Understanding API Usage and Testing: An Empirical Study of C Libraries

Ahmed Zaki
ahmed.zaki@imperial.ac.uk
Imperial College London
London, UK

Cristian Cadar
c.cadar@imperial.ac.uk
Imperial College London
London, UK

Abstract

For library developers, understanding how their Application Programming Interfaces (APIs) are used in the field can be invaluable. Knowing how clients are using their APIs allows for data-driven decisions on prioritising bug reports, feature requests, and testing activities. For example, the priority of a bug report concerning an API can be partly determined by how widely that API is used.

In this paper, we present an empirical study in which we analyse API usage across 21 popular open-source C libraries, such as OpenSSL and SQLite, with a combined total of 3,061 C/C++ clients. We compare API usage by clients with how well library test suites exercise the APIs to offer actionable insights for library developers.

To our knowledge, this is the first study that compares API usage and API testing at scale for the C/C++ ecosystem. Our study shows that library developers do not prioritise their effort based on how clients use their API, with popular APIs often poorly tested. For example, in LMDB, a popular key-value store, 45% of the APIs are used by clients but not tested by the library test suite. We further show that client test suites can be leveraged to improve library testing—e.g., improving coverage in LMDB by 14.7%—with the important advantage that those tests are representative of how the APIs are used in the field.

For our empirical study, we have developed LibProbe, a framework that can be used to analyse a large corpus of clients for a given library and produce various metrics useful to library developers.

ACM Reference Format:
Ahmed Zaki and Cristian Cadar. 2025. Understanding API Usage and Testing: An Empirical Study of C Libraries. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Libraries provide reusable code for many applications. As a library becomes more popular, the demands on library developers in terms of fixing bugs, implementing new features, and testing the code increase. Understanding how the library's Application Programming Interfaces (APIs) are used can provide invaluable insight for developers. In particular, library developers would be able to prioritise their time and effort according to data retrieved from a representative sample of clients of the library. For instance, prioritising bug reports and feature requests is a difficult challenge that has attracted significant research [1, 11, 55, 63]. Without sufficient information about API usage, library developers can spend time and effort maintaining features that are never used by clients.

To better understand how API usage information can help library developers, we have conducted a large-scale empirical study of 21 popular open-source C libraries with a combined total of 3,061 C/C++ clients. Our empirical study is enabled by LibProbe, a framework we have developed and made available as open source, to analyse library usage information across a large number of clients.

Our study aims to understand how library APIs are used in the field, how well they are tested by the library test suites, and whether there is a correlation between the two. It also aims to understand whether the size, in terms of lines of code in an API has an impact on API test coverage. Finally, we investigate whether client test suites could be leveraged to improve testing of API implementations.

In our study, we define an API of a library as an entry function (exported symbol) of that library. We measure the size and coverage of an API implementation by considering only the code within the entry function itself, excluding any code in its callees. This design choice is further discussed in §2.1. For succinctness, in the rest of the paper, we use the terms *API implementation*, *API size* and *API coverage* to refer, respectively, to the code implementation, number of lines of code, and percentage of lines of code covered in the entry function.

1.1 Research Questions

Our empirical study answers the following research questions:

RQ1: What is the distribution of library API uses across clients? What percentage of a library's APIs are used by clients and how commonly do clients use the full set of APIs from that library? Does API utilisation depend on the number of APIs offered by the library? Is there a large difference in number of uses between the most and least used APIs?

RQ2: How well are API implementations tested by the library test suite? Does API implementation size matter? Is there a correlation between the API implementation size and the API test coverage achieved by the library test suite?

RQ3: Are APIs widely used by clients also well tested? Is there a correlation between the number of clients using an API and the API test coverage achieved by the library test suite?

EASE 2025, Istanbul, Türkiye
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

RQ4: Can API coverage be improved by using the client test suites? Is it possible to leverage the client test suites to better test libraries? This would be particularly valuable as the client test suites are more likely to reflect how the libraries are used in practice.

1.2 Contributions

To our knowledge, this is the first large-scale empirical study for C/C++ that tries to understand how library APIs are used in practice and correlate that information with test coverage, based on a large number of client applications. Our study includes 21 libraries and 3,061 clients hosted on the popular GitHub platform. Our major findings can be summarised as follows:

(1) *API testing compared to usage:* Many libraries have APIs that are used by a large number of clients, yet the testing of those API implementations is poor. Conversely, there are many APIs with few or no clients which are well-tested by the library developers. This shows that library developers can benefit from metrics about API usage to prioritise their testing efforts.
(2) *API utilisation by clients:* Most libraries have unused APIs, and the percentage of unused APIs does not depend on the number of APIs offered. This information can be used to improve API design and retire unnecessary APIs.
(3) *Improving API coverage using client test suites:* The test suites of library clients can be leveraged to improve API coverage. Such tests have the further advantage of being representative of how the APIs are used in the field. For example, we could improve coverage in Vorbis by 7.7% reaching 14 previously untested APIs by leveraging the test suite of a single client.

To conduct our study, we have developed a fast, lightweight framework for large-scale API usage analysis of C libraries and C/C++ clients that produces helpful metrics for library developers. We make our framework, LibProbe, and the results of our empirical study available as an artifact [61].

2 LibProbe

The high-level architecture of LibProbe is shown in Figure 1. The inputs to LibProbe are the libraries of interest and a dependency database. The latter consists of (C, L) pairs, signifying that client C uses library L, where C can itself be a library. We use the dependency database provided by CCScanner [52], which includes GitHub repositories for all entries. We discuss this database further in §3.1.

LibProbe starts by processing each library to obtain its set of APIs and the test coverage achieved on the code of each API by the library's test suite (§2.1). It then uses information from the dependency database to download and prepare for analysis all the known clients of the library (§2.2). LibProbe then analyses each client to extract all the uses of the library APIs (§2.3). Optionally, LibProbe determines the extra coverage achieved by the client in the library under analysis. Finally, LibProbe processes and summarises the collected data in the form of *JSON* files and graphs.

Scope and requirements. LibProbe is meant to analyse popular libraries, which may have dozens if not hundreds of C/C++ clients. This imposes several constraints on its design.

First and foremost, we need to avoid the need to build each client, which is both error-prone and expensive. This means we cannot use compiler infrastructures to parse and analyse the code. Instead, we rely on simple lexical analysis of the code to collect usage information. This is one of the reasons for which we restrict LibProbe to C libraries, as a lexical search for C++ APIs is more error-prone. However, LibProbe can process both C and C++ clients. We discuss this aspect in more detail in §2.3.

Second, we need a way to distinguish between client and library code in the common scenario in which the client codebase incorporates the library code. We discuss this aspect in §2.2.

Third, we need to be able to build the library itself into a shared library, as LibProbe uses the exported symbols from shared library archives as a step to determine the set of APIs of a library. As such, header-only libraries cannot be processed by LibProbe.

2.1 Library Processing

The first stage of LibProbe is to extract the set of exported symbols from each shared library file[1], which is often a superset of the APIs documented for that library. This is because some symbols are used only for communication between different modules of a library. For instance, this was the case for NCurses [38] and MbedTLS [33], as documented in our conversations with their developers [32, 37]. In the case of MbedTLS, the developers mentioned that functions exported and not part of the API are not made private or have no name mangling because they never got around to doing this in their build scripts. To tackle this, we filter out symbols that are not part of the library API by excluding those that do not appear in the headers of the library after installation. We do not solely rely on the installed header files since that would be less precise in the presence of macros.

LibProbe stores all APIs in a database to search for them later in the clients. It will also record the size (in number of ELOC) and test coverage (if provided) of each API implementation, by processing all the coverage files present in the library's directory. In our study, we measure test coverage in terms of *line coverage*, as reported by `GCov` [12] and `LCov` [28]. While high line coverage alone is not sufficient, it is nevertheless necessary; library test suites cannot find issues in code that is not exercised. It is worth noting that we process all *.gcno* files which are generated when the library is compiled. This can include test or other auxiliary files so our coverage reporting is an overapproximation of the actual library coverage. Generally, we make a best effort to identify specific directories that are part of the core library source code, otherwise, we process all compiled files and report the total coverage on those files.

The entry functions of an API represent a critical interface between the library and its clients. Such functions often perform input validation and high-level decisions concerning the functionality offered by the API. Depending on how the code is structured, in some cases the entry function may not reflect the complexity of the API implementation (because most of the core logic may be in the callees). We explored measuring the full size of an API implementation by aggregating the number of ELOC of all callees used by the API recursively. That approach suffered from two fundamental problems. First, callees can be shared between different APIs, which would make calculating the coverage for each API inaccurate. For instance, consider an API which is never tested, but for which its

[1]We do so by running `nm -CD library.so | grep " T "` on the shared library.

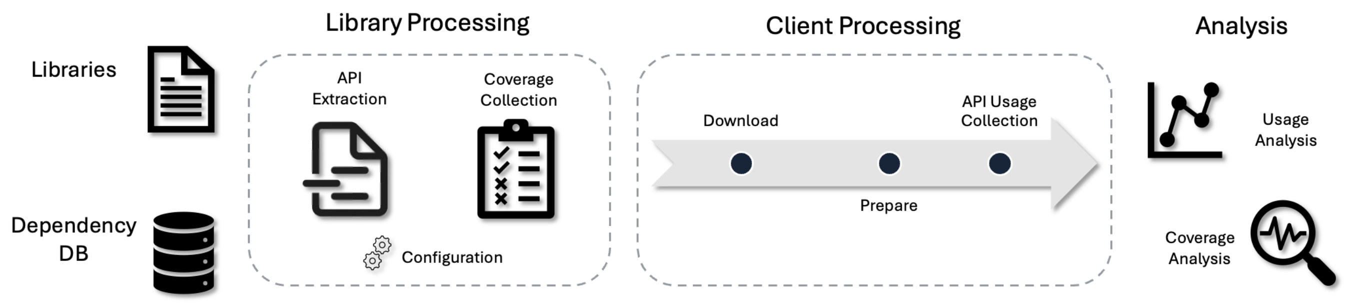

Figure 1: LibProbe architecture.

callees are exercised through other APIs; the API coverage in this case would be considered non-zero, when in fact the API is not exercised at all by the test suite. This problem extends recursively, in the same way, to the callees themselves. Second, the size of the API, in ELOC, would often be a large overestimate rather than an accurate measurement, as only a small part of the code of the callees might be used by the given API . In summary, we believe it is more meaningful to restrict our code coverage and size measurement to the API entry functions.

2.2 Client Preparation

Clients of a library sometimes include the library code in their own codebase to make it easier to build the code, and/or to ensure the use of specific library versions. To be able to get an accurate assessment of API usage, we have to exclude client directories which contain the library code. To do this, we use two approaches.

Our first approach handles Git submodules [13], when they are present in the client's codebase. We read the *.gitsubmodules* file and exclude all paths in it. This could include other dependencies, which is fine, as we are only interested in the client code itself.

Our second approach handles the case where the library code is added as a sub-directory. We list all the directories in the library's repository which include source files and collect all the file names in those directories recursively. Following that, we look for matches in the client's code. Given a client directory, we exclude it if 80% of the files in the client directory exist in the library directory (as long as the latter has more than two files). The reason for allowing a partial match is that sometimes clients use an old version of a library that may contain slightly different files. We arrived to the threshold of 80% through several experiments across library-client pairs to maximise accuracy.

Some libraries have a small number of source files with distinctive names, in which case we explicitly exclude them using their name. The reason we do not do this for all libraries is that many files have generic names, with different files being given the same name in the library and the client.

2.3 API Usage Collection

Our empirical study involves a large number of clients—CCScanner reports 3,198 different clients for our libraries. Therefore, our API usage collection method needs to be both fast and lightweight. In particular, this excludes techniques which rely on building each client codebase, such as those using a compiler framework like Clang [3]. In addition to taking considerable time, building such a large number of clients would likely be infeasible given the variety of build systems and dependencies involved.

Therefore, we set as a strict requirement to use a simple lightweight analysis based on textual search. We considered two approaches: one based on the Grep [16] text search utility, and the other based on the Weggli [58] semantic search tool.

Grep search. For each library API, we use a multi-step Grep pipeline on all source files in each client directory. We start by excluding comments by running the following command on the client root directory:

```
grep −rEIv  \/\/[^\ n   ]∗|\/\∗.∗[\∗\/]?|^\   s \∗  −−include=∗.cc −−
        include=∗.c −−include=∗.cpp −−include=∗.cxx −−include=∗.
        hh −−include=∗.h −−include=∗.hpp −−include=∗hxx −−
        exclude=< client_dir >  ..
```

This command outputs text in source files that excludes comments and specific client directories that we identified in §2.2. We then search for uses of an API across this text, while excluding string literal matches by the running the following two commands:

```
grep −E \b<api>\s ?\(
grep −Ev ".∗< api >.∗"
```

Weggli search. Weggli [58] is a semantic search tool for C and C++ codebases, which is built on top of `tree-sitter` [53]. We used Weggli to search for each library API in each client source file, excluding again the directories that were identified during the client preparation stage.

We compared the Grep and Weggli approaches to each other and to a tool built on top of Clang libtooling [4]. As discussed earlier, the latter would be too expensive to apply in practice but is used as a comparison baseline. Our Clang libtooling-based tool, further referred to as *libtool*, fetches all call expressions invoking APIs from the library of interest.

We randomly selected two libraries, each with five clients, to compare the three approaches. We selected: MbedTLS [33] with clients uacme [54], curl [5], OpenVPN [40], Lighttpd [27] and libCoAP [25]; and LMDB [29] with clients Kerberos [23], Recorder [43], LibEtPan [26], fapolicyd [6] and OSMExpress [41].

We performed two types of analysis; one on identifying the distinct APIs used, and the other on counting the number of uses

Table 1: Grep and WEGGLI precision and recall on clients of MBEDTLS relative to *libtool*.

	Distinct API Uses		Total API Uses	
	Grep	WEGGLI	Grep	WEGGLI
Client	P_D / R_D	P_D / R_D	P_T / R_T	P_T / R_T
UACME	0.94 / 1.00	0.92 / 0.97	0.95 / 0.99	0.90 / 0.97
CURL	0.93 / 1.00	0.92 / 0.86	0.81 / 1.00	0.80 / 0.84
OPENVPN	0.92 / 1.00	0.91 / 1.00	0.87 / 1.00	0.85 / 1.00
LIGHTTPD	0.63 / 0.98	0.64 / 1.00	0.57 / 0.92	0.58 / 1.00
LIBCOAP	0.98 / 1.00	0.92 / 0.95	0.96 / 1.00	0.89 / 0.90

Table 2: Grep and WEGGLI precision and recall on clients of LMDB relative to *libtool*.

	Distinct API Uses		Total API Uses	
	Grep	WEGGLI	Grep	WEGGLI
Client	P_D/ R_D	P_D/ R_D	P_T/ R_T	P_T/ R_T
KERBEROS	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
RECORDER	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
LIBETPAN	1.00 / 0.93	1.00 / 1.00	1.00 / 0.97	1.00 / 1.00
FAPOLICYD	0.95 / 0.95	1.00 / 0.95	0.95 / 0.96	1.00 / 0.97
OSMEXPRESS	1.00 / 1.00	1.00 / 0.92	0.81 / 0.89	0.76 / 0.86

for each API. The former is useful for determining how many of the library's APIs are being used by a client, while the latter for understanding the popularity of each API. We calculate Precision and Recall as

$$P_{D/T} = \frac{Tp_{D/T}}{Tp_{D/T} + Fp_{D/T}} \qquad R_{D/T} = \frac{Tp_{D/T}}{Tp_{D/T} + Fn_{D/T}}$$

P_D and R_D represents precision and recall for distinct API identification while P_T and R_T represents precision and recall for total uses for each API identified. We define Tp_D as the the number of distinct APIs identified by both the tool (Grep or WEGGLI) and *libtool*; and Fp_D as the number of distinct APIs identified by the tool but not by *libtool*. Fn_D is defined as the number of distinct APIs identified by *libtool* but not identified by the tool. Tp_T, Fp_T, and Fn_T are defined similarly for total API uses.

Tables 1 and 2 show the results. Grep generally performed better than WEGGLI for all clients of MBEDTLS except for LIGHTTPD, where both tools had a low precision. For clients of LMDB, Table 2 shows that both tools were largely similar in performance except for LIBETPAN and OSMEXPRESS.

Looking closer at the low precision of both tools on LIGHTTPD we found that Grep reported 37 false positive distinct APIs while WEGGLI reported 39. Almost all of the reported APIs were either inside #ifdef directives that look for a certain MBEDTLS version/configuration or were in source files that were not part of the build of the client. Since *libtool* runs on the source *after* it gets pre-processed and takes into consideration the build configuration, some code gets removed. As such, if an API is within an #ifdef that looks for a certain library version or perhaps a debug build, *libtool* will miss such uses. This was also reflected when looking at the total uses reported by both Grep and WEGGLI. Grep reported 66 false positive uses compared to *libtool* while WEGGLI reported 72. The majority of the false positives were due to either #ifdef directives looking for a version of MBEDTLS or due to files not included in the build. It is possible to argue that these uses are not really false positives. The uses are in client's source files but not included in the standard build configuration or only used when certain conditions are met. This does still mean that under certain conditions such APIs could be used by the client.

WEGGLI performed better than Grep on LIGHTTPD, LIBETPAN and FAPOLICYD. These differences were due to two main factors. First, Grep discards all the lines with comments. Therefore, if a comment is on the same line as a call, it incorrectly gets discarded. While we could improve this aspect, it is difficult to come up with a general solution without parsing the code. The second source of false negatives is when an API function is passed as a function pointer. This is due to the regular expression we used to find API uses, which only looks for call expressions; through experimentation, we found that removing this restriction resulted in a higher number of false positives.

In the majority of cases, Grep performed better than WEGGLI. For instance, in LIBCOAP, WEGGLI was unable to identify the usage of 4 distinct APIs (false negatives) while Grep reported all APIs used by the *libtool*. Similarly, in term of total uses, WEGGLI missed 12 uses, which Grep successfully reported.

Analysing WEGGLI's misses, we identified two causes. A large fraction of the misses come from WEGGLI's failure to parse some functions that use #ifdef directives heavily. Usage of #ifdef directives inside functions resulted in failures of tree-sitter, which WEGGLI uses to generate an AST. Since Grep is lexical, it has no issues identifying these uses. Another limitation of WEGGLI is with respect to macros. WEGGLI is unable to process macro definitions, and as such misses completely API uses that are wrapped in a macro. We confirmed this by raising an issue on the WEGGLI GITHUB project [59].

In summary, both WEGGLI and Grep have some limitations, but overall we were more concerned about WEGGLI's inability to process macros and #ifdef directives, which are common in C code. Combined with the fact that WEGGLI does not seem to be actively developed anymore (which means that any issues encountered during our study could be difficult to resolve), we have decided to use Grep in LIBPROBE.

3 Empirical Study

In this section, we present the results of our empirical study involving 21 libraries and 3,061 clients. We start by presenting our methodology in §3.1, after which we present the results for the first research question in §3.2 and for the last three in §3.3.

3.1 Methodology

Dependency database. In our study, we make use of CCSCANNER [52], which provides a database of dependencies for 24K C/C++ GITHUB projects. In particular, each entry in the CCSCANNER database consists of a GITHUB repository and its dependencies.

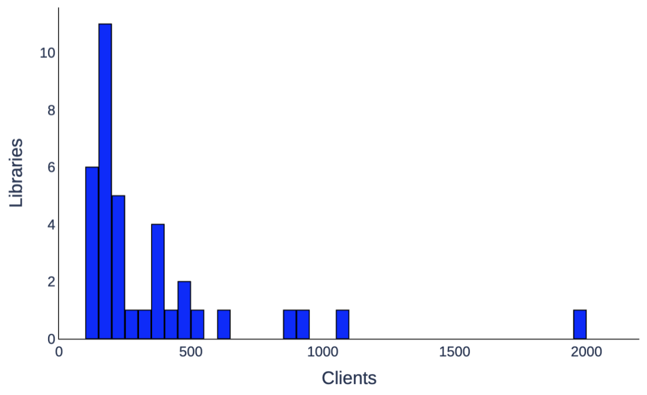

Figure 2: Histogram of the number of libraries from CCScanner with at least 100 clients.

Library selection. From the total of 24K repositories, 229 were not available anymore on GitHub, and thus we discarded them. We then identified the number of C repositories in the CCScanner dataset. Using each repository's GitHub metadata, we identified 10,291 repositories that had C as their main language. Out of these, 2,520 were dependencies of repositories in the CCScanner dataset. In terms of number of clients, 2,067 (82%) of the dependencies had less than 10 clients, 354 (14%) had between 10 to 49 clients, 54 (2%) between 50 and 99 clients, and 45 (2%) at least 100 clients. In our study, we are interested in popular libraries, so we have chosen to focus on the 45 libraries with at least 100 clients.

We manually reviewed the 45 dependencies, and eliminated those which are clearly not libraries, such as CMake, and Systemd. In total, we identified 8 such non-libraries, leaving us with 37 libraries with at least 100 clients.

Figure 2 shows a histogram of the number of libraries with at least 100 clients. To avoid long processing cycles in our study, we select the 32 libraries that have between 100 and 550 clients, which represent the majority of the libraries (the contiguous bars on the left in Figure 2).

In order to identify the APIs available for use by clients, LibProbe requires each library repository to be built into a shared library and installed. We manually attempted to build each of the 32 libraries. However, we could not build 11 of them, as some libraries were header-only, while others required specific system-level environment configurations, such as the Android SDK.

This process left us with 21 libraries to use for our study, which are shown in Table 3, together with their size in ELOC, and the number of APIs they provide. Our selection includes libraries with diverse applications such as encryption (OpenSSL [39]), database management (SQLite [48]), media (SDL [45]), compression (Zip [65]), rendering (FreeType [10]) and general purpose libraries (GLib [15]). The ELOC measurements are provided by GCov and LCov, so they reflect the size of these libraries as compiled on our platform.

Client selection. For each of the 21 libraries, we select all their available clients in the CCScanner database. Note that while the libraries are all written in C, the clients can be either C or C++ projects. Table 3 shows the number of clients downloaded and processed for each library and, between parenthesis, the number of clients where we have actually identified API uses.

Table 3: Libraries analysed, together with their size in ELOC, number of APIs, number of clients reported by CCScanner and found by us, and some examples of clients.

Library	ELOC	APIs	Clients	Example clients
FFmpeg	480,871	880	236 (120)	RoboMaster, Telegram
FFTW3	59,846	66	151 (65)	Atomify, Synfig, Cava
FreeType	82,996	219	355 (263)	Ogre, OpenHarmony
GLEW	22,626	9	490 (387)	openglText, imgui
GLib	213,547	4,417	365 (302)	Mutter, GTK
GSL	122,985	5,222	162 (119)	OpenPilot, GDL
HDF5	344,814	983	215 (159)	RDPFuzz, OpenCV
HIDAPI	1,351	24	100 (83)	OpenSCAD, OpenFPGA
JEMalloc	16,993	24	159 (144)	SpiderMonkey
LMDB	4,469	56	131 (115)	PowerDNS, Caffe, Dali
LuaJIT	6,671	148	284 (256)	Redis-storage
LZ4	5,934	100	498 (31)	FreeBSD, NVBio
MbedTLS	36,437	885	179 (134)	OpenVPN3, Mqtt-c
NCurses	14,716	499	156 (136)	WeeChat, Heimdal
OpenSSL	461,116	5,282	444 (131)	Telegram iOS, Moai
SDL	56,458	838	196 (70)	AliOS, DreamShell
SQLite	211,058	269	428 (143)	Gideros, OrangeC
Vorbis	6,688	78	165 (130)	Spring, Pindrop
xxHash	2,328	49	343 (120)	FreeBSD, Orbit
Zip	4,909	37	109 (81)	Assimp, Radare
Zstandard	24,254	186	189 (72)	Grok, Qemu

In total, CCScanner listed 3,198 unique repositories as clients of our 21 libraries. As seen in Table 3, there is a substantial difference between the number of clients reported by CCScanner and those confirmed by us for each library. We identified two reasons for this discrepancy. First, we noticed that some of the dependency information in CCScanner is incorrect. This could be due to the evolution of the clients over time since the publication of the CCScanner dataset. Second, we explicitly remove third-party dependencies added via Git submodules, as we are interested only in how the client itself uses libraries, while CCScanner would consider that the client uses a library even when only one of its third-party dependencies does.

Even after filtering out these clients, we are left with 3,061 clients across all libraries, of which 2,070 are unique.

Running LibProbe. For each library and client, we run the three LibProbe stages discussed earlier: library preparation (§2.1), client preparation (§2.2) and API usage collection (§2.3). Running LibProbe took approximately 96 hours on a machine with an AMD EPYC 7302P 4 Core Processor at 3.6GHz, and 32GB RAM. The OS was Ubuntu 22.04 LTS, and our compiler was GCC 11.4.0.

3.2 Usage Analysis

To address the first RQ, we study how APIs are used in practice, in particular what percentage of a library's APIs are used per client,

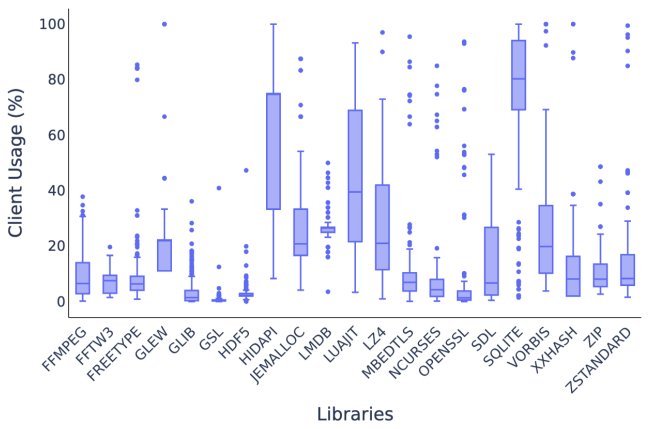

Figure 3: Percentage of library APIs used by each client.

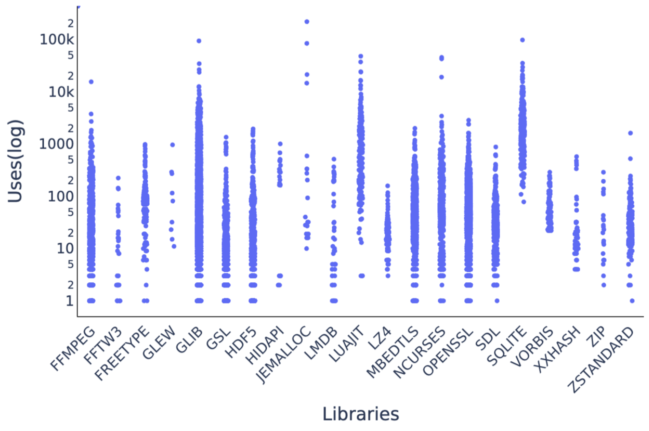

Figure 4: Number of uses for each library API (log scale).

if API utilisation depends on the number of APIs offered by the library, and if there is a large difference in number of uses between the most and least used APIs.

RQ1: What is the distribution of library API uses across clients?

Percentage of APIs used per client. Figure 3 shows the percentage of library APIs used by *each* client. A few libraries, like xxHash [60], HIDAPI [21], Vorbis [57], SQLite [48], GLEW [14] and Zstandard [66], had clients that used 100% of their APIs. All of those libraries have a relatively small number of APIs, as shown in Table 3, except SQLite which has 269 APIs. For most libraries, the upper quartile usage is under 40% of the library API. Some libraries, such as FFTW3 [7], GSL [17] and GLib [15], do not have clients that exceed 40% API utilisation.

Unused APIs. Table 4 shows the number of unused APIs and the percentage of unused APIs for each library. Looking at libraries with a large number of APIs, some have very high utilisation, such as OpenSSL, where only 5% of its 5,282 APIs are unused, while others have low utilisation, such as HDF5, where 61% of its 983 APIs are unused. The same variation holds for libraries with a small number of APIs: for instance, all of xxHash's 49 APIs are used, while 38% of Zip's 37 APIs are unused. Clearly, the number of APIs available from a library does not correlate with usage. This can be due to some libraries requiring clients to use many APIs to achieve a single functionality. As can be seen, 16 of the 21 libraries have at least one unused API, with the percentage of unused APIs reaching 61% for HDF5 [19] and 53% for GSL [17].

Table 4: Unused APIs per library, sorted in descending order of the percentage of unused APIs.

Library	APIs		
	Total	Unused	Unused %
HDF5	983	603	61%
GSL	5,222	2,459	53%
SDL	838	432	52%
FFTW3	66	33	50%
Zip	37	14	38%
GLib	4,417	1,521	34%
FFmpeg	880	221	25%
LMDB	56	10	18%
LZ4	100	17	17%
JEMalloc	24	3	13%
MbedTLS	885	96	11%
FreeType	219	17	8%
OpenSSL	5,282	274	5%
NCurses	499	12	2%
LuaJIT	148	2	1%
Zstandard	186	1	1%
GLEW	9	0	0%
HIDAPI	24	0	0%
SQLite	269	0	0%
Vorbis	78	0	0%
xxHash	49	0	0%

We took a closer look at the unused APIs across libraries and found that they fell into three categories: APIs covering secondary functionality, such as helper and debugging APIs; APIs with a similar functionality to more popular APIs; and less-used modules.

Libraries like OpenSSL, Zip, LMDB and FreeType have many helper-type APIs that were unused. In the case of FreeType and LMDB, many seem to be getter- and setter-type functions such as *FT_Set_Log_Handler* and *mdb_env_set_flags*.

Some libraries have several APIs performing more or less the same functionality as other more popular APIs. For example, users of Zip can use *zip_close* instead of the unused APIs *zip_stream_close* and *zip_cstream_close*.

Libraries that offer multiple modules see little usage of the APIs in some of the modules; this is the case in OpenSSL, FFTW3, HDF5 and MbedTLS. For instance, in the case of FFTW3, the majority of the unused APIs are related to the *Guru* module. This seems to offer a fragile API, which users avoid. According to the documentation, *"For those users who require the flexibility of the guru interface, it is important that they pay special attention to the documentation lest they shoot themselves in the foot"* [8]. A simple internet search returns multiple questions on *Stack Overflow* on how to use this interface [49–51].

Number of API uses. Figure 4 shows, for each library, the distribution of API uses across clients, for all APIs that are used. The y-axis is shown in log scale. We can see that for some libraries, such as GLib [15], NCurses [38] and JEMalloc [22], there is a wide difference between the largest and smallest number of uses.

The information collected as part of this RQ can be of great value to developers in prioritising API development and testing. Information about the API utilisation rate in clients can help in designing tests and potentially consolidating the library's APIs. Knowing that clients on average use a large number of APIs from the library, such as for SQLite, can help the developers of the library construct tests that encompass many of the library's APIs. The data on the API use distribution could be used to focus testing efforts on highly used APIs, while feature requests in little used APIs could be deprioritised. This information would also allow developers to retire unused APIs and assess the impact of changing an API, as changes to highly used APIs can potentially break many clients. As we will see in §3.3, development effort often does not reflect how the library is used in practice.

As a library developer, knowing that the average API utilisation is high could indicate that the APIs are tightly coupled together. This data can help in designing tests and potentially consolidating the library's APIs. The distribution of API uses, including data on the most used and completely unused APIs, can be leveraged to retire APIs, assess the impact of API changes, and prioritise development and maintenance effort.

3.3 API Coverage Analysis

In this part, we aim to understand how API coverage correlates with API implementation size and client usage, and whether clients could be used to improve a library's test suite. To answer these questions, we had to restrict ourselves to libraries that include an automated test suite that is easily runnable from the repository of the cloned project and for which we can gather reliable API coverage statistics. This is typically the test suite used by developers who contribute to the library. In our set of libraries, we identified cases such as LuaJIT [31] and FreeType [10] which do not include an automated test suite in the library repository. In LuaJIT, contributors rely on other test suites to test their pull requests [30]. For FreeType, developers rely on *OSS-Fuzz* [46] for testing (we confirmed the lack of a test suite by asking the developers [9]). NCurses and HIDAPI have interactive test suites that require user input. In the end, we identified 16 libraries for which we could easily run their test suites and obtain coverage.

RQ2: How well are API implementations tested by the library test suite? Does API implementation size matter? We run the test suite of each library and compute the coverage achieved in each API implementation. For FFTW3, the developer test suite uses random values for testing, which results in different coverage on each execution. We ran the test suite nine times and found out that the overall coverage ranges from 36.7% to 93.2%. The coverage measurements were 36.7%, 93.2%, 93.1%, 40.7%, 51.7%, 78.6%, 51.5%, 53.2% and 89.9%. Since the test suite of FFTW3 is non-deterministic, we considered the median run in graphs and tables.

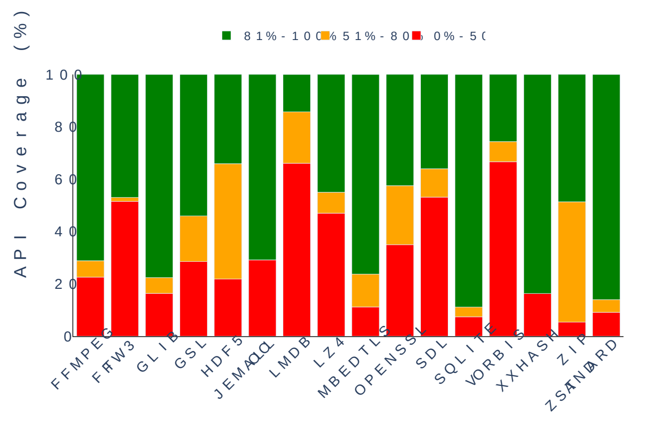

Figure 5: Number of APIs with coverage under 50%, between 50% and 80%, and over 80%.

Table 5: API coverage bucketed by API size. TCov is the total line coverage in the library code. Cov. is the combined line coverage for the API implementations in each size bucket.

Library	TCov	≤ 20		> 20	
		APIs	Cov.	APIs	Cov.
FFmpeg	60.5%	703	78.2%	143	74.3%
FFTW3	51.7%	66	43.3%	0	-
GLib	73.3%	3892	79.6%	328	78.6%
GSL	86.3%	1729	69.8%	407	75.5%
HDF5	69.3%	766	61.7%	217	61.8%
JEMalloc	84.5%	16	63.5%	8	78.3%
LMDB	38.2%	49	28.1%	7	25.6%
LZ4	70.4%	93	63.1%	7	44.2%
MbedTLS	78.9%	663	83.2%	222	80.6%
OpenSSL	63.7%	4520	59.0%	622	64.1%
SDL	18.3%	746	42.3%	84	45.8%
SQLite	61.0%	243	91.7%	26	87.3%
Vorbis	58.9%	59	33.7%	19	36.8%
xxHash	76.0%	47	89.2%	2	100.0%
Zip	42.5%	29	81.6%	8	63.1%
Zstandard	71.1%	166	88.2%	20	84.5%
Fully Covered		6,738	48.8%	286	13.4%

Figure 5 shows how many APIs have coverage under 50%, between 50% and 80%, and over 80% for each library. As we can see, there are libraries with generally high API coverage (such as SQLite and Zstandard) and libraries with generally low API coverage (such as LMDB and Vorbis). Overall, the number of poorly tested API is high, showing the limited budget available for testing.

Table 6: APIs used by clients but not tested, sorted in descending order by their percentage in each library.

Library	Number of APIs	Percentage
Vorbis	51	65%
LMDB	25	45%
LZ4	45	45%
OpenSSL	1560	29%
SDL	182	22%
FFTW3	14	21%
JEMalloc	4	17%
xxHash	8	16%
FFmpeg	128	14%
MbedTLS	87	10%
Zstandard	16	8%
SQLite	19	7%
GLib	335	7%
GSL	262	5%
HDF5	36	4%
Zip	1	3%

We next contrast API implementation size with API test coverage to understand whether there is any correlation between the two. Table 5 shows the coverage of the APIs split by size. We grouped APIs into two buckets: small APIs with up to 20 ELOC, and larger APIs with over 20 ELOC. For example, SDL has 746 APIs that each have up to 20 ELOC with a total coverage, across those APIs, of 42.3%; and 84 APIs which have each over 20 ELOC, with a total coverage of 45.8%.

What is evident from Figure 5 and Table 5 is that many libraries do not focus on API coverage, although ideally this should be close to 100% as discussed in §1. We also counted the number of fully covered APIs in each group as shown in Table 5: 48.8% of the 13,787 APIs with up to 20 ELOC have 100% coverage while only 13.4% are fully covered in the greater than 20 ELOC bucket.

Our analysis suggests that many APIs are poorly tested—or not tested at all—and that smaller APIs are easier to test. Library developers should use coverage data to direct their testing efforts.

RQ3: Are APIs widely used by clients also well tested? An important question is whether APIs which are widely used in the field are also well-tested. Ideally, developers would spend more resources writing tests for popular APIs, and fewer for those APIs which are rarely used.

Figure 6 shows coverage of each API against usage in our data set. The blue bars show the coverage of each API implementation, with the bars sorted in ascending order. For each API, the figure also shows, as a red dot, the percentage of clients that use that API. In almost all libraries, we can see a discrepancy between tested APIs and usage by clients. Very few library, such as Zstandard and Zip, have good overall testing of their APIs relative to usage. For example, *mdv_env_info*, from LMDB, is used by 53% of the library's clients yet it is not tested by the library's test suite. Similarly, *ov_info*, from Vorbis, is used by 81% of the library's clients, with no test coverage.

Table 7: API coverage improvement achieved by the clients' test suites. For each library, we list the total extra coverage added to the library, the number of newly covered APIs, and the number of APIs where coverage was improved.

Library	Client	Extra TCov	APIs New	APIs Improved
LMDB	Knot DNS	14.7%	4	4
Vorbis	SFML	7.7%	14	1
FFTW3	CAVA	2%	2	0
SDL	UFO Alien Invasion	0.3%	6	0

Table 6 shows for each library the number of APIs used but not tested by the library test suite, and their percentage from the total number of library APIs. As can be seen, these percentages vary from only 3% in Zip to 65% in Vorbis, with a median of 15%.

In Figure 6 we can also see cases where the number of clients is low yet the coverage is high. While this is not a problem per se, the overall picture shows that developers could do a better job prioritising their testing efforts if they had information about how their APIs are used in the field.

There is a clear discrepancy between API usage by clients and API implementation test coverage. Library developers should use API usage and coverage information to better prioritise their efforts and ensure the most used APIs are well tested and optimised.

RQ4: Can API coverage be improved by using the client test suites? RQ2 and RQ3 have shown that many APIs used by clients are poorly tested, or not tested at all, by the library test suite. We show that many clients already incorporate tests that exercise the library APIs, and those tests could be used by library developers to enhance testing of the library. In previous work, we explored techniques for extracting these tests automatically from client codebases, such that they can be used in the library's test suite without any dependence on the client code or other libraries [62].

Since we are interested in evaluating whether we can improve API coverage in libraries, we focused our attention on libraries that would benefit the most from this; libraries that have more than 20% of their APIs used by clients but not tested by library test suites. Looking at Table 6 we selected six libraries for our evaluation; Vorbis, LMDB, LZ4, OpenSSL, SDL, and FFTW3. For each of these libraries, we identify the top ten clients and use them to check if running their test suites would lead to an increase in API coverage. We exclude any clients that are complex to build, such as *FreeBSD*, could not be built on a Linux system, or do not have a test suite. For each client we attempt to build and run the test suite to see if there is an increase in API coverage in the target library. Once we find a client that increases API coverage in the target library we report it and stop.

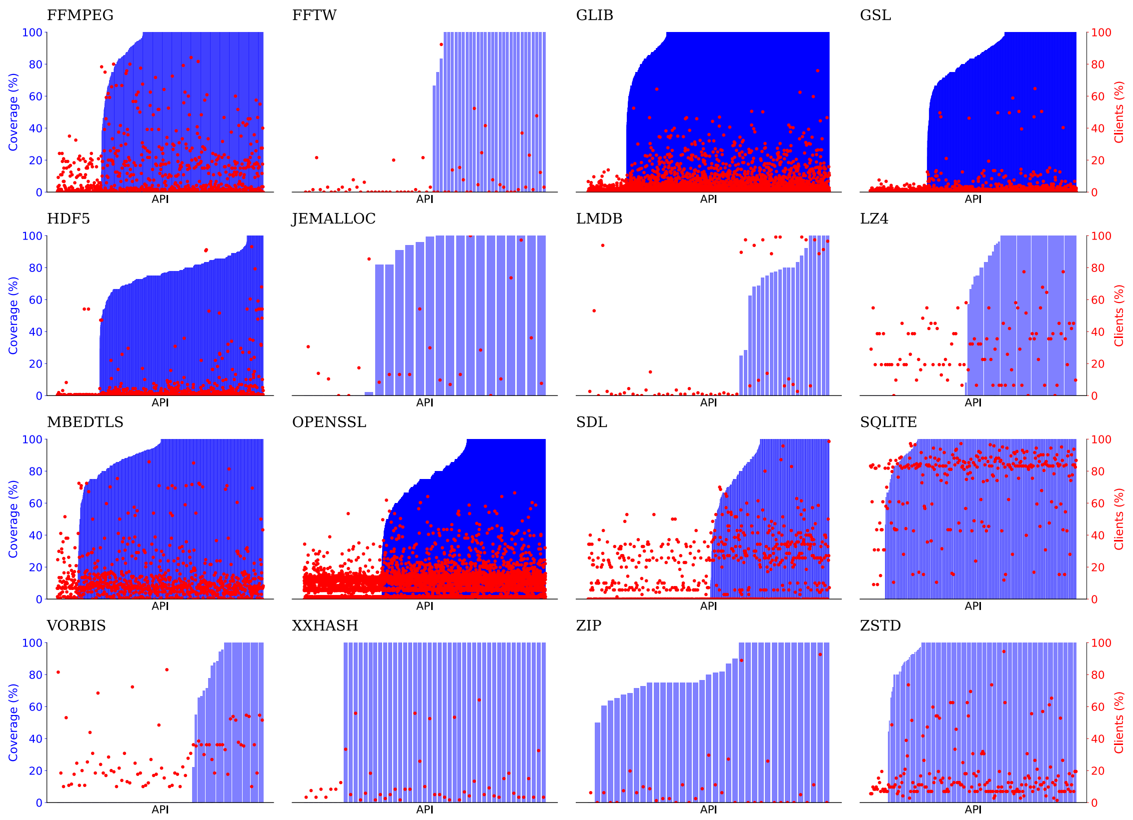

Figure 6: API coverage for each library. The blue bars represent the coverage for each API, while the red dots show the percentage of clients using that API.

For LZ4 and OpenSSL we were not able to improve coverage by building and running the test suites of the top 10 clients. It is worth noting that LZ4 had one of the lowest number of actual clients using the library. Several of those clients were operating systems or required specific hardware (such as recent GPUs), which made it difficult to build them.

We were able to increase API coverage in LMDB, Vorbis, FFTW3 and SDL. For LMDB we used Knot DNS [24], a high-performance authoritative-only DNS server; for Vorbis we used SFML (Simple and Fast MultiMedia Library) [47]; for FFTW3 we used CAVA [2], a cross-platform audio visualizer; and for SDL we used a game called UFO Alien Invasion [56].

Table 7 shows the improvements in terms of coverage. For each library, we list the number of APIs which were *newly covered* as a result of executing the client test suite and the number of APIs where improved coverage was achieved.

For SDL, we cover six previously uncovered APIs: two become fully covered (100%) while the other four achieve coverage of over 60%. For LMDB, we improve the testing of eight APIs, four of which were previously uncovered. Two of the eight APIs become fully covered, one of them uncovered before. For Vorbis, we improve testing of fifteen APIs. Six of them become fully covered, all of them uncovered before. For FFTW3, which has a non-deterministic test suite, we ran CAVA's test suite nine times and consistently cover two new APIs that were previously uncovered: *fftw_alloc_real* and *fftw_alloc_complex*.

For LMDB and Vorbis, 123 and 247 new lines were covered in the API implementations of the libraries respectively. API line coverage in Vorbis went from 35.9% to 51.3% while in LMDB from 26.6% to 44.0%. For the other two benchmarks, the numbers of extra new lines covered was less substantial.

In summary, we were able improve API coverage using client test suites in 4 out of 6 libraries. This shows that library developers can leverage clients that use their libraries to generate tests that reflect how the library APIs are used in practice.

3.4 Threats to Validity

Our study has several threats to validity.

To handle the large number of clients, we used textual search for API use identification (see §2.3), which may introduce some false positives and negatives. We tried to minimise these issues by conducting an initial study to choose the most precise scalable method. Based on our observations, we believe such instances are not significant enough to invalidate the insights gathered from our empirical study.

When we measure size and line coverage for the APIs, we restrict our measurement to the lines in the implementation of the API entry functions—see §2.1 for an extended discussion.

The conclusions of our study might not generalise beyond the libraries and clients examined. However, our 21 libraries are representative of popular C libraries and the clients considered per library are high enough in number and popularity to capture how the libraries are used in practice.

As with any large-scale study, we cannot dismiss the possibility of implementation errors. To mitigate this threat, we double-checked the numbers, and provide an artifact.

4 Related Work

To our knowledge, our study is the first that contrasts usage and test coverage of C libraries in the C/C++ ecosystem. We study 21 libraries with the objective of offering valuable insights to library developers, rather than to the users of the libraries.

There are several studies that look at API usage in the JAVA ecosystem. Qiu et al. [42] performed an empirical study on API usage for 5,000 open-source JAVA projects. This study looked at usage across core and third-party JAVA libraries. It analysed 16,329 distinct third-party libraries and found that only 15 were used by over 10% of the projects in their data set, while 265 were used by 1%, and 9,830 by a single project each. In our study, out of 2,520 dependencies, 82% had less than 10 users, while 14% had 10 to 49 users, and 2% had 50 to 99 users. As we showed in Figure 2, only two dependencies had more than 1,000 users. The study also investigated the usage of the JAVA core library by measuring how much of the library is used by projects in the corpus. The paper reports that 41.2% of the core methods of JAVA 8 were never used across their client corpus. In our analysis, out of the 21 libraries, four had 50% or more of their API unused. The study was conducted with respect to packages and classes, which is not directly comparable to API usage in C libraries.

Other studies analyse library usage from the *clients' perspective* [20, 64]. Hejderup and Gousios [20] assess the effectiveness of JAVA project test suites in covering usages of third-party libraries. The aim of the work is to assess how reliable test suites are as a means to evaluate the compatibility of updated library versions. The study did not explore if client test suites can be leveraged to improve library coverage, instead it focused on determining whether clients test all the APIs they use from direct dependencies.

Zhong and Mei [64] investigate how seven JAVA applications use internal and external APIs. The emphasis of the study is on understanding how clients use APIs by understanding common ways clients call different types of elements of an API. The paper studies how frequently APIs are used and find out that for most libraries only a small portion of APIs are called. We perform our study on 3,061 C/C++ clients and show in §3.2 that for most libraries average client usage sits below 40%. But we also show that full utilisation of the API is possible for some libraries.

Harrand et al. [18] analysed API usage for 94 JAVA libraries across 829,410 clients. The objective of their research was to explore the contradiction between Hyrum's Law and findings that show that for most libraries only a fraction of their APIs are used by clients. Our evaluation shows that five libraries had full API usage. Our findings align with those of Harrand et al. [18] in that in some cases, with enough clients, all APIs of a library are used (Hyrum's Law), yet at the same time most APIs can be significantly reduced and still fulfil the needs of the majority of the clients.

Schittekat et al. [44] conducted an evaluation on four Python packages across 14 clients to assess whether using the tests from the clients can improve coverage in the libraries. They were able to improve coverage in two out of the four packages. The improvements range from a maximum of 28% increase to a minimum of 1%. The paper did not investigate whether new APIs were covered from the Python packages.

Prior work has also analysed library usage with the objective of detecting breaking changes [34–36]. Mujahid et al. [36] performed an empirical study on 391,553 *npm* packages to evaluate if the tests from client projects can be used to detect breaking changes in packages. The paper found that client tests can cover up to 47% of the code for the target dependency but did not look at improving the coverage of the target dependencies, which is what we show is possible in our study. McDonnell et al. [34] conduct an empirical study to understand how clients using the Android SDK keep up with changes in the API. The study looks at how APIs provided by the Android SDK change over time and how clients catch up with those changes. Similar to [36] and [35], the study is more concerned with how changes in the library, as it evolves, impacts clients.

5 Conclusion

Libraries represent an indispensable component of the software ecosystem. Unfortunately, library developers often have little knowledge of how their code is used in practice. In this paper, we present a large-scale empirical study in which we analyse API usage across 21 C libraries and 3,061 C/C++ clients. We developed LIBPROBE, a lightweight analysis framework that can provide valuable insights to library developers regarding how their APIs are used in the field, to help them prioritise their efforts in maintaining, improving, and testing their libraries. Our study shows that library developers do not prioritise their effort based on how clients use their APIs—popular APIs are often poorly tested, with rarely-used ones well tested instead. We further show that client test suites can be leveraged to improve library testing, with the important advantage that those tests are representative of how the APIs are used in the field.

6 Acknowledgements

We thank Martin Nowack for his feedback on the text. This research has received funding from from the Engineering and Physical Sciences Research Council (EPSRC) via a PhD studentship, and from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 819141).

References

[1] Mamdouh Alenezi and Shadi Banitaan. 2013. Bug reports prioritization: Which features and classifier to use?. In *2013 12th International Conference on Machine Learning and Applications*, Vol. 2. IEEE, 112–116.

[2] Cava [n. d.]. Cross-platform Audio Visualizer - Cava. https://github.com/karlstav/cava.

[3] Clang [n. d.]. clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[4] Clang LibTooling 2023. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[5] curl [n. d.]. Curl - command line tool and library for transferring data with URLs. https://curl.se.

[6] fapolicyd [n. d.]. File Access Policy Daemon. https://github.com/linux-application-whitelisting/fapolicyd.

[7] FFTW Website [n. d.]. Fastest Fourier Transform in the West - FFTW. https://www.fftw.org/.

[8] FFTW3 Guru Interface [n. d.]. FFTW3 Guru Interface. https://www.fftw.org/doc/Guru-Interface.html.

[9] FreeType Testing Issue [n. d.]. FreeType Testing Issue. https://gitlab.freedesktop.org/freetype/freetype/-/issues/1272#note_2329977.

[10] FreeType Website [n. d.]. The FreeType Project. https://freetype.org/.

[11] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T Barr. 2021. The assessor's dilemma: Improving bug repair via empirical game theory. *ACM Transactions on Software Engineering Methodology (TOSEM)* 47, 10, Article 6 (Oct. 2021). https://doi.org/10.1109/TSE.2019.2944608

[12] gcov – A Test Coverage Program [n. d.]. gcov – A Test Coverage Program. gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[13] Git Submodules [n. d.]. Git Submodules. https://git-scm.com/book/en/v2/Git-Tools-Submodules.

[14] GLEW Website [n. d.]. OpenGL Extension Wrangler Library (GLEW). https://glew.sourceforge.net/.

[15] Glib Website [n. d.]. GLib. https://docs.gtk.org/glib/.

[16] Grep [n. d.]. Grep. https://www.gnu.org/software/grep/.

[17] GSL Website [n. d.]. GNU Scientific Library (GSL). https://www.gnu.org/software/gsl/.

[18] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit Baudry. 2022. API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages. *Journal of Systems and Software* 184 (2022), 111134.

[19] HDF5 Website [n. d.]. High-performance data management and storage suite - HDF5. https://www.hdfgroup.org/solutions/hdf5/.

[20] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software* 183 (2022), 111097.

[21] HIDAPI Website [n. d.]. A simple cross-platform library for communicating with HID devices - HIDAPI. https://libusb.info/hidapi/.

[22] JeMalloc Website [n. d.]. JeMalloc. https://jemalloc.net/.

[23] Kerberos V5 [n. d.]. Kerberos v5 - The Network Authentication Protocol. https://web.mit.edu/kerberos/.

[24] Knot [n. d.]. Knot DNS. https://www.knot-dns.cz/.

[25] libcoap [n. d.]. libcoap - C-Implementation of CoAP. https://libcoap.net/.

[26] libetpan [n. d.]. LibEtPan - Mail Framework for C Language. https://libcoap.net/.

[27] lighttpd1.4 [n. d.]. lighttpd1.4. https://www.lighttpd.net/.

[28] Linux Test Project. [n. d.]. LCOV. https://github.com/linux-test-project/lcov.

[29] LMDB Website [n. d.]. Lightining Memory-Mapped Database - LMDB. http://www.lmdb.tech/doc/.

[30] LuaJit PR [n. d.]. LuaJit PR. https://github.com/LuaJIT/LuaJIT/pull/54.

[31] LuaJIT Website [n. d.]. The LuaJIT Project - LuaJIT. https://luajit.org/.

[32] MbedTLS API discussion [n. d.]. MbedTLS API discussion. https://lists.trustedfirmware.org/archives/list/mbed-tls@lists.trustedfirmware.org/thread/JVZWDVDMNIJXOTZA7FTYX2KBL65KDXL6/.

[33] MbedTLS Website [n. d.]. Mbed TLS. https://www.trustedfirmware.org/projects/mbed-tls/.

[34] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM'13)* (Eindhoven, The Netherlands).

[35] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *Proc. of the 32nd European Conference on Object-Oriented Programming (ECOOP'18)* (Amsterdam, The Netherlands). Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[36] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th international conference on mining software repositories*. 466–476.

[37] NCurses API discussion [n. d.]. NCurses API discussion. https://lists.gnu.org/archive/html/bug-ncurses/2024-04/msg00021.html.

[38] Ncurses Website [n. d.]. Ncurses (new curses). https://invisible-island.net/ncurses/.

[39] OpenSSL Website [n. d.]. OpenSSL. https://openssl-library.org/.

[40] openvpn [n. d.]. OpenVPN - OpenVPN is an open source VPN daemon. https://openvpn.net/.

[41] OSMExpress [n. d.]. OSMExpress - Fast database file format for OpenStreetMap. https://github.com/bdon/OSMExpress.

[42] Dong Qiu, Bixin Li, and Hareton Leung. 2016. Understanding the API usage in Java. *Information and software technology* 73 (2016), 81–100.

[43] recorder [n. d.]. Recorder - Store and access data published by OwnTracks apps. https://owntracks.org/booklet/clients/recorder/.

[44] Igor Schittekat, Mehrdad Abdi, and Serge Demeyer. 2022. Can We Increase the Test-coverage in Libraries using Dependent Projects' Test-suites?. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 294–298.

[45] SDL Website [n. d.]. Simple Directmedia Layer (SDL). https://www.libsdl.org/.

[46] Kostya Serebryany. 2017. OSS-Fuzz – Google's continuous fuzzing service for open source software. In *Proc. of the 26th USENIX Security Symposium (USENIX Security'16)* (Vancouver, BC, Canada). Invited talk.

[47] SFML [n. d.]. Simple and Fast Multimedia Library - SFML. https://www.sfml-dev.org/index.php.

[48] SQLite Website [n. d.]. SQLite. https://www.sqlite.org/.

[49] Stack Overflow Guru Interface Q1 [n. d.]. Stack Overflow Guru Interface Q1. https://stackoverflow.com/questions/39938409/how-to-use-fftw-guru-interface.

[50] Stack Overflow Guru Interface Q2 [n. d.]. Stack Overflow Guru Interface Q2. https://stackoverflow.com/questions/58513592/confusion-about-fftw3-guru-interface-3-simultaneous-complex-ffts.

[51] Stack Overflow Guru Interface Q3 [n. d.]. Stack Overflow Guru Interface Q3. https://stackoverflow.com/questions/68198005/fftw-guru-interface.

[52] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards Understanding Third-Party Library Dependency in C/C++ Ecosystem. In *Proc. of the 37th IEEE International Conference on Automated Software Engineering, (ASE'22)*.

[53] Tree-sitter [n. d.]. Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

[54] uacme [n. d.]. ACMEv2 client written in plain C with minimal dependencies. https://github.com/ndilieto/uacme.

[55] Jamal Uddin, Rozaida Ghazali, Mustafa Mat Deris, Rashid Naseem, and Habib Shah. 2017. A survey on bug prioritization. *Artificial Intelligence Review* 47 (2017), 145–180.

[56] UFOAI Website [n. d.]. UFO: Alien Invasion. https://ufoai.org/wiki/News.

[57] Vorbis Website [n. d.]. Ogg Vorbis. https://www.xiph.org/vorbis/.

[58] Weggli [n. d.]. Weggli. https://github.com/weggli-rs/weggli/.

[59] Weggli issue [n. d.]. Weggli issue. https://github.com/weggli-rs/weggli/issues/91.

[60] xxHash Website [n. d.]. xxHash - Extremely fast non-cryptographic hash algorithm. https://xxhash.com/.

[61] Ahmed Zaki and Cristian Cadar. 2024. Artifact of Understanding API Usage and Testing: An Empirical Study of C Libraries. https://doi.org/10.5281/zenodo.13862392. Zenodo.

[62] Ahmed Zaki, Arindam Sharma, and Cristian Cadar. 2025. Generating and Contributing Test Cases for C Libraries from Client Code: A Case Study. In *Proc. of the 32nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'25)* (Montreal, Canada).

[63] Min Zhang, Nathan Baddoo, Paul Wernick, and Tracy Hall. 2011. Prioritising refactoring using code bad smells. In *2011 IEEE fourth international conference on software testing, verification and validation workshops*. IEEE, 458–464.

[64] Hao Zhong and Hong Mei. 2017. An Empirical Study on API Usages. *IEEE Transactions on Software Engineering* 45, 4 (2017), 319–334.

[65] Zip Website [n. d.]. Zip. https://github.com/kuba--/zip.

[66] Zstandard Website [n. d.]. Zstandard - Real-time data compression algorithm. https://facebook.github.io/zstd/.