DOI: https://doi.org/10.59294/HIUJS.26.2023.538

Phương pháp sinh trình điều khiển fuzzing trong kiểm thử thư viện phần mềm với điều kiện thiếu thông tin về ngữ cảnh sử dụng

Phùng Thế Bảo[1,*], Trần Chí Thiện[2] và Tạ Hữu Vinh[3]

[1]Trường Đại học Văn Lang, [2]Viện lập trình hệ thống, Viện Hàn lâm khoa học Nga
[3]Trung tâm Công nghệ, Học viện Kỹ thuật quân sự, Hà Nội

TÓM TẮT

Kiểm thử fuzzing đóng vai trò quan trọng trong vòng tuần hoàn phát triển phần mềm an toàn. Đối với các thư viện phần mềm đòi hỏi phải xây dựng các trình điều khiển fuzzing (fuzz driver) để phân phối và thực thi các mẫu kiểm cho các hàm trong thư viện. Các nghiên cứu liên quan chủ yếu trích xuất ngữ cảnh sử dụng của thư viện được kiểm trong các phần mềm khác để xây dựng trình điều khiển fuzzing, điều này dẫn đến việc không kiểm tra, đánh giá được hết các hàm được định nghĩa trong thư viện, hoặc không đánh giá, kiểm tra được trong điều kiện thư viện được kiểm chưa được sử dụng trong bất kỳ phần mềm nào. Bài báo đề xuất phương pháp tự động sinh các trình điều khiển fuzzing cho các thư viện ngôn ngữ C/C++ trong điều kiện thiếu thông tin về ngữ cảnh sử dụng thư viện được kiểm. So với việc viết thủ công các trình điều khiển fuzzing cho một số thư viện phổ biến phương pháp này đã sinh được số trình điều khiển vượt trội, giúp giảm thời gian viết mã cho các trình điều khiển fuzzing. Bên cạnh đó, ứng dụng phương pháp được đề xuất giúp phát hiện một số lỗi trong các thư viện phần phổ biến: libpng, pugixml.

Từ khóa: *kiểm thử phần mềm, kiểm thử fuzzing, tạo mã, phân tích tĩnh*

1. ĐẶT VẤN ĐỀ

Thư viện phần mềm là sản phẩm phần mềm được đóng gói với mục đích tái sử dụng trong các sản phẩm phần mềm khác. Thư viện phần mềm bao gồm các định nghĩa về các kiểu dữ liệu mới, các giao thức xử lý dữ liệu, bộ giao tiếp với các phần mềm khác … Không giống như các phần mềm thông thường, thư viện phần mềm có thể không chứa các điểm thực thi ban đầu (entry point), vì thế trong kiểm tra, đánh giá các thư viện phần mềm phương pháp Unit testing [1] thường được sử dụng. Phương pháp này kiểm tra đánh giá thư viện theo các hàm, các chức năng, đòi hỏi người lập trình viên phải: tham khảo tài liệu sử dụng; xem xét, đánh giá mã nguồn; xây dựng các trình điều khiển kiểm thử (test driver) cho các hàm, chức năng trong thư viện; theo dõi và đánh giá kết quả thực hiện các trình điều khiển kiểm thử.

Trong thời gian gần đây phương pháp kiểm thử fuzzing đã đạt được những thành tựu lớn và trở thành tiêu chuẩn trong vòng tuần hoàn phát triển phần mềm an toàn Microsoft (Microsoft Security Development Lifecycle) [2].

Tác giả liên hệ: TS. Phùng Thế Bảo
Email: bao.pt@vlu.edu.vn

Khi áp dụng phương pháp kiểm thử fuzzing cho thư viện phần mềm, lập trình viên cần xây dựng các trình điều khiển kiểm thử fuzzing cho các hàm, chức năng riêng lẻ của thư viện, trình điều khiển kiểm thử fuzzing này được gọi là "fuzz driver". Bảng 1 mô tả fuzz driver trên nền tảng LibFuzzer được sử dụng để kiểm thử hàm *purple_utf8_salvage()* của thư viện *libpurle*. Fuzz driver là một chương trình nhỏ, có 02 chức năng chính: tiếp nhận và phân phối/gán dữ liệu đầu vào cho các biến; xây dựng các lời gọi hàm theo đúng cú pháp.

Với những thư viện phức tạp, dung lượng lớn với số lượng thực thể nhiều (định nghĩa hàm, bản ghi, kiểu dữ liệu mới) việc tham khảo tài liệu, đọc hiểu mã nguồn và xây dựng các fuzz driver tiêu tốn rất nhiều thời gian và công sức của các lập trình viên. Chính vì vậy việc nghiên cứu tự động sinh trình điều khiển fuzzing dành được nhiều sự quan tâm, trong đó có thể kể đến hệ thống Fudge [3] và FuzzGen [4].

Fudge ứng dụng phương pháp phân tích tĩnh

(static analysis) cây cú pháp trừu tượng (Abstract syntax tree) để phân tích kho mã nguồn ứng dụng của Google (Google codebase) nhằm trích xuất những kịch bản (context) sử dụng của các thư viện phần mềm được kiểm. Fudge tập trung vào việc xây dựng trình điều khiển fuzzing cho các hàm có tham số là chuỗi và độ dài chuỗi - những tham số thường gây ra lỗi trong việc cấp phát, giải phóng bộ nhớ, hay xử lý độ dài chuỗi. FuzzGen phân tích mã đại diện trung gian (IR-Intermediate representation) để tìm mối liên hệ giữa các thực thể trong mã nguồn. FuzzGen làm việc tốt khi có kho thư viện của Android. Tuy nhiên đối với những thư viện trên hệ thống Linux, chưa được sử dụng ở các phần mềm khác, FuzzGen gặp nhiều hạn chế và không sinh được các trình điều khiển fuzzing. Cả Fudge và FuzzGen đều thể hiện những hạn chế trong điều kiện thiếu kịch bản sử dụng.

Với mục đích khắc phục hạn chế nêu trên, tác giả đề xuất một phương pháp sinh trình điều khiển fuzzing trong điều kiện thiếu thông tin về kịch bản sử dụng các thư viện phần mềm C/C++ được kiểm thử, từ đó xây dựng mô hình hoạt động của phương pháp sinh trình điều khiển fuzzing, nhằm giảm hao phí lao động, nâng cao độ tin cậy trong việc đánh giá các thư viện phần mềm. Những vấn đề cần giải quyết khi tự động sinh các trình điều khiển fuzzing:

- Phân tích mã nguồn để tìm đặc trưng và các mối liên hệ giữa các thực thể (entity) trong mã nguồn.
- Khai báo và phân phối dữ liệu fuzzing đầu vào cho các biến, các tham số của hàm được gọi, cấp phát và giải phóng bộ nhớ trong trường hợp cần thiết.
- Xây dựng các lời gọi hàm theo cú pháp với các tham số đã được khai báo và phân phối dữ liệu fuzzing.

Bài báo gồm ba phần chính. Phần một giới thiệu chung về lý do nghiên cứu phương pháp kiểm thử fuzzing; khắc phục những hạn chế của hệ thống đã có, tác giả đề xuất một phương pháp sịnh trình điều khiển fuzzing và xây dựng mô hình hoạt động trong điều kiện thiếu thông tin về kịch bản sử dụng. Phần hai trình bày phương pháp và mô hình hoạt động của phương pháp tự động sinh các trình điều khiển fuzzing. Phần ba là kết quả thực nghiệm sinh các trình điều khiển cho các thư viện nổi tiếng: libjson-c; curl; libpostgres; pugixml nhằm đánh giá hiệu quả của phương pháp được đề xuất.

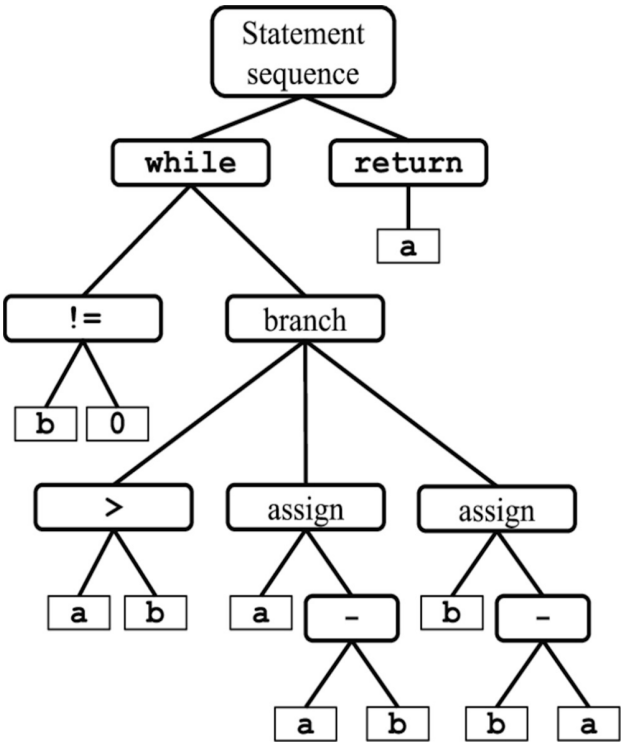
Bảng 1. Ví dụ trình điều khiển fuzzing

Dòng	Mã nguồn	Chú thích
1 2 3 4	*#include "libpurple/util.h"* *#include <stdint.h>* *#include <stdlib.h>* *#include <string.h>*	Kết nối các tệp tiêu đề cần thiết
5	*extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size){*	Khai báo hàm thực thi của nền tảng LibFuzzer “LLVMFuzzerTestO neInput”. Hàm này đóng vai trò như hàm main trong ngôn ngữ C/C++ và có chức năng truyền dữ liệu kiểm thử dưới dạng “Data” và “Size”.
6 7	*char *foo;* *foo = (char *)malloc(Size + 1);*	Khai báo biến foo kiểu chuỗi và cấp phát bộ nhớ độ dài “Size + 1” cho biến này
8	*memcpy(foo, Data, Size);*	Sao chép dữ liệu từ Data có độ dài Size vào biến foo.
9	*foo[Size] = 0;*	Gán ký tự cuối của foo bằng 0
10	*char *tmp = purple_utf8_salvage(foo);*	Khai báo biến “tmp” kiểu chuỗi và gán giá trị trả về của hàm “purple_utf8_salvage” với tham số foo. Trong quá trình thực hiện fuzzing, giá trị của foo sẽ thay đổi theo Data và Size (các dữ liệu này được xử lý bởi công cụ Fuzzing). Đây là chính là lời gọi hàm để đánh giá, kiểm thử.
11 12 13 14 15	*if (tmp == 0) {* *free(foo);* *return 0;* *}* *return 0;* *}*	Giải phóng bộ nhớ đã cấp phát cho foo

2. PHƯƠNG PHÁP TỰ ĐỘNG SINH CÁC TRÌNH ĐIỀU KHIỂN FUZZING

2.1. Phân tích tĩnh để tìm các đặc trưng và các mối liên hệ giữa các thực thể trong mã nguồn

Phương pháp phân tích tĩnh mã nguồn là phương pháp phân tích mà không thực thi mã nguồn được phân tích. Đối với những ngôn ngữ khác nhau có các công cụ khác nhau để phân tích tĩnh. Đối với ngôn ngữ C/C++, Clang static analyzer là công cụ phổ biến và mạnh mẽ nhất. Clang static analyzer biểu diễn mã nguồn chương trình trong cây cú pháp trừu tượng thông qua các nút, cành và lá. Các nút và cành của cây biểu diễn các cấu trúc của chương trình (định nghĩa hàm, định nghĩa bản ghi, định nghĩa kiểu dữ liệu, ...), lá của cây biểu diễn các biến, các giá trị cụ thể hoặc các constant. Ví dụ về biểu diễn của cây cú pháp trừu tượng được mô tả trong Hình 1.



Hình 1. Biểu diễn thuật toán Euclide trong cây cú pháp trừu tượng

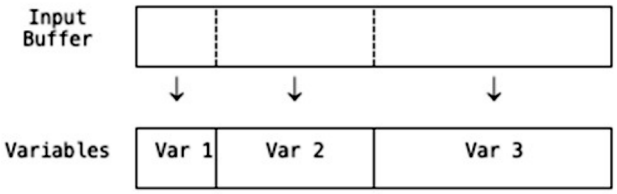
Sử dụng công cụ Clang static analyzer [5] cho phép thực thi phân tích tĩnh trong quá trình biên dịch phần mềm. Các thông tin thu được trong quá trình phân tích tĩnh bao gồm:

- Các định nghĩa hàm: giá trị trả về, danh sách tham số, kiểu dữ liệu tham số.
- Các định nghĩa bản ghi: số lượng trường, tên trường và kiểu dữ liệu tương ứng.
- Các định nghĩa kiểu dữ liệu mới: tên kiểu dữ liệu được định nghĩa, kiểu dữ liệu ban đầu.

Chi tiết về việc phân tích tĩnh có thể tham khảo trong bài báo [6].

2.2. Khai báo tham số và phân phối dữ liệu fuzzing đầu vào

Các công cụ fuzzing cung cấp dữ liệu đầu vào dưới dạng một chuỗi dữ liệu với độ dài xác định. Trong trường hợp hàm được kiểm tra có nhiều tham số với các kiểu dữ liệu khác nhau, cần phải phân tách dữ liệu đầu vào này thành các đoạn dữ liệu tương ứng để cung cấp cho các tham số. Trong [7] đề xuất sơ bộ phương pháp phân tách dữ liệu đầu vào với các tham số có kiểu dữ liệu với độ dài xác định. Đối với các tham số có kiểu dữ liệu phức tạp như tệp văn bản, bản ghi,... phương pháp trong bài báo này sử dụng kết quả của báo cáo [6, 8]. Hình 2 mô tả quá trình phân tách dữ liệu đầu vào cho các tham số của hàm được kiểm tra.



Hình 2. Sơ đồ phân tách dữ liệu đầu vào cho các tham số của hàm được kiểm tra

Trong hình trên, Input Buffer được truyền từ công cụ Fuzzing và được phân tách để khởi tạo dữ liệu cho các biến Var1, Var2 và Var3 của hàm được kiểm.

2.3. Xây dựng lời gọi hàm theo độ phức tạp

Các hàm trong thư viện được phân chia thành 2 nhóm chính:

- **Nhóm hàm xử lý các dữ liệu đầu vào cơ bản**: tham số của những hàm này là những kiểu dữ liệu cơ bản như: char, int, float, double, char *, const char *, file, ... Đối với các thư viện xử lý cấu trúc dữ liệu đặc trưng video, audio, JSON, XML, ... các hàm này cũng có trách nhiệm đọc và tạo cấu trúc dữ liệu tương ứng. Ví dụ: *struct json_object * **json_object_new_int**(int32_t i).*

Hàm json_object_new_int của thư viện json-c có trách nhiệm tạo mới 1 đối tượng struct json_object * từ số nguyên i.

*xmlDocPtr **xmlReadFile**(const char * filename, const char *encoding, int options)*

Hàm xmlReadFile của thư viện libxml2 có chức năng đọc dữ liệu từ filename với các tham số về encoding và options, và tạo mới đối tượng

xmlDocPtr từ dữ liệu đọc được.

- **Nhóm hàm xử lý các cấu trúc đặc trưng của thư viện**: tham số của những hàm này là các biến có kiểu dữ liệu đặc trưng được định nghĩa riêng trong thư viện. Ví dụ:

+ Hàm json_object_object_add (struct json_object *obj, const char *key, struct json_object *val) của thư viện json-c có chức năng bổ sung đối tượng val vào đối tượng obj với từ khóa key.

+ Hàm json_object_get_string (struct json_object *obj) tiếp nhận đối tượng obj với kiểu dữ liệu struct json_object * làm tham số và có chức năng đọc chuỗi từ đối tượng này. Đối với nhóm hàm cơ bản, việc xây dựng lời gọi hàm được thực hiện đơn giản như sau:

+ Khai báo và gán dữ liệu cho các tham số. Các tham số này có kiểu dữ liệu đơn giản, vì thế có thể được gán từ dữ liệu đầu vào của fuzzing.

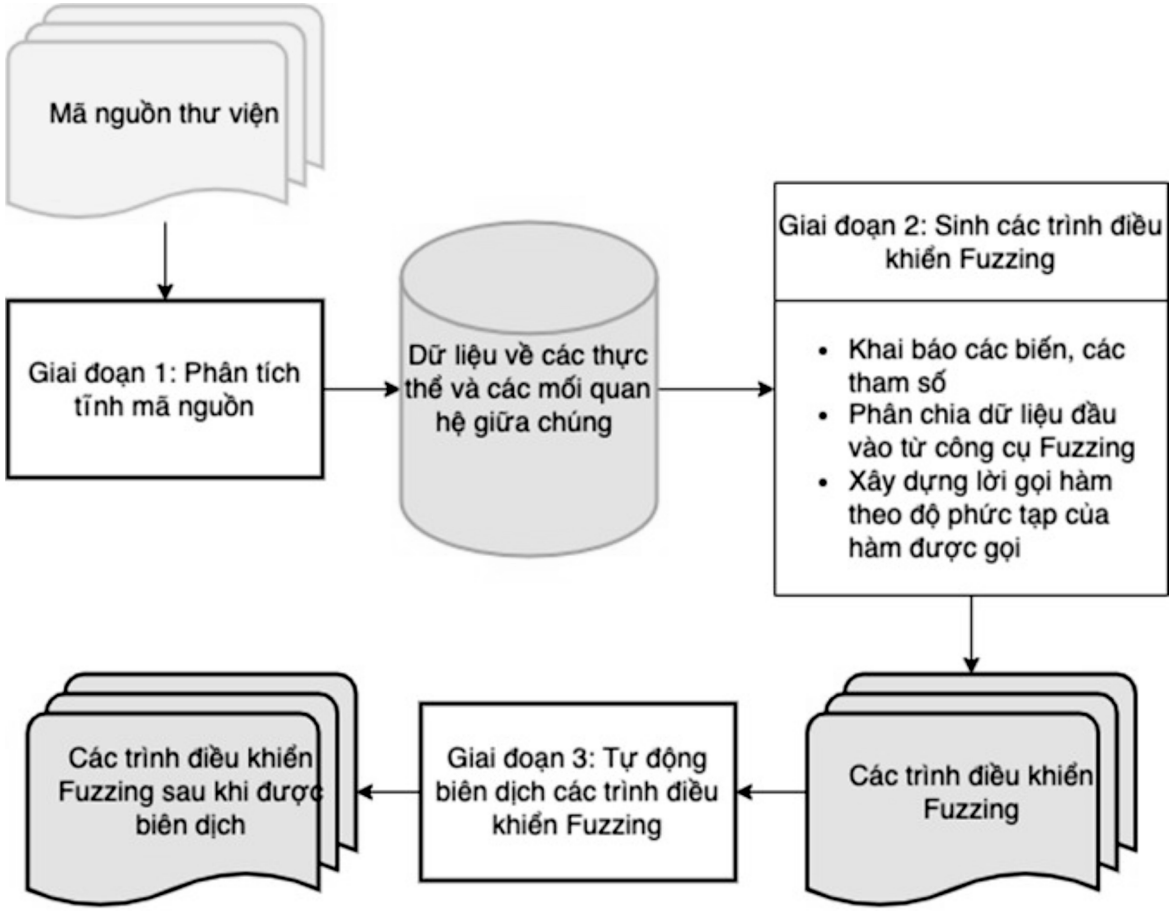
+ Gọi hàm với các tham số được gán.

Đối với các hàm với cấu trúc phức tạp, việc gọi hàm được thực hiện thông qua các hàm cơ bản. Ví dụ, tham số obj với kiểu dữ liệu struct json_object * có thể được khai báo và gán dữ liệu thông qua nhóm hàm xử lý đầu vào cơ bản như hàm json_object_new_int().

Phương pháp xây dựng lời gọi hàm này dựa trên cú pháp và ngữ nghĩa của chính mã nguồn thư viện mà không phụ thuộc vào các kịch bản sử dụng trong phần mềm khác.

2.4. Mô hình hoạt động của phương pháp sinh trình điều khiển fuzzing

Mô hình hoạt động của phương pháp sinh trình điều khiển fuzzing gồm 3 giai đoạn và được thể hiện như hình 3.



Hình 3. Sơ đồ phương pháp sinh trình điều khiển fuzzing

- Trong giai đoạn 1, đầu vào chỉ cần tiếp nhận mã nguồn thư viện để thực hiện phân tích tĩnh. Kết quả phân tích được lưu trữ dưới dạng cơ sở dữ liệu về các thực thể và mối quan hệ giữa chúng.

- Trong giai đoạn 2, các trình điều khiển fuzzing được sinh ra dựa trên dữ liệu về các thực thể và các mối quan hệ. Trong giai đoạn này, việc khai báo tham số, phân chia dữ liệu đầu vào và xây dựng lời gọi hàm được áp dụng. Kết quả của giai đoạn này là các trình điều khiển fuzzing được sinh ra.

- Trong giai đoạn 3, các trình điều khiển fuzzing được biên dịch một cách tự động. Trình điều

khiển fuzzing được xem là được sinh ra tốt khi nó được biên dịch và thực thi fuzzing thành công.

3. KẾT QUẢ THỰC NGHIỆM

Để đánh giá hiệu quả của phương pháp được đề xuất, tác giả tiến hành thực nghiệm sinh các trình điều khiển cho các thư viện nổi tiếng: libjson-c; curl; libpostgres; pugixml.

Các trình điều khiển được đánh giá là được sinh thành công chỉ khi chúng có thể biên dịch và thực hiện fuzzing.

Để đánh giá hiệu quả của phương pháp, tác giả sử dụng nghiên cứu của McConnell [9] so sánh kết quả nghiên cứu với thời gian viết mã thông thường của lập trình viên. McConnell chỉ ra rằng, với những dự án không phức tạp, một lập trình viên có thể viết từ 20 đến 125 dòng mã trong 1 ngày làm việc. Kết quả thực nghiệm của phương pháp đề xuất và kết quả so sánh với thời gian viết mã cho các trình điều khiển fuzzing tương tự một cách thủ công được trình bày trong Bảng 2.

Bảng 2. Kết quả thực nghiệm phương pháp cho một số thư viện phần mềm phổ biến

Thư viện	Phương pháp tự động sinh trình điều khiển fuzzing				Phương pháp viết mã thủ công cho các trình điều khiển fuzzing	
	Số lượng dòng mã được sinh	Số lượng trình điều khiển	Thời gian sinh (s)	Thời gian biên dịch (s)	Thời gian tối thiểu để viết mã (ngày)	Thời gian tối đa để viết mã (ngày)
libjson-c	280019	612	180	3111	2240	11200
libpostgres	84387	29	105	749	675	3375
curl	9617	21	4210	152	77	385
pugixml	2815	58	55	61	22	113

Trong bảng trên, kết quả thực nghiệm đối với thư viện libjson-c đã sinh được 612 trình điều khiển fuzzing cho các hàm trong thư viện trong thời gian 180 giây. Thời gian biên dịch 612 trình điều khiển fuzzing này là 3,111 giây và số lượng dòng mã được sinh là 280,019 dòng. Tương tự, đối với thư viện libpostgres, 29 trình điều khiển fuzzing được sinh trong 105 giây, biên dịch trong 749 giây và chứa 84,387 dòng mã, …

Kết quả so sánh cho thấy, để viết số lượng dòng mã tương ứng với các trình điều khiển fuzzing được sinh tự động, một lập trình viên có thể tốn tối thiểu 2,240 ngày và tối đa 11,200 ngày cho thư viện libjson-c, tối thiểu 675 ngày và tối đa 3,375 ngày cho thư viện libpostgres, …

4. KẾT LUẬN

Trong bài báo này, tác giả đưa ra kết quả nghiên cứu phương pháp sinh tự động các trình điều khiển fuzzing. Cũng như các công cụ fuzzing khác, hiệu quả của phương pháp kiểm thử fuzzing còn phù thuộc vào tính đúng đắn của dữ liệu đầu vào. Nếu dữ liệu đầu vào tốt, thời gian kiểm thử sẽ rút ngắn rất nhiều và độ bao phủ được tăng lên cao.

Kết quả thực nghiệm cho thấy, phương pháp sinh trình điều khiển fuzzing được đề xuất trong bài báo đã rút ngắn được thời gian viết mã cho các hàm của thư viện phần mềm, đặc biệt là các hàm có độ phức tạp không cao và trong điều kiện thiếu thông tin về ngữ cảnh sử dụng của thư viện được kiểm thử.

Tuy vậy, trong trường hợp các thư viện có ngữ cảnh gọi hàm phức tạp, các tham số được khởi tạo thông qua nhiều hàm khác nhau, phương pháp này không sinh được trình điều khiển fuzzing. Để khắc phục hạn chế, có thể tính đến phương án trích xuất và phân tích ngữ cảnh sử dụng của thư viện được kiểm trong các phần mềm khác để tìm ra ngữ cảnh sử dụng đúng và xây dựng trình kiểm thử phù hợp.

TÀI LIỆU THAM KHẢO

[1] S. Shamshiri, "How Do Automatically Generated Unit Tests Influence Software Maintenance", *11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 250-261, 2018.

[2] Potter, Bruce, Microsoft SDL threat modelling tool. Network Security, pp. 15-18, 2009.

[3] D. Babic , "Fudge: Fuzz Driver Generation at Scale", *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[4] K. Ispoglou, "FuzzGen: Automatic Fuzzer Generation, *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, 2020,

pp. 2271-2287. URL: https://www.usenix.org/conference/usenixsecurity20/ presentation/ispoglou.

[5] T. Kremenek, “Finding software bugs with the clang static analyzer”, Apple Inc, pp. 20-28, 2008.

[6] C. T. Tran and S. Kurmangaleev, "Futag: Automated fuzz target generator for testing software libraries", *Ivannikov Memorial Workshop (IVMEM)*, pp. 80-85, 2021. DOI: 10.1109/IVMEM53963.2021.00021.

[7] M. Kelly, C. Treude, A. Murray, “A Case Study on Automated Fuzz Target Generation for Large Codebases”, CoRR, 2019. URL: http://arxiv.org/abs/1907.12214.

[8] C. T. Tran, D. Ponomarev and A. Kuznhesov, "Research on automatic generation of fuzz-target for software library functions", *Ivannikov Ispras Open Conference (ISPRAS)*, Moscow, Russian Federation, 2022, pp. 95-99. DOI: 10.1109/ISPRAS57371.2022.10076871.

[9] S. McConnell, “Software estimation: demystifying the black art”, Microsoft press, 2006.

A method for generating fuzz drivers for libraries in the absence of usage contexts

Phung The Bao, Tran Chi Thien and Ta Huu Vinh

ABSTRACT

Fuzz testing plays an important role in The Security Development Lifecycle. For software libraries, fuzz drivers must be written for each function. Related studies mainly leverage the usage contexts of the tested library in other programs to create fuzzing drivers. These approaches do not test all the functions of the library or do not work in the absence of the usage contexts. This article proposes a method for automatically generating fuzzing drivers for C/C++ language libraries in the absence of information about the usage context of tested library. In practice, this approach generates alot of fuzz drivers, which reduces coding time and finds some bugs in popular libraries: libpng, pugixml.

Keywords: *software testing, fuzzing, code generation, static analysis*

Received: 01/08/2023
Revised: 16/08/2023
Accepted for publication: 21/08/2023