École Polytechnique Fédérale de Lausanne

TestGen API collection and extraction

by Çagin Tanir

Bachelor Project Report

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Nicolas Badoux
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 11, 2021

Acknowledgments

I would like to thank my supervisor Nicolas Badoux for his continuous support and constructive feedback. His guidance towards solution to many problems, and teaching of research process was extremely valuable. I would like to thank Begüm for her encouragement whenever I needed and my parents for believing in me to overcome any obstacle I face. Last but not least, I would like to mention my dear friends Mert, Utku and Tarık who accompanied me with their online presence through long nights of work.

Lausanne, June 11, 2021 Çagin Tanir

Abstract

Libraries are a fundamental part of any software. Debian distribution includes an immense number of libraries. They provide vital functionalities to many consumer programs as well as other libraries. These functionalities are exposed to external consumers via an Application Programming Interface (API). The complete documentation and testing of an API is essential for mitigating software vulnerabilities. However, it is a challenging task that requires building particular states of API function calls. TestGen has the aim of automating generation of test suites for this task. To do so, it requires *API collection and extraction*. We infer the API of a target library leveraging a text-based approach. Scalability of this approach allows analyzing vast amount of consumer programs. Among the possible API functions we extracted; we had a 90% accuracy. We finally identified the minimum set of consumer programs/libraries to use in TestGen to achieve maximum coverage of the API.

Contents

Chapter 1

Introduction

In the past 30 years, we got broader hardware capabilities. We leveraged enlarged computational power to execute more complex and demanding software. This complexity and extra functionalities lead the code to grow bigger and bigger, leaving out the possibility to write stand-alone programs that handles every task we would like. Consider a program that needs a secure connection over the internet, rather than rewriting delicate cryptographic protocols, the programmer could use `libssl` [1] which provides these features. It is not only the convenience of reusing the code but also the reliability of tried and tested code by many other programs. In the case where a programmer rewrites this code; security vulnerabilities may arise which were already solved in `libssl`. Programmers, for reasons like development costs, lack of proficiency or simply for modularity of their code choose to rely on external libraries.

The Application Programming Interface (API) is the subset of functions in a library which is available to software engineers to use the functionalities provided by the library. Documentation of an API is critical to correct usage of the library as many programs and even other libraries depend on it.

Debian, a Linux based operating system, has numerous libraries in its main distribution [2]. Like any software, libraries might have bugs that could incur serious damage to consumer programs and libraries. Unfortunately, Debian libraries are not thoroughly tested and their API is not always well documented.

With TestGen, we would like to prepare automated test suites with input parameter and output validation. These test suites would allow us to mitigate security vulnerabilities. TestGen requires the API of a library to derive test cases.

The main goal of this project is to correctly infer the API of a library. To do so, we acquire the source code of a library and its consumers. We follow a text based search to extract the API, and finally we analyze it to find the consumers that could diversify test cases in TestGen.

The main challenge is to scale API extraction of a library we target, to a large number of consumers which allows us to fully determine the API and analyze the consumers to use in our test suites.

Chapter 2

Background

2.1 Libraries

Libraries are a set of functions, macros and data that provide a certain behavior for programs. They facilitate life for developers as software we use today requires a lot of common features. For example, many programs dealing with images accept an image in the PNG file format. Instead of rewriting the same code, software engineers can benefit from a library providing those features, in this case `libpng` for example. A program that uses an external library can be reffered as a *consumer*, and we will call the external library as *consumed*.

In fact, this consumer-library scheme is so beneficial that usually libraries themselves are consumers of other libraries. Chain of nested dependencies highlights the importance of correct functioning of each library. A bug in a library could have a cascading effect on other programs and libraries that use it. Consequence might be a software vulnerability or even a crash in the consumer. This emphasizes the necessity of thoroughly testing libraries before publishing.

2.2 Library types

There are two types of libraries, static and dynamic. They differ from each other in terms of inclusion to the consumer program. Static libraries are attached to the program at compile time, and their code remain within the consumer program. The greatest advantage of static libraries is compatibility. All code resides in one executable module barring any unexpected external issues or dependencies though it means the overall size of the binary increases. Another disadvantage is the update procedure of a static library. Since a static library resides inside the program that uses it, any updates such as to fix bugs or security flaws would require the update of the whole program.

On the other hand, linking dynamic libraries is the addition of object code that fetches the external shared library and connects it with the executable. This implicates a compatibility constraint in compilation between the core program and the consumed library. Namely, their Application Binary Interface (ABI) which is offered by compilers must be compatible. When a discrepancy occurs between ABIs, the linking could fail. It is no surprise that distribution and usage of the libraries must comply with certain rules to mitigate compatibility issues.

2.3 The C pipeline

As we target libraries in Debian and that they are mostly written in C, we look at the pipeline of a C code, from a text file until execution.

First, the preprocessor prepares the C source code, namely looks for preprocessor directives, provides inclusion of header files, macro expansions, and conditional compilations. The most frequent usage of preprocessor directives is the `#include` directive. A software engineer can use the `#include` directive to *link* an external library in their consumer program/library.

After this phase, the C source code is ready for compilation. The compiler generates multiple C object file. They are independent binary files at this point. As a whole, they form the consumer program. The final step is the linker. Its job is to merge all the C object code into one executable machine code. Crucially, this is where an outside library is linked to the consumer program's —or a library's— executable file.

2.4 Application Programming Interface

Libraries provide common usages which relieve the developer from rewriting code that does the same task. In order to benefit, developers need an entry point to the consumed library. This is where the *Application Programming Interface* (API) comes into play. An API is the interface that connects a library and the consumer program. It abstracts away the internal implementation details of the library. By doing so, it saves the programmer valuable time to understand how the library works. Understanding API functions is enough to leverage the full potential of a consumed library. Therefore, documentation of an API plays an immense role to grasp and correctly use the API.

2.5 Ctags

Ctags is a tool used for generating indexes or equivalently tags [3]. These allow us to quickly search and locate language objects found in source files. The tags Ctags can find include class names, macro definitions, function definitions and function declarations among others. The exact list of possible tags varies between the 41 supported programming languages.

Ctags was first introduced in 1985 by Ken Arnold. It was included in the discontinued operating system Berkeley Software Distribution based on Research Unix [4]. It uses language specific parsers to create indexes. Ctags eventually became a project on its own, and newer versions are maintained by the open-source community [5].

The wide availability of Ctags across multiple platforms is only one of the reasons why it is a beneficial tool [6]. For us, it provides fast and clean output which facilitates the analysis of the API. To further establish suitability and scalability of Ctags in our project, we observe that Debian Sources website is using Ctags to index functions in all the Debian libraries which can be queried by users.

2.6 Cscope

Cscope is a tool for browsing code dating back to early 1980s at Bell Labs [7]. It was first written by Joe Steffen as an aid for his own work. The first version started out with basic parsing tools like `grep` and `sed`. Then, he wrote a C program that parses tags into an updatable database. In a short period of time, it became a demanded tool and the updated version made into the official AT&T Unix distribution. Later, it moved to UNIX System Laboratories (USL) and eventually Santa Cruz Operation who acquired USL. Ctags was open-sourced in 2000 [8].

2.7 Cscope and Ctags comparison

Thanks to its command line interface, Cscope makes the analysis of an API easily integrable in scripts. Although it seems like providing the same knowledge with Ctags, it has a significant advantage which is the possibility to search for *functions called by another function*. In other words, it searches for function *calls* and **not** function *definitions*. This uncovered usage by Ctags is why we needed Cscope as API calls by a program to a consumed library is done by function *calls*. On the other hand, Cscope provides all the global definitions and not only functions which create a lot of overhead that is irrelevant to the API.

Chapter 3

Design

Here we give a general outline of the method we developed for API extraction. We explain two different design choices we considered for API analysis and compare how they fared against each other.

Our goal is to extract the API of a library from its consumers. Then, we determine which consumers to test whilst maximizing the coverage of API usage. Consumers generally don't use every function of an API. Usage of an API depends on a consumer's functionality requirements. Consider `libxml2` [9], an extensive library to handle XML files and two of its consumers `imagemagick` [10] and `libxdmf3` [11]. While `libxdmf3` uses `xmlBufferShrink`, an API function of `libxml2`, `imagemagick` doesn't need the functionality this API function provides.

As the number of consumers we analyze increases, we go towards finding the totality of the API functions. So, our first step is to find a list of consumers. We depend on documentation and package managers for this task because libraries are not necessarily linked to their customers.

They might have forward dependency lists. For example, a program could list the libraries it uses to correctly perform. In this case, we retain the programs/libraries that have our consumed library in its dependencies. Another possibility is that the package manager might propose a tool showing reverse dependencies [12] [13].

Once a list of consumers is found, we considered two options to proceed.

3.1 Source Files Based Search with Header Files

We parse consumers' source files for `#include` directives as this is how libraries are included in the main program. We also enumerate the function calls of these consumer files. Then, we extract included header file names. We search each name in consumed library's source files. If

we find it, we enumerate its function definitions because the consumer must have used some or totality of the functions in that *consumed* library's source file. We will find the API by taking the intersection of these two sets.

3.2 Source Files Search without Header Files

This approach mainly consists of two steps. First, the enumeration of all function definitions in a consumed library. The API will be a subset of these functions. To find the desired subset, we iterate over every source file of consumers. Then, we extract all the function *calls* and not the function definitions. Function calls are the potential API usages. Finally, we get the intersection of the two, which is the API of the consumed library. Below is an example of consumed library, `libxml2` at Listing 3.1, and a conceptual consumer at Listing 3.2. `xmlBufferAddHead` is a function definition in the consumed library which is one of the API functions for `libxml2`. While parsing the consumed library, we will add `xmlBufferAddHead` in our function list. This is where we enumerate the function definitions in the consumed library.

Underneath, we have a consumer of `libxml2`. Here, we extract the function calls. They are `xmlBufferAddHead`, `foo` and `baz`. This is the set of possible API function calls.

By taking the intersection, we will only retain `xmlBufferAddHead` which is an API function of `libxml2`. Notice that, an internal function call `foo()` in line **8** is dropped of during the intersection.

Listing 3.1: `libxml2` library

```
1 int xmlBufferAddHead(xmlBufferPtr buf,
2                       const xmlChar * str, int len){
3    .
4    .
5 }
```

Listing 3.2: a consumer of `libxml2` library

```
1  int foo(){
2      .
3      int a = xmlBufferAddHead(buf, str, len);
4      .
5  }
6  void bar(){
7      .
8      int foo_result = foo();
9      baz(arg1, arg2);
10     .
```

```
11 }
```

3.3 Parse the API documentation of a library

Another approach to determine the API of a library is to parse the documentation provided by the distributer of the library. First, the documentation of each library is different. It would be extremely time-consuming to adapt our parser for the structure of each library's documentation. Also, there might not be a documentation for a library.

3.4 Search design comparison

The first choice has some drawbacks. It depends on how header file inclusions are organized. They can in practice be very unorganized in Debian Libraries. This comes from two types of inclusion, namely `#include "filename"` and `#include <filename>` [14]. The `#include` with angle brackets searches a sequence of implementation-defined places for a header. The version with quotation marks first searches the current directory and if a file is not found, it then acts as `#include <filename>`. Unfortunately, this difference is not enforced for libraries distributed by Debian. Programmers might use quotation marked version for external source file inclusion as well. This ambiguity results in search and analysis of incorrect source files.

Another and more critical drawback of first choice is the usage of relative paths, absolute paths and wildcard patterns when looking for a source file [15]. Inclusions such as `#include <../libxml2/read.h>`, `#include <libxml2/../write.h>` are possible options for convenience. However, they make text-based search hard to automate and result in missing files for our analysis.

We opted for the approach without header files. It is more convenient to automate and easily scalable as it does not suffer the drawbacks of version with header files.

3.5 Identification of consumers for test suite

In order to determine which consumers to test, we use the API coverage information of each consumer. We require the user to input a cover percentage threshold. We choose the minimal set of consumers such that their combined coverage exceeds the threshold. This is in fact analogous to Set Cover Problem which is NP-complete [16]. We follow a greedy approach. At each iteration, we look for the consumer that covers the largest part of the remaining API functions. As a result, we find the best candidates to write test suite which can reach most of the API use cases.

Chapter 4

Implementation

4.1 Libraries to analyze

The API extraction we outlined is valid for any system and language though for the scope of this project we considered any Debian packages. To analyze meaningful libraries with non-trivial number of consumers, we first gathered the information from Debian Popularity Contest [17]. This project gathers reports sent by users. Its purpose is to find out the most downloaded and the most regularly used libraries/programs.

Among the libraries we found on Debian Popularity Contest, we filtered those with the highest number of consumers. The reverse dependencies of a library, in other words its consumer list, is available via `apt-cache rdepends [library]` provided by APT package management system.

4.2 API extraction

We ask user the name of the consumed library and (**n**) number of consumers to analyze. Using `APT`, we list its consumers. We choose **n** of them. Then, using APT's `apt-get source [consumer/consumed library]` we download the source files of the target library and the consumers.

We run `Ctags` to enumerate all function definitions in the consumed library. For each consumer, we use `Cscope` to list all function *calls* inside another function. Their intersection results in the API of the target library as explained in design.

4.3 Optional post-processing

Due to text-based approach, we could get name collisions that result in false positives for the API. They are the functions we identified as part of the API whereas they are functions of another library or internal functions of a consumer.

A recurrent example is the standard C library, `libc6`. Libraries usually override functions from `libc6`, so they appear in our consumed library's function list. It also appears in the function calls list of a consumer, so it is incorrectly included in our API extraction.

To mitigate these false positives, we rely on a heuristic that API functions of a library have the same prefix. In fact, this is a common and highly recommended convention [18]. For example, `libxml2` has the prefix `xml_` and `libcairo2` has the prefix `cairo_`.

We take an optional input from the user, the prefix for consumed library's API functions. In case this input is absent, this filtering is skipped. We check the function **definitions** in the **consumers**. If a function definition in a consumer doesn't have the prefix and was found to be in API, we filter it out.

4.4 Outputs

We output the API functions we extracted. Also, we plot which percentage of API each consumer uses. Last but not least, we list the consumers such that their combined coverage of API exceeds the threshold user chooses. We show the amount of coverage change with addition of each consumer.

Chapter 5

Evaluation

We tested our code on three libraries, namely `libxml2`, `libcairo2` [19], `libopus` [20]. The execution time depends largely on size of the consumed library and the consumers. However, after testing numerous times with different consumers, we could provide a stable estimation. (Table 5.1).

#consumers	duration
10	< 2
25	< 5
50	< 10

Table 5.1: Estimated duration of the program in minutes

We have listed how many possible API functions we discovered in terms of the number of consumers. (Tables 5.2, 5.3 and 5.4).

We also output what percentage of the extracted API each consumer uses. (Figures 5.1(a), 5.2(a), and 5.3(a)).

Among the consumers we analyze, we greedily look for the minimum number of consumers that covers maximum of the API we found, as we explained in Section 3.5. We then output these results. (Figures 5.1(b), 5.2(b), and 5.3(b)).

Notice that we have listed two different executions for libmxl2 (Tables 5.4, 5.5). The reason is that two consumers had a largely superior coverage to the other consumers which can be observed in Figure 5.3(a). We compared our extracted API functions with the official API documentation of libxml2 [21]. We realized that almost half of the functions were false positives. They were internal functions of `libxml2` library. Upon further research of the consumers in question, namely `libvtk6.3` [22] and `libxdmf3` [23], we found out that these consumers has **the source files of the consumed library libxml2** in their package. This incorrect way of using a `Shared Library` makes our API extraction process hard to automate. We need to exclude these corner

cases to avoid unacceptable number of false positives. When we removed these two consumers from our analysis, the API functions we extracted were 90% correct. (Table 5.4, Figure 5.4)

#consumers	#possible API functions
10	172
25	238
50	274

Table 5.2: libcairo2 API extraction evaluation

#consumers	#possible API functions
5	28
10	41
20	46

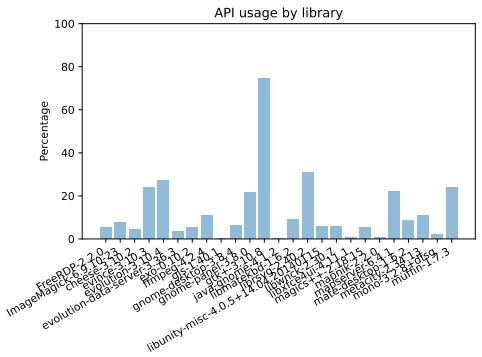
Table 5.3: libopus API extraction evaluation

#consumers	#possible API functions
10	2093
25	2017
50	2148

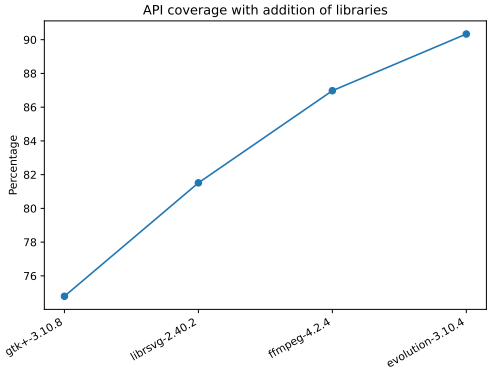
Table 5.4: libxml2 API extraction evaluation

#consumers	#possible API functions
10	113
25	386
50	426

Table 5.5: libxml2 API extraction evaluation exluding `VTK` or `xdmf`

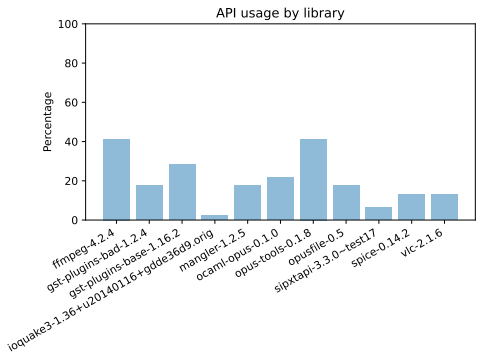


(a)

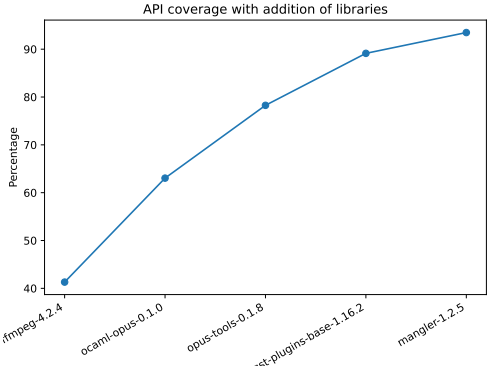


(b)

Figure 5.1: libcairo2 with 25 consumers

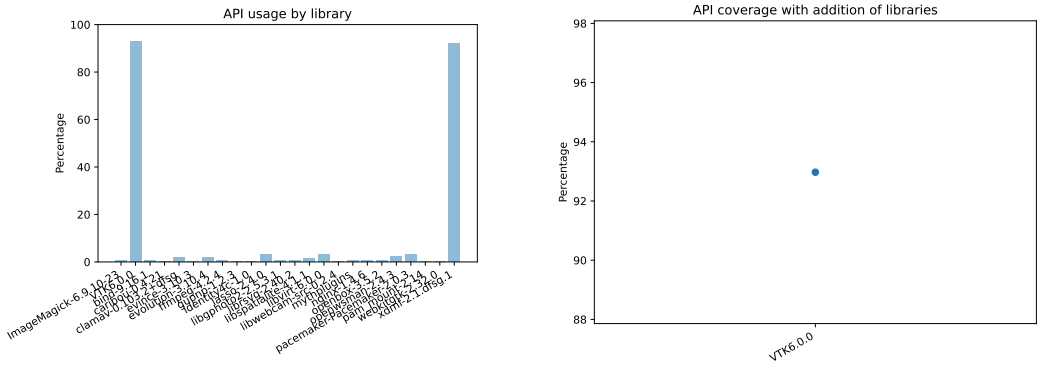


(a)



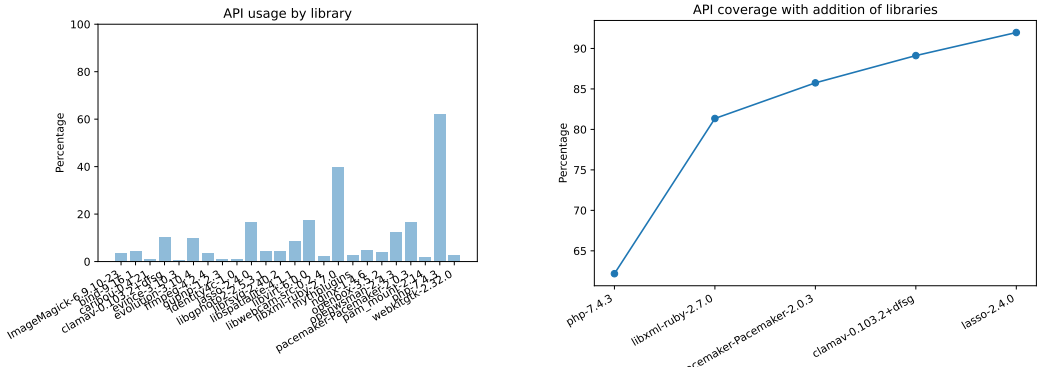
(b)

Figure 5.2: libopus with 11 consumers



(a) (b)

Figure 5.3: libxml2 with 25 consumers



(a) (b)

Figure 5.4: libxml2 with 25 consumers excluding `VTK` and `xdmf`

Chapter 6

Related Work

6.1 Fuzzgen

FuzzGen [24] requires an API extraction for its automatic fuzzer generation. It leverages source files of consumers. It enumerates all declared functions in the consumed library. Then, it identifies all functions that are declared in all included headers of all consumers. The intersection of the two gives the potential API functions as a result. To mitigate over-approximation of inferred library functions, it attempts to compile each potential API function in a test program linked with the target library.

As Fuzzgen looks for all included headers of all consumers, it faces the drawbacks described at Section 3.4. We can observe in the source code of Fuzzgen that it needs blacklists, or library specific filters to handle header files on relatively small number of consumers. Our approach scales better without library specific code for the main API extraction. However, we do take an optional input to minimize false positives.

On the other hand, Fuzzgen's linker-based check on validity of possible API function calls is a robust way to eliminate false positives. Unfortunately, we couldn't include it in our design as we've faced compilation errors other than expected linker errors. Furthermore, it would become an extremely time-consuming step as we can analyze much more consumers.

6.2 APISan

APISan [25] is a tool that checks API usages through semantic cross-checking. Like our tool, it relies on source files of the consumers of a target library. It does not only index the function but rather builds symbolic contexts for different types of code. Then, it infers correct usage of the API in a probabilistic manner.

Chapter 7

Conclusion

With our API extraction that can take numerous consumers into account, we find the API of a consumed library. Our text-based approach lets us easily scale the number of consumers we can analyze. We highlight which consumers are the most suitable for TestGen to generate test suites. This way, we hope to have maximum performance out of TestGen and discover potential vulnerabilities.

Bibliography

[1] *libssl Library Information.* URL: https://packages.debian.org/jessie/libssl1.0.0.

[2] *Debian Distribution Libraries.* URL: https://packages.debian.org/stable/libs/.

[3] *What is Ctags?* URL: http://ctags.sourceforge.net/whatis.html.

[4] *Ctags BSD Manual Page.* URL: https://www.unix.com/man-page/bsd/1/ctags/.

[5] *Universal Ctags Github Page.* URL: https://github.com/universal-ctags/ctags.

[6] *Advantages of Ctags.* URL: http://ctags.sourceforge.net/desire.html.

[7] *Cscope.* URL: http://cscope.sourceforge.net.

[8] *History of Cscope.* URL: http://cscope.sourceforge.net/history.html.

[9] *libxml2 Library.* URL: http://www.xmlsoft.org/index.html.

[10] *imagemagick package information.* URL: https://packages.debian.org/buster/imagemagick.

[11] *libxdmf3 package information.* URL: https://packages.debian.org/buster/libxdmf3.

[12] *R Reverse Dependency Tool.* URL: https://stat.ethz.ch/R-manual/R-devel/library/tools/html/dependsOnPkgs.html.

[13] *Debian Reverse Dependency Tool.* URL: https://manpages.debian.org/stretch/apt/apt-cache.8.en.html.

[14] *The C Standard.* URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf#page=182.

[15] *Specifying Path Names for Include Files.* URL: https://caligari.dartmouth.edu/doc/ibmcxx/en_US/doc/complink/tasks/tuinclud.htm.

[16] R. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations.* Ed. by R. Miller and J. Thatcher. Plenum Press, 1972, pp. 85–103.

[17] *Debian Popularity Contest.* URL: https://popcon.debian.org.

[18] *GNU Standards.* URL: https://www.gnu.org/prep/standards/standards.html.

[19] *libcairo2 Library Page.* URL: https://packages.debian.org/buster/libcairo2.

[20] *libopus0 Library Page.* URL: https://packages.debian.org/sid/libopus0.

[21] *libxml2 API Documentation.* URL: http://xmlsoft.org/APIfunctions.html.

[22] *libvtk6.3 Library Page*. URL: https://packages.debian.org/buster/libvtk6.3.

[23] *libxdmf3 Library Page*. URL: https://packages.debian.org/buster/libxdmf3.

[24] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. "FuzzGen: Automatic Fuzzer Generation". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[25] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. "APISan: Sanitizing API Usages through Semantic Cross-Checking". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 363–378. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun.