Tự động chọn hàm bị loại bỏ trong kiểm thử bản vá khi sử dụng Thực thi tượng trưng lược bớt

Ninh Thái Phan, Nguyễn Việt Hùng

***Tóm tắt*— Kiểm thử bảo mật bản vá phần mềm cập nhật (Patch testing) được sử dụng khi trong phần mềm có một module thêm mới hoặc có module được chỉnh sửa. Nhiệm vụ của kiểm thử bản vá là đảm bảo phần mềm mới được cập nhật hoạt động mà không có bất kì lỗ hổng bảo mật nào xảy ra. Kiểm thử bảo mật bản vá đòi hỏi rất nhiều thời gian, công sức và yêu cầu người kiểm tra cần phải có kiến thức chuyên sâu về hệ thống. Trong những năm gần đây, Thực thi tượng trưng lược bớt (Chopped symbolic execution) đã được áp dụng thành công trong kiểm thử an toàn thông tin cho các phần mềm một cách tự động hoặc bán tự động. Sử dụng kỹ thuật Thực thi tượng trưng (Symbolic execution), phương pháp này cho phép người kiểm thử chỉ định các hàm, module "không cần thiết" để bỏ qua trong quá trình phân tích, do đó nó cho phép kiểm tra một module cụ thể mà không cần phải thực hiện Thực thi tượng trưng trên toàn bộ chương trình. Thách thức của phương pháp này là cần chuyên gia có hiểu biết sâu về phần mềm cần kiểm thử để chọn ra các hàm "không cần thiết". Trong bài báo này, nhóm tác giả đề xuất một phương pháp mới để tự động chọn ra các hàm được loại trừ khi áp dụng Thực thi tượng trưng lược bớt. Kết quả cho thấy rằng phương pháp của nhóm tác giả có thể được áp dụng một cách hiệu quả hơn so với các thử nghiệm ban đầu của Chopper trong hầu hết các các trường hợp thử nghiệm.**

***Abstract*— Patch testing is the problem that the modified modules (a software update or patch) need to be checked to ensure that they work as expected (function testing) and do not have any vulnerabilities inside it (security testing). Security patch testing requires a lot of time and a professional security knowledge from the tester. In recent years, chopped symbolic execution was successfully applied in automatic or semi-automatic program testing to reduce the amount of testing work. Chopped symbolic execution (Chopper) allows users to specify "uninteresting" functions to ignore during analysis, therefore allows testing a module of software without running all functions of the program. Effectiveness of chopped symbolic execution method in patch testing depends on how good the ignored functions are chosen. In this paper, we proposed a novel method to automatically exclude functions for chopped symbolic execution in patch testing using control flow graph. Moreover, we used cyclomatic complexity to optimize the speed of testing process. Experimental result shows that our method can choose the ignored func-tions automatically with the testing time less than the Chopper in most cases.**

Từ khóa*— *kiểm thử bản vá bảo mật; thực thi tượng trưng; lược bớt.

Keywords*— *security patch testing; symbolic execution; chopper.

I. GIỚI THIỆU

Theo khảo sát, một tỉ lệ đáng kể người dùng có xu hướng từ chối cập nhật phần mềm lên phiên bản mới nhất, vì lo ngại mức độ an toàn của các chức năng mới [10]. Việc kiểm thử bảo mật các bản nâng cấp, bản vá module phần mềm là một công việc thường xuyên phải thực hiện, trước hết người đầu tiên thực hiện công việc đó chính là các lập trình viên. Họ phải tiến hành đánh giá mã khi có một bản cập nhật được công bố, điều đó rất tốn thời gian và công sức. Các bản vá, bản cập nhật có thể dẫn tới những nguy cơ về mất an toàn với những đoạn mã mới [11], [12] bởi vì các lập trình viên không đủ thời gian để tìm ra toàn bộ lỗi trong mã của họ. Do đó, một giải pháp tự động để tìm ra các lỗ hổng bảo mật trong các bản vá là điều thiết yếu.

Thực thi tượng trưng (Symbolic execution) là một kỹ thuật hiệu quả khi áp dụng vào tìm lỗ

Bài báo được nhận ngày 02/3/2022. Bài báo được nhận xét bởi phản biện thứ nhất ngày 21/3/2022 và được chấp nhận đăng ngày 05/4/2022. Bài báo được nhận xét bởi phản biện thứ hai ngày 28/3/2022 và chấp nhận đăng ngày 15/4/2022.

hổng phần mềm. Kỹ thuật này xuất phát từ ý tưởng sẽ sử dụng các biến đầu vào tượng trưng thay cho các biến của chương trình. Sau khi thực thi chương trình và ánh xạ các biến thành các biểu thức kí hiệu với các ràng buộc, sẽ thu được một biểu thức tương ứng với mỗi đường đi của chương trình có thể xảy ra. Tại mỗi câu lệnh điều kiện đều sẽ thêm vào biểu thức điều kiện (Path constraint). Lưu ý rằng việc Thực thi tượng trưng sẽ thực hiện tất cả các điều kiện có thể xảy ra đối với biến khi kiểm tra ràng buộc. Cuối cùng kết quả thu được sẽ là điều kiện biến tương ứng với mỗi đường đi của chương trình [24].

Rất nhiều công cụ đã được xây dựng dựa trên kỹ thuật này đã được giới thiệu để tìm lỗ hổng phần mềm [1], [3-5], [7-9]. Tuy nhiên phương pháp này thường hay gặp khó khăn khi muốn phân tích sâu vào các đường thực thi của các phần mềm phức tạp, cũng như khi giải quyết các điều kiện ràng buộc [2]. Trường hợp này xảy ra rất phổ biến khi gặp các vòng lặp lồng hoặc các điều kiện chồng nhau, khi đó số đường đi sẽ tăng theo cấp số nhân (khi gặp mỗi điều kiện sẽ nhân đôi số đường đi) do đó gây quá tải cho hệ thống hoặc công cụ.

Đối với bài toán kiểm thử bảo mật bản vá, đã có một số công cụ được giới thiệu sử dụng kỹ thuật Thực thi tượng trưng như KATCH, KLEE [6]. Trong đó, KLEE hiện là một trong những công cụ phổ biến nhất trong kiểm thử phần mềm sử dụng Thực thi tượng trưng. Công cụ này là một máy ảo Thực thi tượng trưng được xây dựng với đầu vào lấy từ trình biên dịch LLVM [1]. KLEE được sử dụng trong kiểm thử phần mềm, tuy nhiên lại tỏ ra kém hiệu quả với các chương trình lớn vì vấn đề bùng nổ đường đi (Path explosion problem). Đa số các phương pháp sử dụng Thực thi tượng trưng trong kiểm thử phần mềm sẽ làm việc với toàn bộ các đường đi trong chương trình, do đó đều gặp phải vấn đề bùng nổ đường đi và tỏ ra chưa thực sự hiệu quả. Thực thi tượng trưng lược bớt (Chopped symbolic execution) [2] đã được giới thiệu để giải quyết vấn đề bùng nổ đường đi cùng với công cụ Chopper - một phiên bản mở rộng của KLEE. Các tác giả của Chopper đã đề xuất cơ chế xác định các hàm có thể tạm loại bỏ trong quá trình phân tích và điều này rất hữu dụng với bài toán kiểm thử bản vá, khi có thể giảm bớt quá trình phân tích đoạn mã không có thay đổi sẽ giải quyết vấn đề bùng nổ đường đi.

Về mặt kỹ thuật, Chopper không thực sự loại bỏ các hàm, bởi vì điều đó sẽ dẫn đến kết quả bỏ sót lỗ hổng hoặc cảnh báo nhầm lỗ hổng bảo mật. Thực tế, các phần được loại bỏ sẽ được Thực thi lười biếng (Lazy execute) [2] và được kích hoạt khi phát hiện ra có sự ảnh hưởng (Side effect) tới đoạn mã đang được phân tích. Thực thi tượng trưng lược bớt sẽ tận dụng việc phân tích tĩnh đối với các con trỏ ảnh hưởng trong quá trình runtime để tái kích hoạt việc thực thi các đoạn mã được loại bỏ [2]. Mặc dù phương pháp này tỏ ra rất hữu hiệu, nhưng nó vẫn còn điểm hạn chế chính là việc xác định module nào, hàm nào sẽ lược bỏ hiện đang xác định bởi chính người kiểm thử. Đối với bài toán kiểm thử bản vá, việc xác định hàm nào không có thay đổi và để loại bỏ là có thể thực hiện, tuy nhiên yêu cầu phải tiến hành hiểu và phân tích mã nguồn chi tiết. Vì vậy, điều này sẽ làm giảm đáng kể hiệu quả của các công cụ phân tích hay tìm lỗ hổng tự động sử dụng phương pháp Thực thi tượng trưng lược bớt.

Chính vì lí do được đề cập ở trên, phương pháp tiếp cận của nhóm tác giả sẽ cải thiện khả năng của Chopper bằng cách sử dụng phân tích tĩnh để chỉ ra các hàm nào không cần thiết và được tạm thời loại bỏ trong quá trình phân tích. Việc xác định các hàm không có thay đổi trong bản vá để cho Chopper hoạt động hiệu quả rất khó khăn. Nếu liệt kê toàn bộ hàm không có thay đổi vào trong danh sách loại bỏ hàm sẽ khiến cho Chopper mất rất nhiều thời gian trong việc tìm ra các ảnh hưởng (Side effects), cũng như thời gian để tiến hành snapshot và khôi phục khi cần. Trái ngược với việc đó, việc xác định ra hàm có thay đổi trong bản vá rất đơn giản bằng việc so sánh các commit trên các trình quản lý mã [13-17].

Trong bài báo này, nhóm tác giả đề xuất một phương pháp gọi là "Tự động xác định hàm loại bỏ trong Thực thi tượng trưng lược bớt" (Chop's automatic exclusion). Phương pháp này sẽ xác định các hàm phù hợp để tạm thời loại bỏ với phương pháp Thực thi tượng trưng lược bớt trong bài toán kiểm thử bản vá. Đầu vào được

yêu cầu sẽ là hàm mục tiêu - là hàm có sự thay đổi trong bản vá.

Những đóng góp chính của nhóm tác giả bao gồm:

- Đề xuất phương pháp tự động xác định hàm loại bỏ khi Thực thi tượng trưng lược bớt trong kiểm thử bảo mật bản vá.

- Xây dựng công cụ ChopperAEx, chương trình ứng dụng kĩ thuật đề xuất dưới dạng một thư viện động của opt - LLVM optimizer and analyzer [18].

Trong các phần tiếp theo, nhóm tác giả sẽ trình bày tổng quan kỹ thuật chọn hàm loại bỏ trong phần II; đề xuất phương pháp tự động xác định hàm loại bỏ trong phần III và đánh giá hiệu quả của phương pháp trong phần IV.

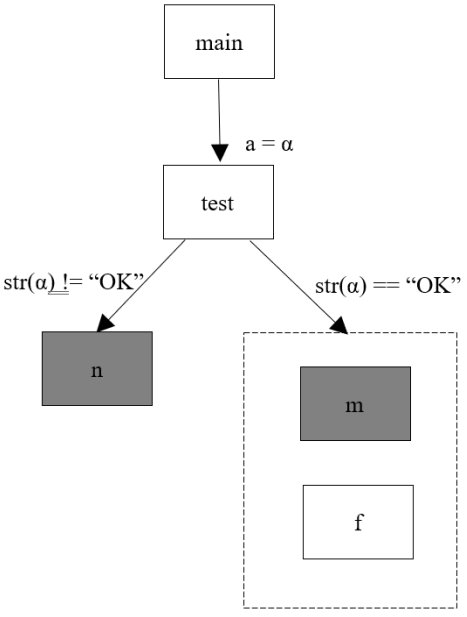
II. KỸ THUẬT CHỌN HÀM LOẠI BỎ

Trong phần này, nhóm tác giả sẽ đưa ra một cái nhìn tổng quan về vấn đề chọn hàm loại bỏ khi sử dụng phương pháp Thực thi tượng trưng trong bài toán kiểm thử bản vá. Hình 1 là một phần giả mã của chương trình minh họa (bao gồm hàm *main* và *test*) cần kiểm thử bản vá. Trong minh họa này, *f* là hàm có thay đổi và là mục tiêu để kiểm thử bảo mật. Hình 2 là đồ thị luồng thực thi (Control flow graph - CFG) của chương trình. Hàm *n* và hàm *m* là đại diện cho các hàm không liên quan trực tiếp đến hàm mục tiêu và có thể được xem xét loại bỏ trong quá trình thực thi kiểm thử hàm mục tiêu.

```
1.void test(char* a){
 2.  char* str1 = "OK";
 3.  if(str(str1) == str(a)){
 4.    m();
 5.    f(a);
 6.    }
 7.  else{
 8.    n();
 9.  }
10.}
11.
12.int main(int args, char** argv){
13.  int x,y;
14.  char* a = argv[1] //symbolic
15.  test(a);
16.}
17.
18.void f(char *a){
  ...
}
```

Hình 1. Giả mã của chương trình ví dụ

Bắt đầu từ hàm main, khi sử dụng Thực thi tượng trưng, sẽ có một biến tượng trưng với biến *a* có giá trị *α*. Để thực thi hàm mục tiêu, đường đi sẽ là *main* đến *test* sau đó *m* và *f* ở cuối cùng, miễn là có điều kiện *str(α)* bằng "OK". Dễ thấy rằng hàm *n* nên được chọn loại bỏ vì nó không nằm trên đường đi dẫn tới *f*. Nếu hàm *n* được loại bỏ, công cụ kiểm thử tự động sẽ không mất thời gian và tài nguyên để khám phá các đường đi được sinh ra từ hàm *n*. Tuy nhiên với trường hợp hàm *m* thì không đơn giản như vậy bởi vì theo luồng thực thi chương trình, muốn gọi hàm *f* thì phải thực hiện hàm *m* trước. Hàm *m* vẫn có thể xét để loại bỏ trong trường hợp toàn bộ con trỏ trong hàm *f* không bị ảnh hưởng bởi hàm *m*. Nói cách khác, không có biến nào trong hàm *f* bị thay đổi khi hàm *m* có thay đổi. Trong ví dụ này, giả sử điều đó đúng và hàm *m* sẽ được chọn để loại bỏ. Cuối cùng, hàm *m* và hàm *n* được chọn loại bỏ với hai lí do khác nhau, lí do đầu là vì hàm không nằm trên đường gọi đến hàm mục tiêu và lí do còn lại, tuy hàm nằm trên đường gọi hàm đến hàm mục tiêu nhưng không ảnh hưởng đến các con trỏ trong hàm mục tiêu.



Hình 2. Minh họa chọn hàm loại bỏ với phương pháp Thực thi tượng trưng lược bớt trên một ví dụ đơn giản

Ví dụ trên đã cho thấy cách chọn tối ưu sẽ được sử dụng để loại bỏ hàm cho bài toán kiểm thử bản vá với phương pháp Thực thi tượng trưng lược bớt. Tuy nhiên, khi sử dụng Chopper, người kiểm thử cần tự xác định các

hàm loại bỏ như trên bằng cách phân tích phần mềm, đó chính là điểm hạn chế đã được đề cập ở phần trên. Để giải quyết vấn đề này, nhóm tác giả đề xuất một giải pháp chọn hàm loại bỏ một cách tự động và sẽ được đề cập chi tiết dưới đây.

III. TỰ ĐỘNG XÁC ĐỊNH HÀM LOẠI BỎ TRONG THỰC THI TƯỢNG TRƯNG LƯỢC BỚT

Thuật toán sử dụng trong giải pháp đã đề cập (tự động xác định hàm loại bỏ trong Thực thi tượng trưng lược bớt) được minh họa trong Hình 3 thông qua sơ đồ khối với các bước chính để chọn ra danh sách hàm loại bỏ được.

Đầu vào là hàm mục tiêu (hàm có sự thay đổi trong bản vá), nhóm tác giả sẽ sử dụng thuật toán duyệt theo chiều rộng - BFS (Breadth-first search algorithm) để tìm đường gọi hàm ngắn nhất tới hàm mục tiêu. Với từng hàm trên đường đi ngắn nhất sẽ xem xét các khối cơ bản trên đó, mục tiêu là tìm ra tất cả các lời gọi hàm liên quan, sau đó kiểm tra các đường đi bắt nguồn từ hàm được gọi có dẫn tới hàm mục tiêu hay không. Những hàm dẫn tới hàm mục tiêu sẽ không bị loại bỏ để đảm bảo việc khám phá tất cả các trường hợp có thể xảy ra khi gọi hàm mục tiêu. Với các hàm còn lại sẽ có thể được loại bỏ nếu thõa mãn thêm điều kiện có độ phức tạp lớn hơn ngưỡng đã được định nghĩa trước. Khi một hàm xác định bị loại bỏ, thông tin về số thứ tự dòng mã của lời gọi hàm sẽ được lấy thông qua thông tin gỡ rối khi biên dịch. Cuối cùng danh sách các hàm bị loại bỏ và vị trí gọi hàm đó sẽ được đưa ra để dùng làm đầu vào của phương pháp Thực thi tượng trưng lược bớt.

Phương pháp tính độ phức tạp của hàm số được trình bày qua sơ đồ trong Hình 4. Độ phức tạp chu kì của một chương trình được định nghĩa theo đồ thị luồng thực thi chương trình (CFG). CFG là một đồ thị có hướng với các đỉnh là các khối cơ bản của chương trình. Hai đỉnh trong CFG có một cạnh nối nếu tồn tại lời gọi từ khối cơ bản này đến khối cơ bản kia. Độ phức tạp M sẽ được tính theo công thức:

$$M = E - N + 2P \quad (1)$$

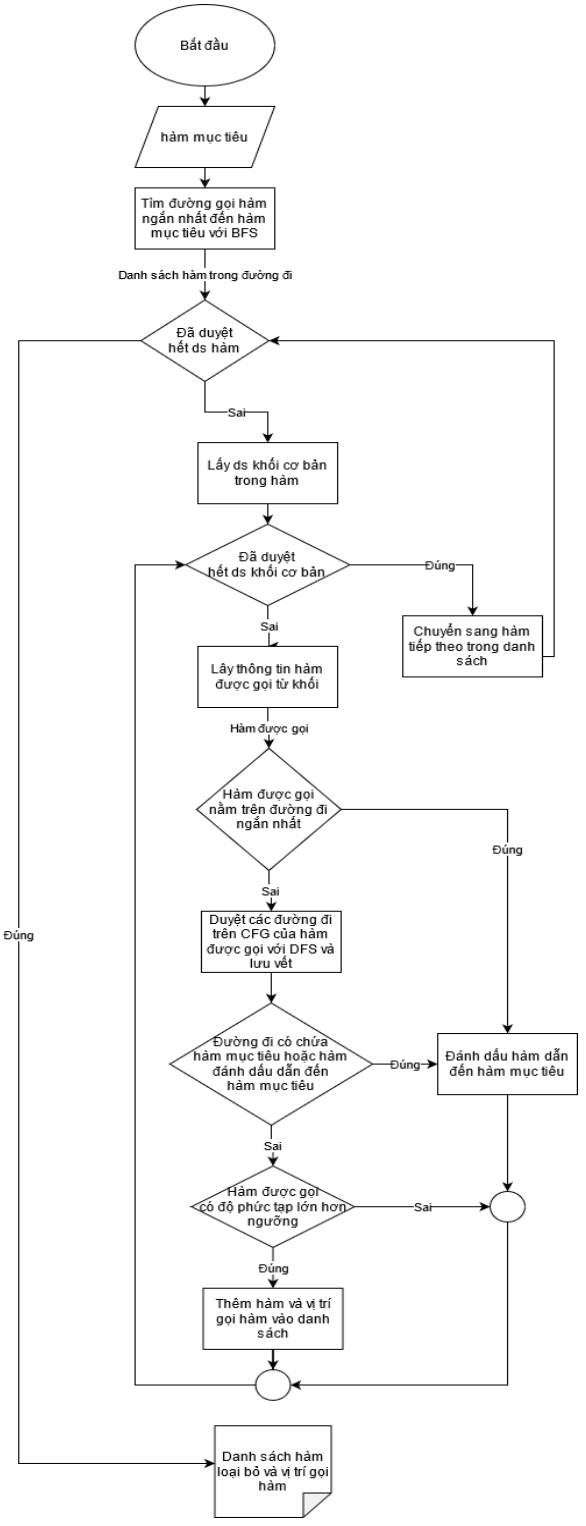
Với E: số cạnh

N: số đỉnh

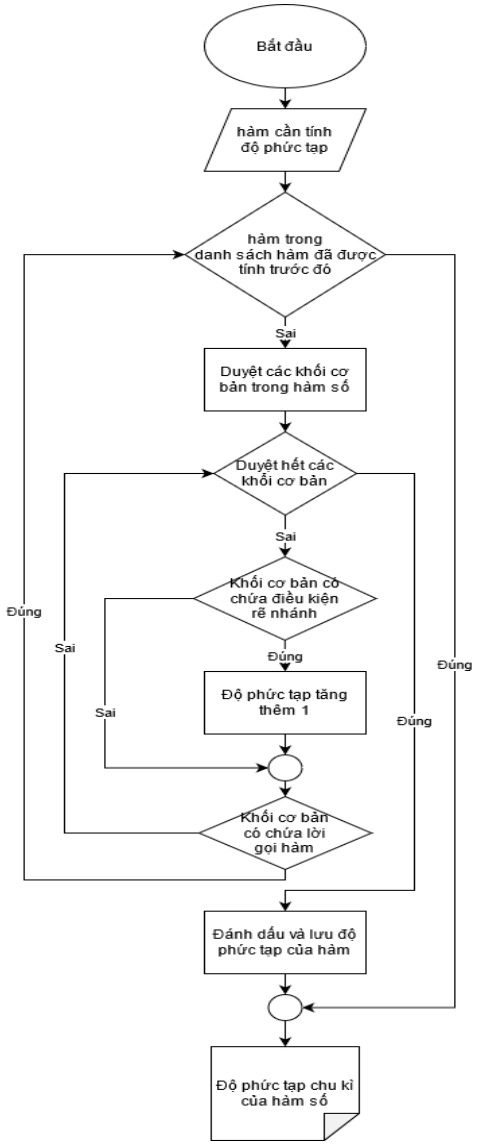
P: số thành phần liên thông của đồ thị

Mà P trong CFG luôn bằng 1 nên sẽ có công thức đơn giản hơn như sau:

$$M = E - N + 2 \quad (2)$$



Hình 3. Sơ đồ khối thuật toán Tự động xác định hàm loại bỏ trong Thực thi tượng trưng lược bớt



Hình 4. Sơ đồ khối thuật toán tính độ phức tạp chu kì của hàm số

Độ phức tạp của hàm số được chia làm hai phần để tính toán. Một là độ phức tạp đến từ đồ thị luồng thực thi nội hàm, tức là đồ thị lời gọi giữa các khối cơ bản trong chính hàm đó. Ở đây nhóm tác giả tính toán phần đó bằng cách duyệt các khối cơ bản và kiểm tra điều kiện rẽ nhánh, vì với một hàm cơ bản nhất sẽ có độ phức tạp là 1, chỉ khi có điều kiện rẽ nhánh độ phức tạp mới tăng lên. Phần thứ hai tạo nên độ phức tạp chu kì của một hàm đến từ các hàm được gọi từ hàm đó. Các hàm được gọi sẽ tạo ra các đường đi mới trong đồ thị luồng thực thi, việc mở rộng đồ thị dẫn đến việc có thể thay đổi độ phức tạp. Lúc này, nhóm tác giả sử dụng thuật toán đệ quy, các hàm được gọi sẽ được tính toàn độ phức tạp nội hàm tương tự như vừa đề cập và có cơ chế lưu vết trong quá trình đệ quy được thực hiện.

IV. ĐÁNH GIÁ

Để đánh giá hiệu quả của thuật toán đề xuất, nhóm tác giả cũng sử dụng các trường hợp kiểm thử với những lỗ hổng bảo mật CVE như trong bài báo giới thiệu Chopper [2], bao gồm 4 CVE với tổng 6 lỗi trong mã nguồn. Toàn bộ các thử nghiệm được tạo dưới dạng một driver thực thi, sử dụng thư viện *libtasn1* để thực hiện một số tác vụ tương tự các tính năng được miêu tả và sử dụng trong các phần mềm khác (như GnuTLS).

CVE 2012-1569 xuất hiện trên các phiên bản trước bản 2.12 (ở đây sử dụng bản 2.11), được thử nghiệm với lỗi truy cập bộ nhớ ngoài vùng cho phép (tức là lỗi Memory corruption) trong tệp *decoding.c* xảy ra tại dòng 137 thông qua việc phá vỡ cấu trúc của dữ liệu đầu vào ASN1.

CVE 2014-3467 xuất hiện trên các phiên bản trước bản 3.6 (ở đây sử dụng bản 3.5), được thử nghiệm với 3 lỗi truy cập bộ nhớ ngoài vùng cho phép trong tệp *decoding.c* xảy ra tại các dòng 152, 709 và 1131 thông qua việc phá vỡ cấu trúc của dữ liệu đầu vào ASN1.

CVE 2015-2806 xuất hiện trên các phiên bản trước bản 4.4 (ở đây sử dụng bản 4.3), được thử nghiệm với lỗi tràn bộ nhớ đệm trên ngăn xếp (Stack-based Buffer overflow) trong tệp *parser_aux.c* tại dòng 574.

CVE 2015-3622 xuất hiện trên các phiên bản trước bản 4.5 (ở đây sử dụng bản 4.4), được thử nghiệm với lỗi đọc dữ liệu ngoài vùng cho phép trong tệp *decoding.c* tại dòng 91 thông qua việc tạo giả chứng thực của ASN1.

Nhóm tác giả sử dụng máy tính với hệ điều hành Ubuntu 14, 4GB ram, CPU i3-8100 @ 3.6GHz với 1 nhân 2 luồng. Kết quả bao gồm thời gian để phát hiện ra lỗ hổng lần lượt với KLEE, Chopper với bản thử nghiệm giống tác giả và Chopper khi sử dụng kèm ChopperAEx để tìm ra hàm loại bỏ một cách tự động. Ở đây ChopperAEx sẽ sử dụng hàm trực tiếp chứa lỗ hổng làm hàm mục tiêu.

Bảng 1 ghi lại thời gian tìm ra lỗi với từng công cụ. Trong đó, *timeout* tức là thời gian vượt

quá 3 tiếng và *Error* có nghĩa đã xảy ra lỗi trong quá trình chạy Chopper.

BẢNG 1. KẾT QUẢ CHI TIẾT KHI TÁI PHÁT HIỆN LỖ HỔNG TRONG LIBTASN1 (CHỈNH BẢNG VÀO 1 CỘT)

CVE	Kỹ thuật tìm kiếm	KLEE	CHOP-PER	ChopperAEx
		Time(s)	Time(s)	Time(s)
2012-1569	Random	499.94	125.11	16.43
	DFS	99.91	27.56	4.96
	Coverage	291.72	136.68	10.88
2014-3467 (1)	Random	**Time out**	487.70	12.12
	DFS	111.87	7.41	5.32
	Coverage	**Time out**	195.45	8.30
2014-3467 (2)	Random	2.29	579.33	67.29
	DFS	**Time out**	159.07	**Error**
	Coverage	2.09	495.18	41.23
2014-3467 (3)	Random	**Time out**	426.40	**Time out**
	DFS	**Time out**	13.90	**Time out**
	Coverage	**Time out**	259.65	**Time out**
2015-2806	Random	281.86	142.68	225.22
	DFS	7733.60	584.82	5990.59
	Coverage	210.33	54.84	171.25
2015-3622	Random	**Time out**	64.37	18.03
	DFS	**Time out**	1123.22	20.54
	Coverage	**Time out**	69.59	18.18

V. KẾT LUẬN

Tự động xác định hàm loại bỏ trong Thực thi tượng trưng lược bớt là một phương pháp loại bỏ các hàm tự động trong bài toán kiểm thử bản vá khi sử dụng Thực thi tượng trưng lược bớt. Đánh giá sơ bộ cho thấy rằng, phương pháp này đem lại hiệu quả tốt hơn so với bản giới thiệu của Chopper trong đa số các trường hợp. Hơn nữa, khi sử dụng phương pháp này, bài toán kiểm thử bản vá có thể thực hiện một cách hoàn toàn tự động và không sử dụng kiến thức chuyên gia cũng như phân tích của người kiểm thử như với Chopper truyền thống. Trong tương lai, nhóm tác giả sẽ tiếp tục cải thiện hiệu năng hoạt động và tìm ra cách chính xác hơn nữa để tránh tối đa việc loại bỏ các hàm nằm trong vùng ảnh hưởng của hàm mục tiêu.

TÀI LIỆU THAM KHẢO

[1]. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Trong: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI'08, pp. 209–224. ACM, California (2008).

[2]. Trabish, D., Mattavelli, A., Rinetzky, N., Cadar, C.: Chopped symbolic execution. Trong: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, pp. 350–360. ACM, New York (2018).

[3]. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: Automatically Generating Inputs of Death. Trong: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS'06, pp. 322–335. ACM, New York (2006).

[4]. angr Homepage, http://angr.io, truy cập cuối vào 12/08/2021.

[5]. BINSEC Homepage, https://binsec.github.io, truy cập cuối vào 12/08/2021.

[6]. Marinescu, P.D., Cadar, C.: KATCH: High-Coverage Testing of Software Patches. Trong: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 235–245. ACM, New York (2013).

[7]. Babic, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test

generation. Trong: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA'11, pp. 12–22. ACM, New York (2011).

[8]. Godefoid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Communications of the ACM 55(3), 40-44 (2012).

[9]. Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. Trong: Proceedings of the 17th ACM conference on Computer and communications security, CCS'10, pp. 585-594. ACM, New York (2010).

[10]. Crameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W.: Staged deployment in Mirage, anintegrated software upgrade testing and distribution system. Trong: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP'07, pp. 221-236. ACM, New York (2007).

[11]. Gu, Z., Barr, E.T., Hamilton, D.J., Su, Z.: Has the bug really been fixed? Trong: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10, pp.55-64. ACM, New York (2010).

[12]. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? Trong: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, pp. 26-36. ACM, New York (2011).

[13]. git Homepage, https://git-scm.com, truy cập cuối vào 12/08/2021.

[14]. GitHub introduce page, https://github.com/about, truy cập cuối vào 12/08/2021.

[15]. GitLab introduce page, https://about.gitlab.com, truy cập cuối vào 12/08/2021.

[16]. SourceForge introduce page, https://sourceforge.net/about, truy cập cuối vào 12/08/2021.

[17]. Bitbucket Homepage, https://bitbucket.org, truy cập cuối vào 12/08/2021.

[18]. opt - LLVM optimizer documentation, https://llvm.org/docs/CommandGuide/opt.html, truy cập cuối vào 12/08/2021.

[19]. Wallace, D., Watson, A., Mccabe, T.: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500- 235, National Institute of Standards and Technology, Gaithersburg (1996).

[20]. LLVM bitcode documentation, https://llvm.org/docs/BitCodeFormat.html, truy cập cuối vào 12/08/2021.

[21]. LLVM 3.4 documentation, https://releases.llvm.org/3.4/docs/, truy cập cuối vào 12/08/2021

[22]. Website: CMake overview, https://cmake.org/overview, truy cập cuối vào 12/08/2021.

[23]. STP Homepage, https://stp.github.io, truy cập cuối vào 12/08/2021.

[24]. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C.S., Sen, K.: Symbolic execution for software testing in practice: preliminary assessment. Trong: Proceedings of the 33rd International Conference on Software Engineering, ICSE'11, pp. 1066–1071, ACM, New York (2011).

SƠ LƯỢC VỀ TÁC GIẢ

Ninh Thái Phan

Đơn vị công tác: Học viện Kỹ thuật Quân sự.

Email: phan123123@gmail.com

Quá trình đào tạo: Học viên chuyên ngành An ninh hệ thống thông tin khóa học 2016-2022.

Hướng nghiên cứu hiện nay: Phân tích lỗ hổng bảo mật; đánh giá kiểm thử; thực thi tượng trưng.

Nguyễn Việt Hùng

Đơn vị công tác: Học viện Kỹ thuật Quân sự.

Email: hungnv@lqdtu.edu.vn

Quá trình đào tạo: Nhận bằng Đại học năm 2006; Thạc sĩ năm 2008 và Tiến sĩ năm 2012 tại Đại học Vật lý Kỹ thuật Matxcova.

Hướng nghiên cứu hiện nay: An toàn thông tin; phát hiện mã độc; trí tuệ nhân tạo.