: (1) replacing incompatible APIs with their OpenHarmony equivalents, or (2) implementing missing API implementation in *.patch files.

For example, for the `zbar` case described in §III-C, developers replace the original API (i.e., *png_set_gray_1_2_4_to_8()*) with the OpenHarmony-compatible one (i.e., *png_set_**expand**_gray_1_2_4_to_8()*).

Answer to RQ2: Our analysis revealed eight adaptation strategies addressing all three CPI issue types (including subtypes) from RQ1. Successful adaptation demands developers' thorough comprehension of OpenHarmony-Linux divergences along with their respective workarounds.

IV. Cross2OH Approach

Our empirical study reveals key platform divergences between OpenHarmony and Linux that hinder successful cross-compilation of C/C++ software using Lycium. To address these challenges, we developed Cross2OH, an automated tool for: (1) diagnosing CPI issues, and (2) generating code adaptation patches. Figure 5 shows an overview of Cross2OH. It constructs a comprehensive adaptation knowledge base containing:

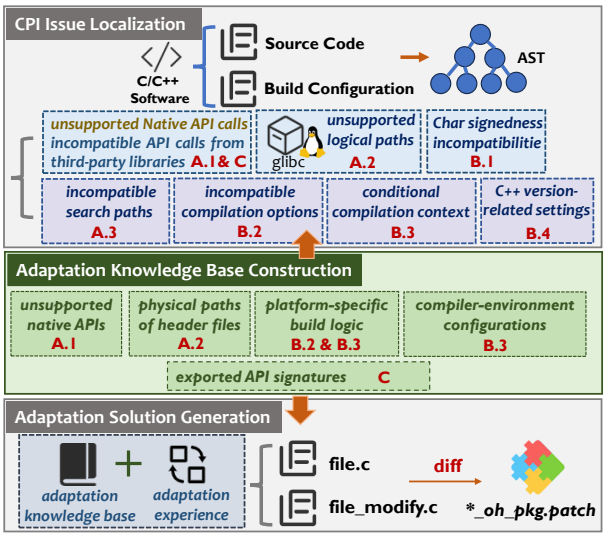


Fig. 5: An overall Architecture of Cross2OH.

- Linux-specific code features incompatible with OpenHarmony (derived from RQ1 root cause analysis);
- OpenHarmony-compatible implementation alternatives (based on RQ2 adaptation strategies).

Cross2OH integrates static analysis with its adaptation knowledge base to systematically identify various types of OpenHarmony incompatibilities: (1) problematic API calls, (2) invalid logical paths, (3) incompatible build tool/library/header filesystem hierarchy, (4) compiler-specific type signedness defaults (particularly for `char`), (5) ISA-dependent compilation configurations, (6) OS-specific conditional compilation macros, and (7) C++ version mismatches. Based on the adaptation knowledge base, Cross2OH generates adaptation patches for each identified issue type.

A. Adaptation Knowledge Base Construction

Based on the RQ1 findings, our knowledge base systematically documents code features causing cross-OS compilation failures and their corresponding OpenHarmony-compatible implementation alternatives across three aspects: *system libraries/runtime environments*, *build system* and *third-party library version requirements*. The knowledge base underpins Cross2OH's CPI issue diagnosis and adaptation.

Part 1: Incompatible Code Features in System Libraries/Runtime Environments. Cross2OH collects: (1) *Linux Native APIs* unsupported by OpenHarmony, and (2) *Physical Path Mapping* of all header files supported by OpenHarmony, resolving ***Type A.1*** and ***Type A.2*** CPI issues.

- ***Unsupported Native APIs:*** The community's official doc [13] provides 305 Native APIs that are unsupported by OpenHarmony, which form the basis of our knowledge base. These unsupported Native APIs fall into several groups: *Environment Variables* (e.g., *_environ*), *Thread Control* (e.g.,

TABLE I: Statistics of the Adaptation Knowledge Base

Part 1	*Unsupported Native APIs*	305
	Physical Paths of Header Files	1,599
Part 2	*OS-Specific Build Logic*	9
	Compiler-environment Configurations	15
Part 3	*API Signatures of ported libraries*	185,915

pthread_cancel), *Filesystem Extensions* (e.g., *__xmknod*), *Time Handling* (e.g., *__adjtime64*), etc.

For the Linux Native APIs unsupported by OpenHarmony, **CROSS2OH gathers their equivalents** through two approaches: (1) Extracting validated substitutions from successfully code adaptation patches in official repositories, identifying community-approved mappings between unsupported Linux Native APIs and their alternatives in *musl* and the OpenHarmony kernel. (2) For APIs lacking alternatives for substitution, CROSS2OH directly collects their code implementations from *glibc* and the Linux kernel, along with critical context such as: API signatures (parameters, return types), and required header files and macros.

- ***Physical Paths of Header Files:*** Unlike *glibc*'s logical path inclusion mechanism, *musl* creates physical path mapping for header files, directly referencing their actual filesystem locations. This causes ***Type A.2*** CPI issues. To enable accurate issue diagnosis, the tool systematically catalogs all OpenHarmony SDK header files with their physical paths, in order to adapt logical paths in header files.

Part 2: Incompatible Code Features in Build Systems. In C/C++ development, conditional compilation (e.g., #if, #ifdef, build-file conditionals) enables cross-platform support, but OpenHarmony's distinctive OS-detection macro could result in ***Type B.3*** CPI issues. To enable effective diagnosis, CROSS2OH collects platform-dependent code features, including OS conditionals (e.g., `if(UNIX)`), OS-specific macros (e.g., `add_definitions`), compiler detection (e.g., `CMAKE_CXX_COMPILER_ID`).

Part 3: API Signatures from OpenHarmony's Ported Libraries. To resolve CPI issues of *Type C* (API breaking change), CROSS2OH constructs a knowledge base of *API Signatures* from libraries successfully ported to OpenHarmony.

Table I presents the statistics of the adaptation knowledge base (#*code features*).

B. CPI Issue Localization

Based on the Linux-specific code features documented in the adaptation knowledge base—which may be incompatible with OpenHarmony—CROSS2OH statically analyzes both source code and build configurations leveraging TREE-SITTER tool [14] to identify CPI issues.

B.1. C/C++ Source Code Analysis

CROSS2OH scans all `*.c`, `*.cpp`, and `*.h` files to identify code that potentially trigger ***Types A.1, A.2, B.1*** and ***C*** CPI issues. Specifically, CROSS2OH performs **three tasks**:

Task 1: Identifying API calls unsupported by OpenHarmony. CROSS2OH locates API call information in the abstract syntax tree (AST) of source code, including API signatures (*identifier* nodes in AST) and argument lists (*argument_list* nodes in AST). Each of the collected API is then further analyzed as follows:

- *Identifying unsupported Native API calls (**Type A.1** CPI issues).* CROSS2OH performs cross-comparison between the collected API and OpenHarmony-incompatible native APIs recorded in our adaptation knowledge base.
- *Identifying incompatible API from third-party libraries (**Type C** CPI issues).* CROSS2OH cross-compares the collected APIs and APIs from third-party libraries already ported to OpenHarmony (as recorded in our adaptation knowledge base). If the collected API does not match any API from those third-party libraries, CROSS2OH considers an API breaking change potentially occurs.

Task 2: Identifying logical paths unsupported by OpenHarmony (Type A.2 CPI issues). CROSS2OH uses two steps to identify incompatibility issues caused by using logical paths. First, it extracts relative paths from header files via regex matching. Second, it checks whether these paths contain *glibc*-specific logical path patterns (e.g., `sys/`, `gnu/`, or `bits/`).

Task 3: Identifying `Char` signedness incompatibilities with OpenHarmony's ABI (Type B.1 CPI issues). CROSS2OH employs static analysis to identify potentially affected code segments through three steps. First, CROSS2OH extracts extracts **AST nodes** containing *assignment expressions* (`assignment_expression`), *binary operations* (`binary_expression`), and *Initializer declarations* (`init_declarator`). These expressions are plausible infection locations of the CPI issue. Second, CROSS2OH statically locates the definitions of `Char` variables which are used by above expressions. Finally, CROSS2OH checks whether these variables have been assigned negative values (e.g., `Char c = -1;` or `Char arr[] = {1, -1};`).

B.2. Build Configuration Analysis

CROSS2OH statically analyzes build configuration files (e.g., `CMakeLists.txt`, `Makefile`) to detect code that potentially cause ***Types A.3, B.2, B.3***, and ***B.4*** CPI issues. Specifically, our tool performs **four tasks**:

Task 1: *Extracting filesystem hierarchy related to build tools, shared library files, and header files.* To resolve ***Type A.3*** CPI issues arising from discrepant filesystem hierarchies between Linux and OpenHarmony, our tool statically checks whether paths in build tools, shared library files, and header files comprise Linux-specific directory hierarchies.

Task 2: *Extracting compilation options.* To identify ***Type B.2*** CPI issues, CROSS2OH statically extract ISA-related compilation options (e.g., -mcpu) declared in build files.

Task 3: *Extracting code blocks controlled by OS-detection macros.* To identify ***Type B.3*** CPI issues, CROSS2OH extracts code blocks within a conditional compilation branch associated with OS-detection macro. These code blocks are potentially OS-specific.

Task 4: *Extracting C++ version.* To resolve ***Type B.4*** CPI issues, CROSS2OH identifies the C/C++ version in configuration files to check whether it complies with OpenHarmony's requirement.

C. Adaptation Solution Generation

For code segments identified as potential compatibility issues in the CPI localization phase, CROSS2OH utilizes its

adaptation knowledge base to generate adaptation patches.

Solution 1: Adaptation of unsupported API calls (Types A.1 and C). For unsupported native API calls, CROSS2OH first queries its adaptation knowledge base for community-validated OpenHarmony equivalents from earlier adaptation patches. If no direct equivalents exist, it finds pre-collected code implementations from *glibc* and Linux kernel sources documented in the knowledge base. The tool then inserts the found implementation at the original call sites.

Solution 2: Adaptation of unsupported logical paths (Type A.2). Our knowledge base maintains a mapping between header files and their corresponding physical paths in OpenHarmony. CROSS2OH uses this to transform *glibc*-style logical paths into *musl*-compatible paths.

Solution 3: Adaptation of incompatible filesystem hierarchy (Type A.3). CROSS2OH detects incompatible filesystem hierarchy for build tools, shared libraries, and header files, then adapts them to be compatible with OpenHarmony's filesystem hierarchy. Specifically: (1) Shared libraries and headers are relocated to developer-local paths under `lycium/usr/`; (2) Build tools are redirected to `ohos_sdk/linux/native/llvm/bin/`.

Solution 4: Adaptation of identified `Char` signedness incompatibility issues (Type B.1). For libraries which have char signedness incompatibility issues, CROSS2OH adds the `-fsigned-char` compilation option in build configuration files to enforce `Signed Char` type interpretation.

Solution 5: Adaptation of incompatible ISA-specific compilation configurations (Type B.2). For ISA-specific compilation configurations, CROSS2OH verifies the inclusion of target architecture flags ("`armeabi-v7a`" and "`arm64-v8a`"). When absent, it injects architecture-specific *cflags* in conditional compilation predicate (e.g., `if(CMAKE_SYSTEM_PROCESSOR STREQUAL "aarch64")`) to enable platform-aware builds.

Solution 6: Adaptation of incompatible conditional compilation (Type B.3). For code blocks governed by conditional compilation, CROSS2OH analyzes whether they include OpenHarmony-specific macros (e.g., `__OHOS__`). When absent, our tool extends the original OS detection logic by: (1) injecting OpenHarmony-specific macro checks, or (2) adding system identification clauses ensuring correct compilation and execution on OpenHarmony.

Solution 7: Adaptation of incompatible C++ version-related settings (Type B.4). CROSS2OH verifies whether a library's configured C++ version is lower than C++14 (i.e., the minimum C++ version required by OpenHarmony's `Clang++`). If so, it modifies the library's build configurations with `set(CMAKE_CXX_STANDARD 14/17)` directives.

Based on these adaptation solutions, CROSS2OH generates a `*.patch` file for adapting the library's code.

V. EVALUATION

To evaluate CROSS2OH 's effectiveness and usefulness, we study the following two research questions.

RQ3: (Effectiveness of Resolving CPI issues): Can CPI issues be effectively resolved?

RQ4: (Usefulness of CROSS2OH): Can CROSS2OH reliably port real-world C/C++ libraries to OpenHarmony?

A. Dataset Collection

To answer RQ3, we collected 110 C/C++ libraries ported to OpenHarmony between April 2024 and May 2025—after our empirical study cutoff (March 2024)—to prevent overfitting. Among these, 46 libraries contain `*.patch` files providing 103 verified patches for CPI issues. As Table II shows, our manual analysis confirms these developer-created patches comprehensively cover all eight CPI issue types, forming a robust benchmark with diverse real-world adaptations for evaluating our tool's effectiveness.

To answer RQ4, we collected C/C++ libraries used by highly downloaded Android apps but not yet ported to OpenHarmony. After porting them using CROSSOH, we submitted PRs to the ***openharmony-sig/tpc_c_cplusplus*** repository following community specifications. We evaluated our tool's usefulness based on developer review feedback. Specifically, we crawled 207 top-downloaded commercial apps from `Google Play Store` [15] mirror (as of Jan 30, 2025) and 477 popular open-source apps from the `F-Droid` [16] repository that were updated within the past year. From these 682 apps, we extracted 10,936 shared library files (`*.so`) via decompression. Using BINARYAI [17]—a SOTA C/C++ binary analysis tool—we traced these binaries back to their source repositories, **identifying potential candidates for porting** after excluding failed traces, already-ported OpenHarmony libraries, and duplicate results, ultimately yielding 720 unique C/C++ source code repositories.

From the 720 C/C++ source repositories, we retained only 454 software projects that used build systems supported by LYCIUM (*CMake*, *Make*, or *Configure*). For the RQ4 experiment, Cross2OH's porting targets needed to be verifiably compilable on Linux. We thus manually compiled all 454 projects on Linux, and 48 of them can be successfully compiled without excessive effort in resolving compilation failures. As these 48 projects are readily compilable, we selected them as RQ4's porting targets.

B. RQ3: Effectiveness of Resolving CPI Issues

*1) **Methodology**:* We apply CROSS2OH to each of the 46 C/C++ libraries which has a ground-truth `*.patch` file. CROSS2OH generates a `*.patch` file for each library. Using the *Data Analysis* method proposed in our empirical study (§III), we identify the CPI issues and the corresponding patches in the generated `*.patch` file. We also perform this step for the ground-truth `*.patch` file, in order to construct the ground-truths for verifying the generated patches. Then, we match each generated patch to a ground-truth patch based on the CPI issue they address. Upon successful matching, we replace the ground-truth patch in the ground-truth `*.patch` file with the corresponding generated patch. We validate each generated patch individually by replacing only one ground-truth patch in the ground-truth `*.patch` file each time. With the resultant ground-truth `*.patch` file, we cross-compile the library on `HH-SCDAYU200` microprocessor development board, which has OpenHarmony 3.2 installed. Upon successful cross-compilation, we verify the correctness of the cross-compiled library, by executing the built-in test script in its source code.

Upon successful testing, we consider the generated patch successful in fixing the CPI issue.

*2) **Metrics**:* We compute *Recall* and *Precision*. We first define **True Positive (*TP*)**, **False Positive (*FP*)**, and **False Negative (*FN*)** for computing these metrics. A patch in a generated `*.patch` file is considered a ***TP*** if it successfully fixes a CPI issue. A patch in a generated `*.patch` file is considered an ***FP*** if it attempts to fix a CPI issue that does not exist, or provides an incorrect fix to a CPI issue. A CPI issue in a ground-truth `*.patch` file is considered an ***FN*** if it is not fixed by any patch in the generated `*.patch` file.

*3) **Result**:* Table II shows that CROSS2OH attains high *Recall* (0.94) and *Precision* (0.91). The table further shows that CROSS2OH attains high (0.80-1.00) *Recall* and *Precision* for each types of CPI issues. This result indicates that our empirical study successfully captures essential patterns for CPI issues identification and adaptation, facilitating the derivation of an automated and effective approach for porting C/C++ libraries to OpenHarmony.

We analyze cases which an *FP* or *FN* occurs. Essentially, we found that these cases are caused by features (e.g., APIs or compilation configurations) which no OpenHarmony-compatible alternative is available, or cannot be trivially derived from existing features. For instance, the native API *pthread_cancel* is not supported by OpenHarmony, as stated in OpenHarmony's documentation [18]. In addition, an OpenHarmony-compatible alternative cannot be trivially derived for two reasons. First, there is no equivalent API provided by OpenHarmony. Second, the original implementation of *pthread_cancel* is incompatible with OpenHarmony's system libraries or runtime environment.

In this case or alike, CROSS2OH could be unable to find a solution for adaptation, resulting in *FN*. Occasionally, CROSS2OH may replace an OpenHarmony-incompatible feature with an OpenHarmony-compatible but not entirely equivalent one, resulting in *FP*. These kinds of CPI issues may require developers to manually adapt the library source code, in order to resolve the incompatibility. Fortunately, our evaluation result shows that these cases are rare, so that CROSS2OH has a high probability of enabling automatic and seamless library porting. Besides, given the extraordinary code completion capability of recent large language models [19], [20], CROSS2OH or developers may leverage large language models to automatically provide alternatives for OpenHarmony-incompatible features. We leave the integration of CROSS2OH and large language models to future work.

C. RQ4: Usefulness of CROSS2OH

*1) **Methodology**:* We apply CROSS2OH to each of the 48 C/C++ libraries that have not been ported to OpenHarmony. CROSS2OH generates a `*.patch` file for each library. Then, we validate the generated `*.patch` file by checking whether it enables a successful cross-compilation of the library on OpenHarmony. Specifically, we carry out the cross-compilation on `HH-SCDAYU200` microprocessor development board, which has OpenHarmony 3.2 installed. Upon successful cross-compilation, we further verify the correctness of the cross-compiled library, by executing its built-in test script. Upon successful testing, we construct a pull request following the community specifications of ***openharmony-sig/tpc_c_cplusplus*** (discussed in §II-A), which require a pull request to comprise a `*.patch` file and a link to the library's repository. The porting of a library is considered successful if the pull request is accepted by OpenHarmony's developers and merged into ***openharmony-sig/tpc_c_cplusplus***.

*2) **Result**:* Out of the 48 libraries analyzed, all (100%) of these libraries can be successfully cross-compiled on OpenHarmony with the `*.patch` files generated by CROSS2OH. 40 of these 48 libraries successfully pass all builtin test cases on OpenHarmony after cross-compilation. For the remaining 8 libraries that cannot pass the built-in test cases, the main reason is that their built-in test scripts are not executable on OpenHarmony. Specifically, 4 libraries implement their test scripts in Python, which are not yet supported by OpenHarmony. 3 libraries' test scripts depend on other third-party libraries not yet ported to OpenHarmony. 1 library's test script needs to be compiled with a compiler which cannot be executed on OpenHarmony. Excluding the libraries which their test scripts are not executable on OpenHarmony, CROSS2OH achieves **100% success rate** in enabling libraries used by popular Android Apps to be successfully cross-compiled and tested on OpenHarmony.

For the 40 libraries successfully cross-compiled and tested, we submit a pull request for each of them. Eventually, **29** pull requests have been merged to OpenHarmony's repository, and 11 pull requests are still being reviewed by OpenHarmony developers. Hence, CROSS2OH's success rate in porting the libraries is 29/40 = 72.5%. Meanwhile, we studied pull requests submitted by other developers to OpenHarmony's repository. Specifically, we focused on pull requests that comprise a `*.patch` file prepared by other developers for porting a C/C++ library. Among all those pull requests that are relevant and have been validated by OpenHarmony's developers (each PR will be annotated with the result of validation check after the validation is completed, either "successful" or "failed"), we found that only 35.7% of these pull requests successfully pass the validation checks (i.e., the annotation shows "successful").

One possible reason is that OpenHarmony has a strict validation process for pull requests for library porting, as pointed out in the community specification [21] that this kind of pull requests needs to be validated by multiple steps of automatic and manual checks by OpenHarmony's developers. Hence, CROSS2OH's high success rate indicates that CROSS2OH can generate high-quality `*.patch` file that complies with the strict community specifications.

Table III shows the statistics of the 40 libraries merged to OpenHarmony's repository or under review. The table shows that these libraries are used by **popular Android Apps**, such as ***Microsoft Excel***, ***Duolingo***, ***Agoda***, ***Amazon Shopping***, ***WeChat***, which have millions to billions of downloads, indicating these libraries are extremely important to OpenHarmony's ecosystem. **This result shows that CROSS2OH enables a reliable porting of important real-world C/C++ libraries, having the potential of playing a critical role in expanding OpenHarmomy's ecosystem and accelerating the real-**

TABLE II: CROSS2OH's effectiveness in resolving CPI issues

Issue Type	*#Libraries*	*#Patches*	*TP*	*FP*	*FN*	*Recall*	*Precision*
Type A.1 *OpenHarmony and Linux support discrepant sets of native APIs*	8	24	22	4	2	91.7%	84.6%
Type A.2 *glibc and musl use different include path structures*	2	5	5	0	0	100.0%	100.0%
Type A.3 *OpenHarmony and Linux have different file system hierarchies*	5	14	14	2	0	100.0%	87.5%
Type B.1 *The compilers of Linux and OpenHarmony configure the signedness of char type differently*	3	3	3	0	0	100.0%	100.0%
Type B.2 *OpenHarmony and Linux differ in their compilation configurations for underlying ISA*	5	5	4	0	1	80.0%	100.0%
Type B.3 *OpenHarmony and Linux have discrepant OS-detection macros for conditional compilation*	16	43	40	4	3	93.0%	91.0%
Type B.4 *The compiler toolchains of OpenHarmony and Linux have different requirements on C++ versions*	6	6	6	0	0	100.0%	100.0%
Type C *API breaking changes of third-party libraries pre-installed in OpenHarmony and Linux*	1	3	3	0	0	100.0%	100.0%
Total	**46**	**103**	**97**	**10**	**6**	**94.2%**	**90.7%**

TABLE III: Statistics of the **40** Real C/C++ Libraries from Popular Android Apps that are successfully cross-compiled by CROSS2OH.

Library Name	*Category*	*#Stars*	*#KLOC*	*Android App*	*App Download*
draco	Multimedia	6,503	619.7	***Trip.com***	10M-50M
mbedtls-polarssl	Networking	5,480	71.5	***BeautyCam***	50M-100M
libiec61850	Networking	873	170.4	***WeChat***	100M-500M
zopfli	File Handling	3,427	17.2	***StarMaker***	100M-500M
libucl	File Handling	1,625	37.8	***VK***	100M-500M
bdwgc	File Handling	2,978	85.5	***HKDisneyland***	1M-5M
cppcodec	Data Handling	614	4.1	***Microsoft Excel***	1B-5B
bullet3	Multimedia	12,577	1,277.6	***Godot Editor***	500K-1M
Log4z	Utilities	334	6.5	***Mi Fitness***	10M-50M
meshoptimizer	Multimedia	5,621	45.8	***Agoda***	50M-100M
World	Multimedia	1,182	13.2	***WeSing***	100M-500M
yyjson	Data Handling	3,105	92.8	***VooV Meeting***	1M-5M
kcp	Networking	15,342	4.0	***iQIYI***	50M-100M
ogg	Multimedia	347	17.8	***Microsoft Edge***	100M-500M
reactphysics3d	Multimedia	1,529	304.4	***Lazada***	500M-1B
utf8proc	Data Handling	1,046	23.9	***CamScanner***	100M-500M
massdns	Networking	3,339	150.2	***NextDNS Manager***	100K-500K
webui	Utilities	3,559	145.9	***Mixer Controller***	–
enet	Networking	2,911	10.4	***Rethink***	500K-1M
Collections-C	Data Handling	2,908	27.3	***EteSync***	10K-50K
tinyglif	Multimedia	2,022	440.5	***Tango***	100M-500M
json11	Data Handling	2,552	1.5	***Trip.com***	10M-50M
libsamplerate	Multimedia	613	380.4	***HKDisneyland***	1M-5M
spirv-reflect-sdk	Utilities	687	158.7	***WeSing***	100M-500M
SQLiteCpp	Data Handling	2,216	382.4	***HKDisneyland***	1M-5M
capstone	Security	7,566	1,123.6	***bilibili***	5M-10M
yaml-cpp	File Handling	5,122	181.6	***bilibili***	5M-10M
libmill	Utilities	3,056	20.0	***BleOta***	100-500
re2	Utilities	8,945	47.5	***DeepL Translate***	10M-50M
squirrel	Other	915	26.0	***TIC-80***	–
libdivsufsort	Scientific	363	4.8	***Shopee***	5M-10M
cpu_features	Utilities	2,451	7.4	***Intune***	50M-100M
g3log	Utilities	908	12.5	***Amazon Shopping***	500M-1B
enchive	File Handling	631	4.0	***Duolingo***	500M-1B
lizard	File Handling	654	25.5	***bilibili***	5M-10M
u8g2	Multimedia	5,618	680.3	***Remote Video Camera***	–
n2n	Networking	6,545	134.2	***Mullvad VPN***	1M-5M
zap	Networking	2,875	73.7	***PlainApp***	1M-5M
sc	Data Handling	2,478	23.8	***EngineDataLogger***	–
oniguruma	Utilities	2,446	125.2	***Meitu***	100M-500M

*Green color indicates the PR of the library has been merged to OpenHarmony's repository. Orange color indicates the PR of the library is under review.
*To reduce the tables' size, a couple of Android Apps' name are presented in abbreviated form: "HKDisneyland" represents "Hong Kong Disneyland" and "Intune" represents "Intune Company Portal". * "–" in the *App Download* column means that the number of downloads is not available.

world adoption of OpenHarmony.

VI. DISCUSSIONS

A. Threats to Validity

Generalizability of Findings. One possible threat is the generality of CPI issues we identified in our empirical study, because these issues were observed by studying only 92 C/C++ libraries ported to OpenHarmony. To mitigate this threat, we used a different set of libraries to evaluate CROSS2OH (RO3 and RQ4), and found that CROSS2OH achieves high recall (0.94) and precision (0.91). This result suggests that our findings on CPI issues are generalizable.

Subjectivity of Inspection. We adopted manual analysis to construct the taxonomy of CPI issues, which may pose threats to the validity of empirical findings. To minimize this threat, we adopted an open coding procedure [8]. Each issue report was inspected and labeled by three co-authors independently. Any discrepancy was discussed and resolved by all authors, until a consensus was reached.

B. Lessons Learned

In this section, we discuss our experience and lessons learned while developing and applying CROSS2OH.

1) ***Identifying build configurations and dependencies before software porting can avoid build failures caused by missing environmental dependencies****:* Although LYCIUM simplifies cross-compilation by abstracting environmental differences, it struggles with the unique build systems and dependencies of third-party libraries. Each library may use different tools (e.g., *CMake*, *Make*, *Autotools*) and have distinct runtime and build-time dependencies, which cannot be handled by LYCIUM's default settings.

To overcome this, CROSS2OH automates the identification of build methods and dependencies by:

- ***Identifying build methods.*** Identify build methods by analyzing entry configuration files specific to each build system (e.g., *CMakeLists.txt* for *CMake*, configure for *GNU Autotools*, *Makefile* for *Make*).
- ***Parsing dependency declarations.*** Parse build configuration files using TREE-SITTER [14] to extract dependency declarations—such as *find_package()* in *CMake*, *AC_CHECK_LIB* in *Autotools*, and *LDLIBS* in *Make*—enables automated identification of required libraries.
- ***Detecting missing dependencies.*** Monitor build output logs and match keywords to capture missing dependency errors (e.g., `cmake: command not found`, `autoconf: command not found`), identify missing build-time tools, and automatically install them locally.

CROSS2OH integrates these build configurations into the `HPKBUILD` file, enabling LYCIUM to accurately resolve dependencies and execute build commands tailored to each build system, ensuring correct compilation of third-party libraries.

2) ***Automated analysis and completion of installation commands can reduce adaptation issues caused by non-standard build files****:* During software porting, the `install` command in build files is used to copy generated target files—such as *executables*, *dynamic/static libraries*, and *headers*—to a specified installation directory after the build completes. However, software build files may contain non-standard implementations, as illustrated in Figure 6(a), including:

- Missing `install` commands, preventing target files from being placed in the designated directory.
- Use of hard-coded paths or custom variables unsupported by LYCIUM, causing conflicts with the default configuration of

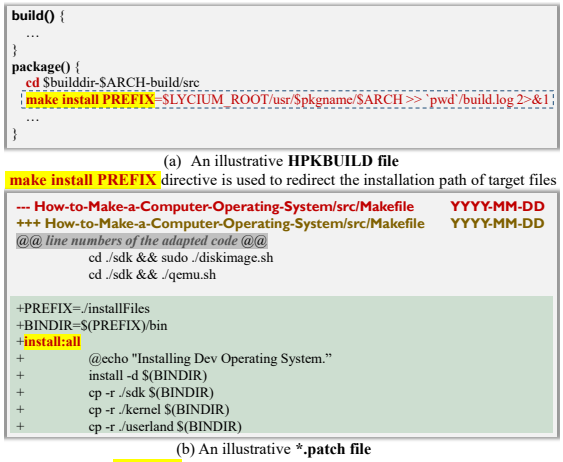


Fig. 6: An illustrative example of non-standard build files

LYCIUM's cross-compilation template.

Although these issues do not directly affect cross-compilation, they prevent target files from being correctly installed under the required directory (e.g., lycium/usr/), making them unusable in the OpenHarmony environment. To address these problems, CROSS2OH can:

- Parse the `install:all` directive (see Figure 6(b)) to locate target files generated after compilation.
- Analyze the context for installation path definitions; if hard-coded paths or custom variables are detected, it supplements the definitions and corrects path redirection variables (e.g., `make install PREFIX`) in the `HPKBUILD` file.
- Add appropriate operations to the `install` commands.

*3) **Standardizing integer formatting helps prevent potential runtime errors**:* During software porting, differences in compilation environments can generate warnings that may impact the build or runtime behavior:

- When -Werror compilation configuration is enabled, these warnings are treated as errors and interrupt compilation.
- Even without -Werror, platform-specific undefined behaviors—such as using %ld to print an int64_t, potentially causing unexpected program behavior.

For example, a type mismatch warning occurs in 64-bit environments when using %llx to format a uint64_t, since the underlying type of uint64_t varies across platforms. The OpenHarmony offical porting guideline recommends using the macros from `inttypes.h`—such as PRIu64 and PRIx64—for formatting fixed-width 64-bit integers. During preprocessing, these macros automatically resolve to the correct format specifier—such as %llx or %lx—based on the target platform, ensuring consistent and reliable output across both 32-bit and 64-bit environments.

VII. RELATED WORKS

Porting software across operating systems. Zhou et al. [2] port popular Typescript/Javascript libraries from Linux to OpenHarmony, by leveraging LLM to translate Typescript/Javascript code to ArkTS code, a programming language optimized on OpenHarmony [22]. Compared to Zhou et al.'s work, CROSS2OH focuses on addressing CPI issues during porting C/C++ libraries to OpenHarmony, instead of direct translation between programming languages.

To the best of our knowledge, Zhou et al.'s work [2] is the only related work that focuses on automatic libraries porting from Linux to OpenHarmony. Other related works [23]–[29] focus on software porting between other OSs (e.g., IOS and Android). Moreover, these works focus on assisting developers to manually adapt applications for porting, such as generating documentation (e.g., UI skeleton, guidelines) for developers, rather than automatically porting the libraries.

Lamhaddab et al. [23] presents an approach that utilizes static analysis to extract design information from iOS apps, such as user interfaces and call graphs between functions, in order to generate Android UI skeleton that aids developers to manually adapt the apps for porting. Madadipouya et al. [24] provides guidelines for developers to manually redesign an application for porting. Damset al. [25] conduct a survey about challenges and solutions faced by developers in testing during the porting process. Samet et al. [26] discuss the authors' experiences in porting NewsStand, a Web-based application designed by the authors, to IOS. These works are orthogonal to CROSS2OH, which focuses on automatic libraries porting from Linux to OpenHarmony.

Porting software across hardware architectures. Another category of works related to software porting explores the porting of operating systems across different hardware architectures [30]–[36]. For instance, Hu et al. [30] explores the feasibility of using program synthesis techniques to automate the porting of OSs across different hardware architectures. Cho et al. [33] examine the challenges of porting embedded operating systems. Zhang et al. [32] discuss their experiences in porting OSs from AUTOSAR (a software architecture in the automotive industry) to a Raspberry Pi platform. Zhadchenko st al. [31] discuss the adaptation of the X Window System for a real-time operating system called Baget.

These works essentially target issues induced by discrepancies in hardware (e.g., differences in computational resources, availability of devices). As CROSS2OH focuses on porting software across operating systems instead of hardware architectures, these works are orthogonal to CROSS2OH.

VIII. CONCLUSION

In this paper, we conducted an empirical study on CPI issues occur during the porting of C/C++ libraries from Linux to OpenHarmony. We found that discrepancies between OpenHarmony and Linux can be divided into three categories, and CPI issues can manifest through eight dimensions. Based on our findings, we developed a novel technique CROSS2OH to automatically address these issues. Our evaluation shows that CROSS2OH achieves 0.94 recall and 0.91 precision in addressing CPI issues.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFF0908000) and the China Postdoctoral Science Foundation (Grant No. 2024M750375).

REFERENCES

[1] L. Li, X. Gao, H. Sun, C. Hu, X. Sun, H. Wang, H. Cai, T. Su, X. Luo, T. Bissyandé, J. Klein, J. Grundy, T. Xie, H. Chen, and H. Wang, "Software engineering for OpenHarmony: A research roadmap," *ACM Computing Surveys (CSUR)*, Feb. 2025.

[2] B. Zhou, J. Shi, Y. Wang, L. Li, L. T. On, H. Yu, and Z. Zhu, "Porting software libraries to OpenHarmony: Transitioning from TypeScript or JavaScript to ArkTS," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025)*, 2025.

[3] "openharmonysig/tpc_c_cplusplus repository." [Online]. Available: https://gitcode.com/openharmony-sig/tpc_c_cplusplus

[4] "Openharmony docs." [Online]. Available: https://gitee.com/openharmony/docs

[5] "Lycium framework." [Online]. Available: https://gitee.com/openharmony-sig/tpc_c_cplusplus/tree/master/lycium

[6] "Porting history of the libuv library." [Online]. Available: https://gitcode.com/openharmony-sig/tpc_c_cplusplus/blob/master/community/libuv/libuv_oh_pkg.patch?init=initTree

[7] "Openharmony community c/c++ software porting specifications." [Online]. Available: https://gitcode.com/openharmony-sig/tpc_c_cplusplus/blob/master/lycium/README.md

[8] C. Jw, "Qualitative inquiry and research design," *Choosing among five traditions*, 1998.

[9] "Functional differences from glibc." [Online]. Available: https://wiki.musl-libc.org/functional-differences-from-glibc

[10] "A method for rapidly porting the openharmony linux kernel." [Online]. Available: https://docs.openharmony.cn/pages/v5.1/en/device-dev/porting/porting-linux-kernel.md

[11] "glibc vs. musl." [Online]. Available: https://edu.chainguard.dev/chainguard/chainguard-images/about/images-compiled-programs/glibc-vs-musl/

[12] "Functional differences from glibc." [Online]. Available: https://wiki.musl-libc.org/functional-differences-from-glibc.html

[13] "Unsupported native apis." [Online]. Available: https://docs.openharmony.cn/pages/v4.1/zh-cn/application-dev/reference/native-lib/musl-peculiar-symbol.md

[14] "Tree-sitter tool." [Online]. Available: https://github.com/tree-sitter/tree-sitter

[15] "Google play store." [Online]. Available: https://play.google.com/store/apps

[16] "F-droid." [Online]. Available: https://f-droid.org/zh_Hans/packages/

[17] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: binary software composition analysis via intelligent binary source code matching," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[18] "Openharmony." [Online]. Available: https://docs.openharmony.cn/pages/v4.1/zh-cn/application-dev/reference/native-lib/musl-peculiar-symbol.md

[19] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[20] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.

[21] "An open-source community for developers worldwide, and an open-source code hosting platform." [Online]. Available: https://gitcode.com/openharmony-sig/

[22] "Learning arkts." [Online]. Available: https://docs.openharmony.cn/pages/v5.0/en/application-dev/quick-start/introduction-to-arkts.md

[23] K. Lamhaddab, M. Lachgar, and K. Elbaamrani, "Porting mobile apps from ios to android: A practical experience," *Mobile Information Systems*, vol. 2019, no. 1, p. 4324871, 2019.

[24] K. Madadipouya, "Critical evaluation of application porting in mobile platforms," *Journal of Engineering and Technology (JET)*, vol. 6, no. 2, pp. 9–17, 2015.

[25] G. L. Dams and J. T. Ogbiti, "Issues affecting testing activities in porting mobile applications," *Journal of Sciences and Multidisciplinary Research*, vol. 9, no. 3, 2017.

[26] H. Samet, M. D. Adelfio, B. C. Fruin, M. D. Lieberman, and B. E. Teitler, "Porting a web-based mapping application to a smartphone app," in *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, 2011, pp. 525–528.

[27] M. Rauter, J. Wachtler, and M. Ebner, "Porting a native android app to ios: Porting process shown by the example of the "schoolstart screening app"." *International Journal of Interactive Mobile Technologies*, vol. 17, no. 23, 2023.

[28] T. Tisdall, P. Chobharkar, and D.-K. Kim, "Challenges in porting enterprise applications to mobile platforms," *GetMobile: Mobile Computing and Communications*, vol. 22, no. 1, pp. 21–25, 2018.

[29] E. Jansson and N. Jonasson, "Speedflirt for android, a study of porting a mobile application and its effects on user experience," 2011.

[30] J. Hu, E. Lu, D. A. Holland, M. Kawaguchi, S. Chong, and M. Seltzer, "Towards porting operating systems with program synthesis," *ACM Transactions on Programming Languages and Systems*, vol. 45, no. 1, pp. 1–70, 2023.

[31] A. V. Zhadchenko, K. A. Mamrosenko, and A. M. Giatsintov, "Porting x windows system to operating system compliant with portable operating system interface," *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 7, 2020.

[32] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang, "Porting an autosar-compliant operating system to a high performance embedded platform," *ACM SIGBED Review*, vol. 11, no. 1, pp. 62–67, 2014.

[33] D. Cho and D. Bae, "Case study on installing a porting process for embedded operating system in a small team," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion*. IEEE, 2011, pp. 19–25.

[34] G. Oikonomou and I. Phillips, "Experiences from porting the contiki operating system to a popular hardware platform," in *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*. IEEE, 2011, pp. 1–6.

[35] M. Faisal and S. Montenegro, "Porting a real-time objected oriented dependable operating system (rodos) on a customizable system-on-chip," in *Advances in Intelligent Systems and Computing II: Selected Papers from the International Conference on Computer Science and Information Technologies, CSIT 2017, September 5-8 Lviv, Ukraine*. Springer, 2018, pp. 124–145.

[36] P. A. D. Legaspi, K. K. Khan, K. D. Santiago, D. F. Sayson, H. O. Aquino, C. V. J. Densing, J. R. E. Hizon, and L. P. Alarcon, "Porting an operating system on an arm-based sensor platform," in *TENCON 2015-2015 IEEE Region 10 Conference*. IEEE, 2015, pp. 1–3.