Generating and Contributing Test Cases for C Libraries from Client Code: A Case Study

Ahmed Zaki, Arindam Sharma, Cristian Cadar
Imperial College London
{ahmed.zaki, arindam.sharma, c.cadar}@imperial.ac.uk

***Abstract*—Software libraries are at the core of software development, and any bugs can affect a potentially large number of present and future client applications. Therefore, thorough testing of libraries is of key importance. Unfortunately, writing library test cases is often difficult, requiring awareness of complex data structures and preconditions.**

We report our experience implementing APISLICER, a technique (and tool) which starting from the client of a library under test, extracts self-contained library test cases that can be used to enhance the library's test suite. Such test cases provide a key benefit: they represent real-world usage scenarios of the APIs exported by the target library, which may not have been envisioned by the library developers.

We have conducted a case study in which we have applied APISLICER on seven mature libraries, with a total of twelve clients. Our experience highlights that while APISLICER has successfully extracted compilable test cases for all these libraries, library developers are oftentimes cautious about accepting the test cases. Out of seven libraries, three (LIBUNISTRING, AMPLGSL, and GSL) accepted our contributions. We report on the reaction of the developers to our contributions and more generally on the opportunities and challenges facing this approach.

***Index Terms*—Test case contributions, software libraries, API testing, test generation**

I. INTRODUCTION

While comprehensively testing any type of software is difficult, testing library code is particularly so, because it involves writing complicated test harnesses. In particular, writing test cases for libraries often requires the creation of complex data structures, invoking APIs in a certain order, and satisfying various additional preconditions. What is worse, such test cases might not reflect the way real clients of the library use the library's APIs.

Developers of software libraries test their software by creating test suites that represent the way *they* envision their APIs to be used. A client's use of a library does not always adhere to those usage scenarios. For example, a library developer might not foresee a certain permutation of a set of APIs that a client invokes. By writing test cases based on clients usage scenarios, library developers can exercise the library's APIs in ways that represent how the library is used in the real world.

In this paper we report on our experience generating and contributing test cases to seven libraries, using a total of twelve clients. Our objective is to generate test cases that not only represent the sequence of statements clients use to invoke an API, but also the parameter values the clients use to call the API. Our hypothesis is that such an approach can help library developers in various ways including, but not limited to, increasing test coverage of the library and having realistic test cases that mimic how clients use their API.

We developed an automated technique and tool, APISLICER, for library test case generation, which we complement with manual steps in order to integrate the test cases into a library's test suite. APISLICER uses clients of a target library to generate test cases that are independent of the clients they are extracted from. Our technique starts by identifying the client functions which call a given library's APIs. It then uses a combination of data- and control-flow analysis to prune from the client code statements which are not needed to invoke the library APIs. The remaining statements are then used to generate a test case which can be compiled independently of the client. Using the client test suite, we then extract the values required to initialise the parameters of the APIs. The end result is a test case that represents very closely how the client of a library uses its API. Finally, to validate the utility of the generated test cases, we integrate the test cases into the library's test suite and contribute them to the library developers. We report on our end-to-end experience, from test case generation to contribution, and discuss the lessons learned throughout each stage.

Our experience has highlighted some positive and negative insights. On the positive side, we show that our design decisions allow us to generate compilable test cases from clients of a library for the C programming language. These test cases represent how client functions invoke a library's API. We have managed to construct test cases for all library-client pairs, for a total of 176 test cases. We also show that for some of the generated test cases, we are able to increase test coverage in the target library. In fact, we were able to successfully contribute our test cases to three different libraries. This proves our initial hypothesis that library developers can benefit from using their clients for test case generation.

Unfortunately, our experience also highlighted several challenges. Using client test suites to extract the exact values used to initialise parameters of an API might not always be feasible, as clients do not always test the APIs they use from third-party libraries, directly or indirectly [1]. As a result, we could not initialise many of the test cases extracted from the clients. Furthermore, generating human-readable test cases is challenging, as is integrating them into existing test suite infrastructures—we have found these to be important aspects, which have been largely neglected by the research community.

On a non-technical front, the reaction from library developers towards our contributed test cases was in many cases underwhelming. Since our test cases do not uncover bugs in the library, it was hard to convince some developers of the utility of the test cases. In cases where our test cases did not improve coverage, it was challenging to convince developers of the benefit of client-extracted test cases.

Lastly, we designed our tool, APISLICER, to work on the source code level, in order to construct test cases that look as similar as possible to how the clients use the library. This imposes several constraints: the tests need to be created in the same programming language as the one used for the library APIs (as opposed to say, at the LLVM bitcode or binary level) and the tests have to be self-contained, readable, with good oracles, and integrated into the existing library test suites. This is particularly challenging for C, as the tools are not as mature as in the case of the intermediate compiler representation level (e.g., LLVM IR) and one needs to deal with numerous language constructs, including macros, complex expressions, variadic functions and various types of loops.

To summarise, our paper makes the following contributions:

1) An automated approach for extracting source-code, self-contained library test cases starting from client code.
2) An implementation of our approach into APISLICER, a tool targeting the C programming language. We complement APISLICER with manual steps to integrate the generated test cases into the libraries test suites and contribute them to the library developers.
3) Our experience and lessons learned applying APISLICER to seven libraries and twelve clients. While we were able to contribute test cases to three libraries, we discuss the challenges encountered in contributing to the other libraries and potential research directions that could address them.
4) An artifact containing our implementation and benchmarks for reproducibility.

II. OVERVIEW

APISLICER aims to generate *small* and *self-contained* source-code test cases that are useful to library developers and represent how clients of the library use the API. In particular, this means that the test cases should only contain statements that are needed to call the library APIs and only depend on definitions in the target library and system header files. To keep extracted test cases small, APISLICER operates at the level of individual client functions. Given a client function that invokes one or more library APIs, its job is to remove irrelevant statements, include only the necessary function and data structure definitions, and initialise all the relevant parameters of the client function. More precisely, using a combination of data- and control-flow analysis, APISLICER computes dependency information between the statements of the client function. This information is used to slice the client function, keeping only the statements that are relevant for invoking the library APIs. Starting from the APIs of a target library (in the form of a C header file) and a client application that uses that library, APISLICER produces as output a set of library test cases extracted from the client code (*test case generation*). Our techniques are inspired by research in fuzz driver generation [2]–[4], but our goal is different. We aim to generate and contribute self-contained, concrete test cases which is arguably more challenging to generate than fuzz drivers due to the fact that not only do these test cases have to capture all necessary dependencies, but they also need to be human-readable, realistic, and contain test inputs that represent how the clients use the API.

To obtain inputs needed to execute the extracted test case (*test case initialisation*), we run the test suite of the client to record values with which the original client function is run. We instrument the client functions of interest to collect these values. Previous research in unit test carving used system executions to gather values to initialise unit tests [5]–[8]. However, while work in this area generated test cases that use real-world values extracted from execution of the target, they did not use the source code of the caller. BASILISK [6] operates on the LLVM-IR level of a target application and generates unit tests in LLVM-IR, making them not human-readable and as a result hard to integrate into a test suite. Our objective is to generate readable test cases that mimic how a client uses a library and integrate those test cases in the test suite of the library.

Finally, we integrate the test cases into the target library's test suite (*test case integration*) and gather feedback from the library developers (*developer feedback*). Integrating a test case into the test suite of a library is non-trivial to automate. The process involves understanding the test suite design in order to identify how and where to add the new test case. Each library has its own coding guidelines that need to be adhered to when contributing to it. As a result, contributing a test case to a library involves taking feedback from the library developers and incorporating it in the test case. Not much attention has been given in the literature to the challenges involved in adding test cases to a test suite. We believe our experience can help show opportunities for future work.

Figure 1 shows the main components of APISLICER, which are discussed in detail in §III.

III. APISLICER

In this section we discuss the different stages of our approach including the design of APISLICER. As a running example, we show how APISLICER generates a test case for the LIBICONV [9] text encoding/decoding library, using the GNUPG [10] client. Figure 2 shows one of the client functions selected from GNUPG, while Figure 3 shows the generated test case. For the sake of readability, we have removed some preprocessing code and comments, and abbreviated some of the conditional checks while preserving the structure of the original function.

A. Test Case Generation

Since our goal is to contribute tests to library test suites, we need to conduct our analyses at the source code level. This is particularly challenging for C, as we need to deal with numerous language constructs, including macros, complex

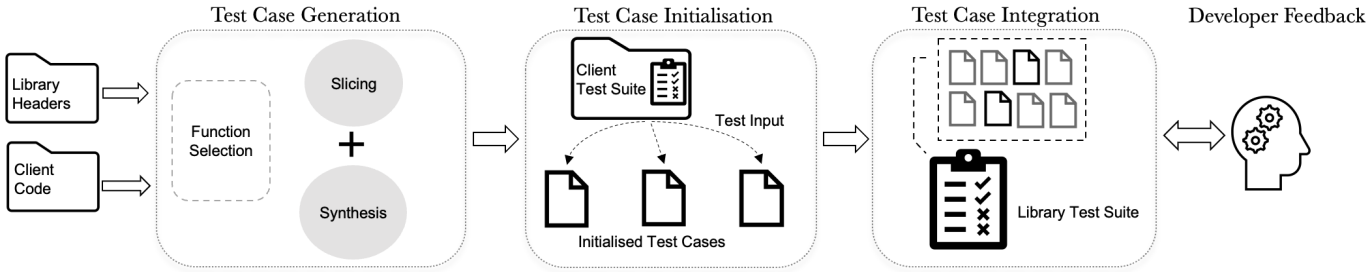

Fig. 1. The main components of APISLICER.

expressions and various types of loops. We use CLANG LIBTOOLING [11] to analyse the abstract syntax tree (AST) of the client functions. To simplify our analysis, we designed a number of semantics-preserving normalisations, which make the resultant program's AST easier to deal with in the subsequent steps. Prior to processing a translation unit, we transform *for loops* into *while loops* and split nested function calls into separate statements. We also simplify *if-statement* conditions by replacing them with temporary variables and convert *switch-case* statements into blocks of simplified *if-else* statements. The normalised translation unit is then processed in multiple stages by APISLICER.

Function Selection For each translation unit, we identify functions that invoke APIs from the target library by checking if the APIs are defined in the library header files. Statements that call the API of the target library in those functions are seed statements. We filter these functions using two strategies:

- **Primitive-client**: The *primitive-client* strategy looks for client functions that either take no parameters at all or only parameters of primitive types or compounds of primitive types such as arrays or pointers to primitive types. Void pointers are excluded from the criteria since it is too complex to infer the pointee type and initialise it accordingly. Our rationale behind this mode of selection is to try and capture the initialisation of any variables of such types that could be passed, directly or indirectly, as parameters to the API calls in the function. This allows us to limit our analysis to be intra-procedural.
- **Primitive-API**: In this mode, we select a client function if it calls a library API with primitive types as parameters and/or compounds of primitive types. Since our seed statements are the API call expressions, then our backwards slicing will capture the correct initialisation of the parameters. The rationale for the *primitive-API* was the observation that while the *primitive-client* strategy works in many cases, we found that in some instances it discards functions that take compound types of user-defined types or pointers to user-defined types which are not required for the correct invocation of the API call, since the call would not have any data- or control-flow dependency on those parameters.

In our example test case shown in Figure 2, both modes would select the function *set_native_charset* for analysis since the parameters of the function are pointers to primitive types and so are the parameters to the library API calls.

Slicing Once we have identified the client functions of interest, we slice them with respect to the called APIs. Conceptually, our approach is similar to slicing using a Program Dependence Graph (PDG) [12]. Constructing the PDG for each function of interest in a language like C is difficult and inefficient. Instead, we choose to perform a data- and control-flow analysis on demand as we identify statements of interest.

- **Data-Flow Analysis**: Prior to slicing, we perform a reaching definitions analysis on the selected function. For our example in Figure 2, the API calls for LIBICONV on lines 42 (*iconv_open*), 49 (*iconv_close*), 50 (*iconv_open*) and 57 (*iconv_close*) would be seed statements. Note that the call expressions on lines 44 and 52 are to GNUPG functions and not LIBICONV APIs, so they are not selected. We perform a data-flow analysis which consists of a use-def analysis [13] for each argument in the call expression. We combine this with information from our reaching definitions analysis to select all statements that define variables required for the correct invocation of the call expression.
- **Control-Flow Anlaysis**: For each of the collected statements from the data-flow analysis, we identify any control statements that dominate any of those statements. Taking a closer look at our example, we can see that the *if* statement on line 31 dominates the block where our seed statement is (line 50). We include this control-flow statement and analyse the condition inside it for usages and definitions. We identify that *newset* is used in this condition and consequently find the reaching definitions for that usage using our data-flow analysis. As a result, we would include the definitions at lines 26 and 28. For both those definitions, our control dependency analysis would identify the *if* statement on line 25 and the *if* statement on line 27 as dominating statements for those definitions. Our algorithm would also identify that the *if* statement on line 3 dominates the definition selected on line 22 and include it.

The result of the slicing process is a set of statements that are then passed to the synthesiser to create a test function.

Synthesis The synthesiser first creates a function with no parameters, with the same name as the client function, but

```
1  int set_native_charset (const char *newset){
2    const char *full_newset;
3    if (!newset)  {
4      static char codepage[30];
5      unsigned int cpno;
6      const char *aliases;
7      cpno = GetConsoleOutputCP ();
8      if (!cpno) cpno = GetACP ();
9      sprintf (codepage, "CP%u", cpno );
10     newset = codepage;
11     for (aliases = ("CP936" "\0" "GBK" "\0"...);
       *aliases; aliases += strlen (aliases) + 1,
       aliases += strlen (aliases) + 1){
12         if (!strcmp (codepage, aliases) || (*
       aliases == '*' && !aliases[1])){
13           newset = aliases + strlen (aliases) + 1
       ;
14           break;
15         }
16     }
22     newset = nl_langinfo (CODESET);
23   }
24   full_newset = newset;
25   if (strlen (newset) > 3
        && !ascii_memcasecmp (newset, "iso", 3))  {
26         newset += 3;
27         if (*newset == '-' || *newset == '_')  {
28             newset++;
29         }
30   }
31   if (!*newset
       || !ascii_strcasecmp (newset, "8859-1")
       || ...) {
37         active_charset_name = "iso-8859-1";
38         no_translation = 0;
39         use_iconv = 0;
40   } else{
41         iconv_t cd;
42         cd = iconv_open (full_newset, "utf-8");
43         if (cd == (iconv_t)-1){
44           handle_iconv_error (full_newset,
        "utf-8", 0);
47           return -1;
48         }
49         iconv_close (cd);
50         cd = iconv_open ("utf-8", full_newset);
51       if (cd == (iconv_t)-1){
52           handle_iconv_error ("utf-8",
       full_newset,0);
55           return -1;
56       }
57         iconv_close (cd);
58         active_charset_name = full_newset;
59         no_translation = 0;
60         use_iconv = 1;
61   }
62   return 0;
63 }
```

Fig. 2. GNUPG function with sliced statements highlighted: seed statements in green and other sliced statements in blue.

```
1  int Test_set_native_charset() {
2    // begin function parameters
3    const char *newset = NULL;
4    // end function parameters
5    const char *full_newset;
6    if (!newset) {
7      newset = nl_langinfo(CODESET);
8    }
9    full_newset = newset;
10   if (strlen(newset) > 3 && !ascii_memcasecmp(
       newset, "iso", 3)) {
11     newset += 3;
12     if (*newset == '-' || *newset == '_') {
       newset++; }
13   }
14
15   if (!*newset || !ascii_strcasecmp(newset, "
       8859-1") ||
16       !ascii_strcasecmp(newset, "646") || !
       ascii_strcasecmp(newset, "ASCII") ||
17       !ascii_strcasecmp(newset, "ANSI_X3.4-1968")
       ) {
18   } else {
19     libiconv_t cd;
20     cd = libiconv_open(full_newset, "utf-8");
21
22     if (cd == (libiconv_t)-1) {
23       return -1;
24     }
25     libiconv_close(cd);
26     cd = libiconv_open("utf-8", full_newset);
27
28     if (cd == (libiconv_t)-1) {
29       return -1;
30     }
31     libiconv_close(cd);
32   }
33 }
```

Fig. 3. Test function auto-generated by APISLICER from the code in Figure 2.

prefixed with *Test_*. For example, the client function generated by the synthesiser for our running example is called *Test_set_native_charset()*, as shown in Figure 3.

The synthesiser iterates over each collected statement, checking definitions, function calls and types used. For each statement, the synthesiser ensures that a declaration is available for the variables used by the statement. For example, the declaration of variable *full_newset* on line 2 in Figure 2 is added during this iteration by copying it from the original function. If a declaration is not collected by the slicer, the synthesiser checks if the variable is declared as part of the function signature. In our running example, the synthesiser identifies that *newset* is declared as a parameter to the client function and includes it as a separate statement at the beginning of the synthesized function. Finally, the synthesiser checks the types used in each statement and, if any, calls to functions. If any of the types are user-defined and their definition is in the same source file as the collected statement, the definition is added to the test case. Similarly, if any of the calls are made to functions in the same source file, then the function definition is added. The synthesiser does not try to fetch types and functions defined outside the function source file, as we found that this could lead to recursive fetching which would generate very

large test cases. Instead, we keep all the header files from the translation unit in the generated test case and allow the developer to decide which definitions to fetch manually.

The end result of the test case generation step is a test case that compiles using the compile commands extracted from the compilation database of the client. We perform an automated *de-normalisation* step at the end to reverse as much as possible the normalisations introduced prior to processing the translation unit.

Implementation notes. In our implementation, we used CLANG LIBTOOLING [11] (with CLANG-11), which provides an interface to analyse the program at the abstract syntax tree (AST) level. In particular, we used the recursive AST visitor interface and implemented our algorithms on top of it.

Since CLANG LIBTOOLING requires the same compilation flags that were used for compiling the client code in order to construct a valid AST, we observed that typically for clients using *cmake* build systems, this information can be found in the *compile_commands.json* database. If no compilation database is found, it can be generated using BEAR [14].

We used INCLUDE-WHAT-YOU-USE [15], a tool that trims unnecessary headers in our generated test cases. The tool has some issues and limitations, so we first try to compile the test case with INCLUDE-WHAT-YOU-USE and if it fails to compile, we try again without it.

B. Test Case Initialisation

After the test function is created, we need to initialise the parameters and global state used by the test case. We obtain those values from the client itself, by observing how the original function is called in the client test suite. We identify which of the parameters of the client function and global variables are required by our test case, and add instrumentation in the client function to record their values to a file. We then run the client test suite, obtain those values, and then modify the test case to use them. If the client test suite invokes that function multiple times, we create multiple test cases.

C. Test Case Integration

With the test case initialised, the next step is to integrate it into the library test suite. This stage is non-trivial to automate as it involves first learning how and where to integrate the test case into the test suite. This often includes understanding the test suite structure and the associated configuration files and scripts.

The test case itself might also need to be adapted to use the kind of oracles used by other tests, or to merge it with other test cases that exercise similar APIs. In addition, there could also be formatting guidelines that the test case needs to follow.

D. Developer Feedback

Finally, we submit the updated test suite to the library developers, typically as a pull request, but sometimes by email, according to the library's contribution model. Depending on the feedback, we might need to make further updates to our test case. We use the developer feedback to better understand the needs of developers and the challenges we would need to overcome to contribute such test cases (see §IV).

E. Design decisions

Test case generation. We made a few key design decisions when performing our slicing stage. Our analysis supports *structs* and is field-sensitive. This allows us to fetch definitions for fields of user-defined structures localized in the client function. We also support arrays and the slicing is index-insensitive. We do not perform an alias analysis which means that we could lose some accuracy when it comes to value redefinitions.

Our analysis is intra-procedural so when we include callees of a caller we do not perform any slicing on them. We would still include the callees during the synthesis stage as part of the generated test case. Our function selection strategies help alleviate the absence of an alias analysis due to our focus on primitive types.

Given our goal of generating test cases that represent how clients use library APIs, we aimed to preserve the structure of the original client function code. This allows the developers to use these test cases as representative examples of how their library is being used in the field.

Test case initialisation and integration. Instead of trying to automate everything, we have taken the approach of requiring some development involvement.

The instrumentation done for test case initialisation is currently manual, but we believe it can be automated.

In terms of test case integration, the need for developer involvement is partly because fully automating the process is challenging due to the variety of test suite designs used by the library developers (see §III-C), but also because we see APISLICER as a developer tool, which would be used by library developers, who have the necessary domain knowledge of where and how to integrate the test cases.

IV. CASE STUDY

In this section, we report on our experience applying APISLICER on several mature libraries, and creating and contributing test cases to these libraries.

A. Benchmarks

We applied APISLICER to seven mature libraries, using twelve clients in total. Both libraries and clients are written in C. To pick our libraries, we used as a starting point Awesome-C [16], which lists popular C frameworks and applications. We focused on libraries that were actively developed and easy to build and test on a Linux system. For each library, we searched for obvious clients by using GITHUB's search engine. We picked clients that had a permissive license to allow us to contribute source code extracted from the client to the library test suite. For both libraries and clients, we either use a recent released version or the latest version from their development repositories at the time we started to evaluate that library.

An overview of these libraries and clients is provided in Table I. For each library, we report its size in lines of code

TABLE I
LIBRARIES AND CLIENTS USED, TOGETHER WITH THE LINE COVERAGE ACHIEVED BY THE LIBRARY TEST SUITES, AND THE ADDITIONAL LINES COVERED BY EACH CLIENT IN THE LIBRARY.

Library	LoC	Line Coverage	Client	Extra lib. lines
AMPLGSL	71,547	55,941 (78.2%)	APOPHENIA	40
			HSTAXE	-
GSL	66,130	51,892 (78.5%)	APOPHENIA	43
			HSTAXE	-
LIBICONV	8,895	6,865 (77.2%)	MUTT	-
			GNUSASL	0
			GETTEXT	4
			GNUPG	11
LIBSODIUM	10,289	8,895 (86.5%)	DOVECOT	0
			BLOBCRYPT	-
LIBTASN1	4,322	3,469 (80.3%)	GNUTLS	44
			P11KIT	7
			SWTPM	46
LIBUNISTRING	8,289	5,725 (69.0%)	GNUTLS	1
LIBXML2	117,351	79,946 (68.1%)	XMLSEC	329

(LoC), and the line coverage achieved by its test suite. These numbers are obtained via the GCOV/LCOV tools and thus depend on how the applications are built on our machine (due to conditional compilation, some code is not included in all configurations). For each client, which has a test suite, we also measured extra line coverage (if any) achieved in the target library by running the client's test suite. Some clients did not have test suites such as BLOBCRYPT and HSTAXE. Furthermore, our measurements exclude any test files (on a best-effort basis).

AMPLGSL [17] (commit *d77d742*) and **GSL** [18] (v2.7). GSL is the GNU Scientific Library, a numerical library which provides a wide range of mathematical routines. AMPLGSL enhances GSL with AMPL bindings, which speed up certain numerical computations. We tested two common clients of GSL and AMPLGSL: APOPHENIA [19], from its *pkg* branch (commit *35e1093*) and HSTAXE (commit *0316860*).

LIBICONV [9] (commit *8844744*) is a text encoding/decoding library, part of the GNU project. We tested four clients for LIBICONV: MUTT [20] 2.2.11, GNUSASL [21] 2.2.0, GNUPG [10] 2.4.2 and GETTEXT [22] 0.22.

LIBSODIUM [23] (v1.0.18) is a cryptographic library which includes support for encryption, decryption, signatures, and password hashing. We selected two clients that use LIBSODIUM: BLOBCRYPT [24] (commit *726daa4*) and DOVECOT [25] (commit *a753734*).

LIBTASN1 [26] (commit *2f19433*) is a portable ASN.1 library. We used three clients of LIBTASN1: GNUTLS [27] *3.6.16*, P11KIT [28] 0.24.1 and SWTPM [29] (commit *4b008b9*).

LIBUNISTRING [30] (commit *cb3f069*) is a text processing library, part of the GNU project. We used one client, GNUTLS *3.6.16*.

TABLE II
CLIENT FUNCTIONS SELECTED IN EACH MODE AND TOTAL GENERATED FUNCTIONS FROM APISLICER.

Benchmark	Client	Candidates	P-Client	P-API	Total	Generated
AMPLGSL/GSL	APOPHENIA	69	15	41	41	28
	HSTAXE	247	28	108	116	75
LIBICONV	GETTEXT	7	4	3	4	2
	GNUPG	6	4	4	4	3
	GNUSASL	3	1	1	1	1
	MUTT	10	6	1	6	4
LIBSODIUM	BLOBCRYPT	15	6	6	6	6
	DOVECOT	4	1	4	4	4
LIBTASN1	GNUTLS	146	5	9	13	11
	P11KIT	39	13	13	16	14
	SWTPM	10	2	1	3	3
LIBUNISTRING	GNUTLS	4	1	4	4	4
LIBXML2	XMLSEC	183	18	11	24	21

LIBXML2 [31] (commit *8844744*) is an XML library originally developed for the GNOME Project. We used one client for LIBXML2: XMLSEC [32] 1.2.37.

B. Test Case Generation Results

As discussed in §III-A, APISLICER first selects a subset of the client functions that use library APIs. Table II shows, for each library-client pair, the results of this selection process.

The *Candidates* column reports the number of client functions with library API calls. As can be seen, some clients, such as XMLSEC for LIBXML2 and GNUTLS for LIBTASN1 have many such calls, while others, such as GNUSASL for LIBICONV and GNUTLS for LIBUNISTRING, only a few.

The next two columns, *P-Client* and *P-API* report the number of client functions selected by each mode; *primitive-client* and *primitive-API* respectively (see §III-A). Overall, the total functions selected vary between a single one for GNUSASL/LIBICONV and 116 for HSTAXE/GSL. The next column, *Total* reports the total number of functions selected, i.e. the union of the functions selected in each mode. If a client function has primitive types *and* calls library APIs with primitive types, it will be selected by both modes, so the total is typically much smaller then the sum of the functions selected by both modes. This number represents the total number of test cases generated including test cases that fail to compile.

The *Generated* column reports the number of compilable test cases automatically generated by our approach. These are in most cases close to the *Total* column, meaning that for most client functions selected, APISLICER is able to generate a test function that compiles, but without its parameters initalised yet. We saw significant losses in the case of HSTAXE and APOPHENIA. The reason for which we could not generate compilable test cases in those instances had to do with APISLICER's limitations around function-like macros and variadic functions.

One limitation of the automatically generated test cases was often readability. This is partly due to limitations of the denormalisation process, which may not be able to recover the syntax of the original code, particularly when macros and include directives are involved. We have typically spent a few

TABLE III
GENERATED AND CONTRIBUTED TEST CASES. FOR CLIENTS THAT DID NOT HAVE A TEST SUITE WE SHOW '-' INSTEAD OF 0 FOR THE TESTS INITIALISED.

Library	Client	Generated	Tests Initialised	Contributed	Added Cov.
AMPLGSL/ GSL	APOPHENIA	28	1	0	0
	HSTAXE	75	-	0	
LIBICONV	GETTEXT	2	1	0	0
	GNUPG	3	1	0	
	GNUSASL	1	0	0	
	MUTT	4	-	0	
LIBSODIUM	BLOBCRYPT	6	1	1	112
	DOVECOT	4	2	2	
LIBTASN1	GNUTLS	11	1	1	12
	P11KIT	14	1	1	
	SWTPM	3	1	1	
LIBUNISTRING	GNUTLS	4	4	1	1
LIBXML2	XMLSEC	21	3	3	10

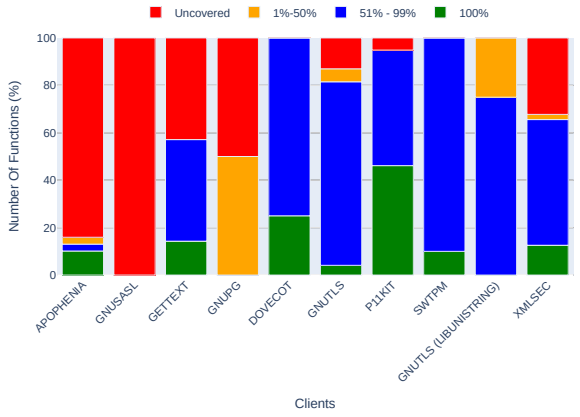

Fig. 4. The percentage of functions from the total number of candidate functions per client split by coverage achieved by each client test suite.

minutes to make the test case more readable before contributing it to the developers.

C. Test Case Initialisation

After generating compilable test cases, we need to initialise them by extracting the values of interest from the client test suites. Some clients did not have test suites, such as MUTT, HSTAXE and BLOBCRYPT. Where available, we used built example tools, such as in the case of BLOBCRYPT, to extract concrete inputs.

In Table III we list the cases we were able to initialise by running the client test suites along with the added coverage the combined test cases were able to achieve. The column *Tests Initialised* shows the number of test cases for which all parameters could be initialised. [1]

[1]A small number of test cases only invoked trivial APIs such as simple logging functions, we did not attempt to initalise these test cases.

The reason for which not all generated tests could be initialised is due to the limited coverage of the client's test suite. Figure 4 shows the distribution of coverage by the client test suites for the candidate functions (*Candidates*) shown in Table II. As shown, in the case of APOPHENIA, only 7 out of 69 candidate functions were fully covered, with the majority of the candidates unreachable by the client test suite. In the case of GNUSASL, all of the 3 functions were unreachable. Note that functions with coverage within 51%– 99% do not guarantee that the library API will be exercised, as the API call could be in the uncovered lines.

For each case we were able to initalise, we investigated if extra coverage could be achieved by the test case in the library. The last column in Table III shows the extra coverage achieved by these tests on top of the library test suite. We managed to increase the library coverage in all cases except for LIBICONV and AMPLGSL/GSL. The extra coverage varies from only 1 LoC in LIBUNISTRING to 112 in LIBSODIUM.

D. Test Case Integration

In our experience, we found libraries to have a wide variety of designs for their test suites. To integrate a test case, we first had to understand how the current tests in the library's test suite are run, and then had to adapt the test case in a format that resembles other test cases already used by the library.

For example, the test case in Figure 5 is already a usable test case that increases coverage in LIBUNISTRING. To integrate this test case in the test suite of LIBUNISTRING, we had to adapt it to the version shown in Figure 6. We found that there was already a test case in LIBUNISTRING that had three tests which resembled our test case. We added our new test case as a fourth assertion. We also trimmed our test function as we realized that to exercise the added coverage we do not need the whole of Figure 5. As a result, we removed the function *is_allowed_exception* and the *for* loop in *Test_check_for_valid_freeformclass*. This trimming could be automated by leveraging existing reducers such as C-Reduce [33] and Perses [34]. The process of integrating this test case into the test suite of LIBUNISTRING took us approximately two hours. Of course, we expect LIBUNISTRING developers to be able to integrate the test case into their test suite much quicker, given their familiarity with the code.

E. Developer Feedback

The *Tests Contributed* column of Table III shows the number of test cases that we contributed to the libraries' test suites. After initial feedback from the LIBXML2 library developers who were reluctant to accept a test case based solely on the fact that it represented usage of the library in the real world [35], we decided not to contribute test cases for libraries where we have not managed to increase line coverage. We started by making a single contribution per library, but we sometimes grouped a few test cases together. We contributed between 1 and 3 test cases to six libraries—LIBSODIUM, LIBTASN1, LIBUNISTRING, and LIBXML2 (see Table III), together with AMPLGSL and GSL, for which we have followed a slightly

```c
#include <stddef.h>
#include <stdint.h>
#include <unictype.h>
#include <uninorm.h>

inline static int is_allowed_exception(uint32_t ch)
{
  switch (ch) {
    case 0xB7:
    case 0x0375:
    ...
      return 0; /* disallowed */
    case 0xDF:
    case 0x03C2:
    ...
      return 1; /* allowed */
    default:
      return -1; /* not exception */
  }
}
int Test_check_for_valid_freeformclass() {
  // begin function parameters
  unsigned ucs4_size = 14;
  uint32_t *ucs4 = (uint32_t *)malloc(sizeof(uint32_t)*ucs4_size);
  // end function parameters
  ucs4[0] = 1055;
  ucs4[1] = 1072;
  ucs4[2] = 1088;
  ...
  unsigned i;
  int rc;
  uint32_t tmp[4];
  size_t tmp_size;
  uint32_t *nrm;
  uc_general_category_t cat;
  unsigned is_invalid;
  cat = uc_general_category_or(UC_CATEGORY_Ll, UC_CATEGORY_Lu);
  cat = uc_general_category_or(cat, UC_CATEGORY_Lo);
  cat = uc_general_category_or(cat, UC_CATEGORY_Nd);
  cat = uc_general_category_or(cat, UC_CATEGORY_Lm);
  cat = uc_general_category_or(cat, UC_CATEGORY_Mn);
  ...
  cat = uc_general_category_and_not(cat, UC_CATEGORY_Cc);
  for (int i=0; i<ucs4_size; i++) {
   if (uc_is_property_default_ignorable_code_point(ucs4[i]) ||
        uc_is_property_not_a_character(ucs4[i])) {
      return 0;
    }
    rc = is_allowed_exception(ucs4[i]);
    if (rc == 0 || uc_is_property_join_control(ucs4[i])) {
      return 0;
    }
    if (rc == 1) {
      continue;
    }
   if (uc_is_general_category(ucs4[i], UC_CATEGORY_Zs)) {
      ucs4[i] = 0x20;
    }
    ...
  }
}

int main(){
  Test_check_for_valid_freeformclass();
}
```

Fig. 5. Test case generated from GNUTLS for LIBUNISTRING.

different approach (described below). We next discuss each of our contributions.

LIBUNISTRING contribution. Our contribution to LIBUNISTRING was accepted and integrated in the test suite of the library [36]. The test case contributed covers an extra line in LIBUNISTRING. That extra line exercised behaviour that was not previously tested. The response from the developer in this instance was quick and positive. They initially inquired about the compatibility of the licensing of GNUTLS but accepted the case, with some simplifications, once this was clarified. Our contribution can be seen in Figure 6.

```c
int control_category_check() {
    uc_general_category_t cat;
    cat = uc_general_category_or(UC_CATEGORY_Ll, UC_CATEGORY_Lu);
    cat = uc_general_category_or(cat, UC_CATEGORY_Lo);
    cat = uc_general_category_or(cat, UC_CATEGORY_Nd);
    cat = uc_general_category_or(cat, UC_CATEGORY_Lm);
    cat = uc_general_category_or(cat, UC_CATEGORY_Mn);
    cat = uc_general_category_or(cat, UC_CATEGORY_Mc);
    cat = uc_general_category_or(cat, UC_CATEGORY_Lt);
    cat = uc_general_category_or(cat, UC_CATEGORY_Nl);
    cat = uc_general_category_or(cat, UC_CATEGORY_No);
    cat = uc_general_category_or(cat, UC_CATEGORY_Me);
    cat = uc_general_category_or(cat, UC_CATEGORY_Sm);
    cat = uc_general_category_or(cat, UC_CATEGORY_Sc);
    cat = uc_general_category_or(cat, UC_CATEGORY_So);
    cat = uc_general_category_or(cat, UC_CATEGORY_Sk);
    cat = uc_general_category_or(cat, UC_CATEGORY_Pc);
    cat = uc_general_category_or(cat, UC_CATEGORY_Pd);
    cat = uc_general_category_or(cat, UC_CATEGORY_Ps);
    cat = uc_general_category_or(cat, UC_CATEGORY_Pe);
    cat = uc_general_category_or(cat, UC_CATEGORY_Pi);
    cat = uc_general_category_or(cat, UC_CATEGORY_Pf);
    cat = uc_general_category_or(cat, UC_CATEGORY_Po);
    cat = uc_general_category_or(cat, UC_CATEGORY_Zs);
    cat = uc_general_category_and_not(cat, UC_CATEGORY_Cc);

    return (!uc_is_general_category(0x00, cat));
}

int main (){
    ASSERT (uc_is_general_category ('a', ct));
    ASSERT (!uc_is_general_category ('7', ct));
    ASSERT (uc_is_general_category (0x00B2, ct));
    ASSERT(control_category_check());
    return 0;
}
```

Fig. 6. Final contributed test case to LIBUNISTRING.

LIBSODIUM contribution. In the case of LIBSODIUM we were able to generate and initialise test cases that achieved significant added coverage—112 line of coverage. Integrating the test case in the library's test suite locally achieved the extra coverage, yet when submitting the pull-request to add it the continuous integration (CI) pipeline, it failed to show this added coverage. Despite our test case independently adding an additional 112 lines of coverage, we discovered that these lines cannot be reached when the test is part of the LIBSODIUM's test suite. On further investigation, we identified that this is due to some macro definitions done at the global level, when running the test suite in the CI pipeline. In other words, our understanding is that our test cases would need to be run from a different CI target/configuration in order for those lines to be reached [37], [38].

LIBTASN1 contribution. In the case of LIBTASN1, the maintainers have not merged our contributions due to two separate concerns: one related to licensing ("contributions should be assigned to FSF and under the libtasn1 license") and the other related to adding generated test cases ("you [must] automate the extraction process") [39]. Since our approach currently has a manual component, we cannot address the latter concern; furthermore, adding APISlicer to the regression test framework of LIBTASN1 would be heavyweight.

AMPLGSL and GSL contribution. As discussed before, we envision APISLICER as a tool which would be used by library developers to understand how clients use their library and to extract and then incorporate library test cases from the client codebase.

Both GSL and AMPLGSL have regression test suites that achieve a relatively high coverage, of 78.5% and 78.2% respectively. However, not all modules are equally tested. For instance, INTERPOLATION achieved lower coverage than most other modules—around 52% in both libraries—so we decided to focus on contributing tests to this module.

We noticed that many functions in that module require as a first parameter an interpolation workspace of type *gsl_interp**. This is quite common in many libraries, where such a context object needs to be passed. Furthermore, after examining the regression test suites of AMPLGSL/GSL we noticed that such an object is already available there, initialised with mock values and used across different tests. Therefore, we relaxed our function selection strategy to also select client functions with API calls that take such objects.

Using HSTAXE as a client, we used this strategy to produce a test case that covers an API which was not tested by the regression test suites of GSL and AMPLGSL, namely *gsl_interp_eval_integ*. We integrated this test into the library test suite by following the way a similar API was tested. Both AMPLGSL and GSL developers reacted positively to our contribution. AMPLGSL is hosted on GitHub: we contributed our tests as a pull request, which was accepted by the developers [40]. GSL contributions are done by email, so we contacted one of the GSL developers about the test cases [41] and they asked us to contribute the test cases to the mailing list, which we did. [42].

LIBXML2 contribution. For the LIBXML2 contribution, we had a long discussion with the developers, who gave three arguments against accepting our contribution [35]. One of them, which we understand, concerns the quality of our oracles, which could indeed be improved. The second one was summarised by *"I don't really see the benefit over simply running the xmlsec tests directly."* We strongly disagree with this, as running the client test suites directly is more difficult (requiring library developers to be familiar with many different codebases), more time intensive (with many client tests potentially unrelated to the library), and any failures would be very difficult to debug (as it would require understanding the client code too). The third argument was that *"xmlsec is integrated with OSS-Fuzz and is continuously fuzzed with the latest libxml2 code from the master branch. So your tests offer very little on top of that."* This is an interesting argument, pointing to a bystander effect [43], [44] with respect to software fuzzing, where developers rely on third-party fuzzing services to find bugs in their code. Ironically, XMLSEC has not been fuzzed by OSS-Fuzz for a long time now, due to an integration error [45]. There was also a discussion at the start concerning the extra line coverage achieved by our tests; developers see value in tests that increase line or branch coverage, but it seems more difficult to convince them of the need for additional tests that do not increase this coverage but nevertheless showcase how the library is used in the field.

Overall, we were disappointed to observe that test contributions receive significantly less attention than other types of contributions, such as bug reports, with which the authors have a lot of experience. This is in some way natural, although the lack of thorough testing is the root cause of those bugs in the first place!

V. LESSONS LEARNED

In this section we discuss the lessons learned designing and applying APISLICER to seven libraries and twelve clients.

Test case generation. Our design decisions have enabled us to automatically generate compilable test cases for each library-client pair. The number of generated test cases that compile was in most cases close to the total number of client functions that fit our selection criteria: *primitive-client* and *primitive-API*. This shows that our intra-procedural analysis approach works for these types of functions.

However, for some libraries, the number of generated test cases was significantly lower than the total number of client functions that call library APIs. Such libraries heavily use complex types in their parameters, which clients tend to initialise once and pass to various functions as parameters. To improve the number of generated test cases extracted from the clients, one needs to perform an inter-procedural analysis that is not limited to primitive types to follow the use of these complex types across client functions.

Performing data and control-flow analysis on the source code level, especially for a rich language like C is challenging. Simplifying the analysis is possible by re-writing the source to reduce the number of types of AST nodes to analyse. This comes at the cost of the readability of the generated test cases, which we fixed manually in our case study. Techniques based on large language models (LLMs) might be able to help to improve the readability of the generated test cases [46].

Test input generation. We managed to initialise test cases for all six libraries, and increase test suite coverage for four libraries. However, relying on client test suites for input generation was an important limiting factor for our approach, preventing it from initialising more test cases. Other studies, for Java, have also shown that client test suites do not achieve high coverage of the APIs they use from third-party libraries [1]. One could explore other ways of initialising test cases, such as with default or random values, but there is the risk that those values would not be realistic.

Test case integration. In our experience, integrating a test case into a library's test suite presented several challenges associated with adapting the test case to follow the overall test suite design. While this process is much easier for library developers familiar with the test suite design, it is nevertheless time-consuming. Some elements of this stage could be automated, such as removing statements that do not lead to extra coverage, or duplicate state initialisation APIs that could already be present for a group of test cases in a test suite. While test generation has been an active area of research, the challenges of adding the generated test cases to existing test suites has been overlooked and would benefit from future research.

Test case contribution. Three of the seven libraries accepted our test cases. Despite our best efforts, we often found it difficult to convince developers of the utility of the test cases. While we were able to produce and integrate test cases that represent real-world usage of library APIs, library developers were primarily interested in test cases that increase coverage. Finally, developers also raised queries about licensing, an aspect which is also under active discussion in the context of LLM code contributions [47]. However, unlike LLM code generation, our approach can provide clear information about the origin of the code and its license, which developers can use to decide on license compatibility.

VI. Related Work

We are not aware of prior work whose main goal is to contribute test cases to library developers starting from client code. Gross et al. [48] generate test cases to reproduce bugs in `Coq`, an interactive theorem prover, based on breaking changes identified from CI failures. APISlicer generates test cases for APIs irrespective of whether they reproduce bugs, and focuses on C libraries.

Our approach is closely related to prior work on fuzz driver generation, API usage example generation, test case generation and code contributions.

Fuzz driver generation. Approaches for fuzz driver generation [2]–[4], [49]–[52] and library test generation share many similarities, due to the overlapping requirements for fuzz drivers and library test cases. Despite this overlap, the end goals for the two domains are significantly different. Generating and contributing to developers self-contained, concrete test cases has considerably more challenges than generating fuzz drivers due to the fact that not only do these test cases have to capture all necessary dependencies, but they also need to be human-readable, realistic, and developer-friendly, as we have noted in our various interactions with developers.

FUDGE [2] is a system that generates fuzz drivers for a library starting from its clients. While both APISlicer and FUDGE use slicing, our technique diverges from FUDGE in significant ways. In particular, our slicing uses a reaching definitions and use-def analysis and can collect client types, while FUDGE uses a less precise usage-only analysis and is limited to types defined in the target library.

FuzzGen [3] carries out the analyses at the LLVM IR level, the results of which are then lifted to source level. This process results in fuzz drivers that are significantly different from the original source code that was analysed. Moreover, the generated fuzz drivers are not easy to read and hence would be difficult to maintain due to nature of the generated code.

AoT [4] tries to extract code from a software under test in order to be able to test it independently. AoT does not perform any slicing on the extracted code and employs symbolic execution and fuzzing to generate the values to be used in the code instead of extracting concrete realistic values from existing test suites. Nevertheless, AoT could be used in conjunction with APISlicer to first extract the desired client function as a separate compilable program, which could simplify the analyses performed by APISlicer.

API usage example generation. Work on extracting API usage examples [53]–[58] aims to gain insights into how APIs are used by extracting illustrative code snippets from client codebases. However, a key difference is that these code snippets do not need to be compilable in a standalone fashion, nor contributed to a library test suite.

Test case generation. Instead of extracting test cases from clients, one could generate them from scratch. For example, Randoop [59] and EvoSuite [60] are popular feedback-directed random test generators for Java. However, such randomly constructed test cases lack the realism of those extracted from clients, as they might not reflect how the library is used in the field. Similarly, other approaches leverage the execution of the software under test to generate unit tests [5]–[8]. These approaches involve more heavyweight dynamic analysis (while APISlicer only needs to execute the client test suite), and do not use the source code of the client to generate the test cases.

JUnitTestGen [61] makes use of disassembled APKs as a source for test cases for Android APIs. While there is some overlap with the APISlicer methodology, this work makes use of an inter-procedural analysis at the level of Java IR (called Jimple). Since their aim is to test for compatibility issues by running those test cases, there is no requirement for using source code-level analysis, since those test cases are not going to be contributed to library test suites. This further highlights our previously mentioned point about the additional constraints imposed by the requirement of generating developer-friendly test cases.

MSeqGen [62] identifies classes and interfaces in an application under test, and then analyses clients of those classes to generate invocation sequences that can later be used to improve random and symbolic execution testing of the application. The objective and target language of MSeqGen are both different from APISlicer.

Code Contributions. Previous work has looked at factors that make code contributions effective [63]–[65]; our experience confirms some of these findings, while others are unique to test case contributions. Grano et al. [66] investigated the readability of automatically vs manually generated test cases, but not in the context of contributions to code repositories.

VII. Conclusion

We have proposed APISlicer, an approach for extracting library test cases from client codebases. Such test cases provide a key benefit: they represent real-world usage scenarios of the APIs exported by the target library, which may not have been envisioned by the library developers. We have reported on our experience generating and contributing such test cases to seven mature libraries, using a total of twelve clients. Our case study has revealed important insights in terms of test case generation, integration and contribution, and presented the challenges and opportunities for future research in these areas.

VIII. Data Availability

We make APISlicer and the experimental infrastructure, data, and results available as open source, at [67].

References

[1] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? a case study of java projects," *Journal of Systems and Software*, vol. 183, p. 111097, 2022.

[2] B. Domagoj, F. I. Stefan Bucur, Yaohui Chen, C. L. Tim King, Markus Kusano, L. Szekeres, and W. Wang, "FUDGE: Fuzz driver generation at scale," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, Aug. 2019.

[3] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*, Aug. 2020.

[4] T. Kuchta and B. Zator, "Auto off-target: Enabling thorough and scalable testing for complex software systems," in *Proc. of the 37th IEEE International Conference on Automated Software Engineering, (ASE'22)*, Oct. 2022.

[5] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'06)*, Mar. 2006.

[6] A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *Proc. of the 41th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion'19)*, May 2019.

[7] S. Joshi and A. Orso, "Scarpe: A technique and tool for selective capture and replay of program executions," in *Proc. of the IEEE International Conference on Software Maintenance (ICSM'07)*, Oct. 2007.

[8] F. Křikava and J. Vitek, "Tests from traces: automated unit test extraction for R," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'18)*, Jul. 2018.

[9] "Libiconv," https://www.gnu.org/software/libiconv/, 2024.

[10] "The GNU Privacy Guard," https://www.gnupg.org/, 2024.

[11] "Libtooling," https://clang.llvm.org/docs/LibTooling.html, 2023.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison Wesley, 2006.

[14] "Bear-github," https://github.com/rizsotto/Bear, 2023.

[15] "IWYU - include-what-you-use," https://include-what-you-use.org/, 2023.

[16] "Awesome c - a curated list of awesome c frameworks," https://github.com/oz123/awesome-c, 2024.

[17] "AMPL Bindings for the GNU Scientific Library," https://gsl.ampl.com/index.html, 2024.

[18] "GNU Scientific Library," https://www.gnu.org/software/gsl/, 2024.

[19] "Apophenia," http://apophenia.info/, 2024.

[20] "Mutt website," www.mutt.org, 2023.

[21] "GNU SASL Library," https://www.gnu.org/software/gsasl/, 2024.

[22] "GNU gettext," https://www.gnu.org/software/gettext/, 2024.

[23] "Libsodium," https://doc.libsodium.org/, 2024.

[24] "Blobcrypt," https://github.com/jedisct1/blobcrypt, 2024.

[25] "Dovecot The Secure IMAP server," https://www.dovecot.org/, 2024.

[26] "libtasn1," https://www.gnu.org/software/libtasn1/, 2023. [Online]. Available: https://www.gnu.org/software/libtasn1/

[27] "The GnuTLS Transport Layer Security Library," https://www.gnutls.org/, 2024.

[28] "P11-kit," https://p11-glue.github.io/p11-glue/p11-kit.html, 2024.

[29] "SWTPM - Software TPM Emulator," https://github.com/stefanberger/swtpm, 2024.

[30] "GNU Libunistring," https://www.gnu.org/software/libunistring/, 2024.

[31] "Libxml2," https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home, 2024.

[32] "XML Security Library," https://www.aleksey.com/xmlsec/, 2024.

[33] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, Jun. 2012.

[34] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proc. of the 40th International Conference on Software Engineering (ICSE'18)*, May 2018.

[35] "Pull request," https://gitlab.gnome.org/GNOME/libxml2/-/merge_requests/211, 2023.

[36] "Mailing list," https://lists.gnu.org/archive/html/bug-libunistring/2023-10/msg00001.html, 2023.

[37] "Pull request," https://github.com/jedisct1/libsodium/pull/1274, 2023.

[38] "Discussion," https://github.com/jedisct1/libsodium/discussions/1337, 2023.

[39] "Pull request," https://gitlab.com/gnutls/libtasn1/-/merge_requests/89, 2023.

[40] "Pull request," https://github.com/ampl/gsl/pull/70, 2023.

[41] "Email exchange with GSL developer." Email exchange, 2023.

[42] "Mailing list message," https://savannah.gnu.org/bugs/?64549, 2023.

[43] B. Latane and J. M. Darley, "Group inhibition of bystander intervention in emergencies," *Journal of Personality and Social Psychology*, vol. 10, no. 3, pp. 215–212, 1968.

[44] W. E. Deming, *Out of the Crisis*. MIT Press, 2000.

[45] OSS-Fuzz: Fuzz-Introspector, https://introspector.oss-fuzz.com/indexing-overview, 2023.

[46] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani, "Core: Resolving code quality issues using LLMs," in *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'24)*, Jul. 2024.

[47] W. Xu, K. Gao, H. He, and M. Zhou, "A first look at license compliance capability of LLMs in code generation," *arXiv preprint arXiv:2408.02487*, 2024.

[48] J. Gross, T. Zimmermann, M. Poddar-Agrawal, and A. Chlipala, "Automatic test-case reduction in proof assistants: A case study in coq," in *13th International Conference on Interactive Theorem Proving*, 2022.

[49] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'22)*, May 2022.

[50] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "Intelligen: Automatic driver synthesis for fuzz testing," in *Proc. of the 43rd International Conference on Software Engineering (ICSE'21)*, May 2021.

[51] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "Apicraft: Fuzz driver generation for closed-source SDL libraries," in *Proc. of the 30th USENIX Security Symposium (USENIX Security'21)*, 2021.

[52] H. Green and T. Avgerinos, "Graphfuzz: library API fuzzing with lifetime-aware dataflow graph," in *Proc. of the 44th International Conference on Software Engineering (ICSE'22)*, May 2022.

[53] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proc. of the 33rd European Conference on Object-Oriented Programming (ECOOP'19)*, Jul. 2019.

[54] A. Hora, "Apisonar: Mining api usage examples," *Software: Practice and Experience (SPE)*, vol. 51, no. 2, pp. 319–352, 2021.

[55] R. P. Buse and W. Weimer, "Synthesizing API usage examples," in *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, Jun. 2012.

[56] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Adding examples into Java documents," in *Proc. of the 24th IEEE International Conference on Automated Software Engineering (ASE'09)*, Nov. 2009.

[57] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API usage examples from test code," in *Proc. of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, Sep. 2014.

[58] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "APISan: Sanitizing API usages through semantic cross-checking." in *Proc. of the 25th USENIX Security Symposium (USENIX Security'16)*, Aug. 2016.

[59] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. of the 29th International Conference on Software Engineering (ICSE'07)*, May 2007.

[60] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, Sep. 2011.

[61] X. Sun, X. Chen, Y. Zhao, P. Liu, J. Grundy, and L. Li, "Mining Android API usage to generate unit test cases for pinpointing compatibility issues," in *Proc. of the 37th IEEE International Conference on Automated Software Engineering, (ASE'22)*, Oct. 2022.

[62] S. Thummalapenta, T. Xie, N. Tillmann, J. D. Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, Aug. 2009.

[63] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? an empirical study," *Journal of Systems and Software*, vol. 171, p. 110806, 2021.

[64] X. Zhang, Y. Yu, G. Gousios, and A. Rastogi, "Pull request decisions explained: An empirical overview," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 849–871, 2022.

[65] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "Acceptance factors of pull requests in open-source projects," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1541–1546.

[66] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th Conference on IEEE/ACM International Conference on Program Comprehension*, 2018, pp. 348–351.

[67] "Artifact for this paper (currently anonymised)," https://doi.org/10.5281/zenodo.13919240, 2024, zenodo.