Bài báo sử dụng 3 kỹ thuật

Kỹ thuật 1: Layout-contributor digraph

Kỹ thuật 2: Layout-oriented fuzzing

Kỹ thuật 3: Control-flow stitching

First, we use dynamic analysis to analyze the vulnerabilities and collect some runtime information in the crashing path. In addition, we inspect corrupted memory objects (denoted as *exceptional objects*), and objects that can be used to locate the exceptional objects. Then we use oriented fuzzing to search alternative diverging paths for exploitable states based on the information collected before. Finally, we try to synthesize new *EXP inputs* to trigger both the exploitable states in diverging paths and vulnerabilities in crashing paths. In certain cases, we can directly generate working exploits. But it is not guaranteed. The complexity of OS kernel is far beyond the ability of current constraint solver. For OS kernel, it is not for the purpose of fully automating exploit generation. Rather, we leverage a *lightweight symbolic execution* to explore exploitability under different contexts.

Đầu tiên, họ sử dụng phân tích động để phân tích các lỗ hổng và thu thập một số thông tin thời gian chạy trong đường dẫn sự cố. Ngoài ra, họ kiểm tra các đối tượng bộ nhớ bị hỏng (được ký hiệu là *các đối tượng ngoại lệ* ) và các đối tượng có thể được sử dụng để định vị các đối tượng ngoại lệ. Sau đó, họ sử dụng làm mờ có định hướng để tìm kiếm các đường dẫn phân kỳ thay thế cho các trạng thái có thể khai thác dựa trên thông tin được thu thập trước đó. Cuối cùng, họ cố gắng tổng hợp *các đầu vào EXP mới để* kích hoạt cả hai trạng thái có thể khai thác trong các đường dẫn phân kỳ và các lỗ hổng trong các đường dẫn gặp sự cố. Trong một số trường hợp nhất định, họ có thể trực tiếp tạo ra các khai thác đang hoạt động. Nhưng nó không được đảm bảo. Độ phức tạp của nhân hệ điều hành vượt xa khả năng của bộ giải ràng buộc hiện tại. Đối với nhân hệ điều hành, nó không nhằm mục đích tự động hoá hoàn toàn việc tạo khai thác. Thay vào đó, họ tận dụng một *thực thi tượng trưng nhẹ* để khám phá khả năng khai thác trong các bối cảnh khác nhau.

*Results* We have build a framework Revery, able to generate working control-flow hijacking exploits for userspace programs. We also build a framework FUZE, able to evaluate the exploit-ability of kernel Use-After-Free vulnerabilities.

*Kết quả* họ đã xây dựng một khung Revery, có thể tạo ra các khai thác chiếm quyền điều khiển luồng điều khiển đang hoạt động cho các chương trình không gian người dùng. họ cũng xây dựng một khung FUZE , có thể đánh giá khả năng khai thác các lỗ hổng Use-after-free của kernel.

**Các khái niệm:**

Crashing point (**Điểm sự cố):** hướng dẫn nơi chương trình gặp sự cố hoặc vi phạm bảo mật bị bắt bởi trình khử trùng, ví dụ: dòng 12

Vulnerability point (**Điểm dễ bị tổn thương):** hướng dẫn nơi xảy ra lỗ hổng (nghĩa là vi phạm bảo mật), ví dụ: dòng 10 trong ví dụ. Đường dẫn sự cố có thể có nhiều lỗi vi phạm bảo mật. Điểm vi phạm đầu tiên được ký hiệu là điểm dễ bị tổn thương.

Exploitable point (**Điểm có thể khai thác):** hướng dẫn có thể dẫn đến khai thác thành công, ví dụ: dòng 14 trong ví dụ. Điểm có thể khai thác dẫn đến trạng thái có thể khai thác khi cỗ *máy kỳ lạ* có thể hoạt động bình thường. Trong thực tế, các hướng dẫn đọc/ghi/thực thi địa chỉ tùy ý (AAR/AAW/AAX) là những điểm có thể khai thác cổ điển.

Diverging path (**Đường dẫn phân kỳ):** đường dẫn có thể tìm thấy các trạng thái có thể khai thác, ví dụ: 9->11->13->14 trong ví dụ.

Hijacking point (**Điểm chiếm quyền điều khiển):** lệnh mà luồng điều khiển có thể bị tấn công, ví dụ: dòng 15 trong ví dụ. Chúng là những điểm có thể khai thác đặc biệt. Trong ví dụ đang chạy, nó là điểm có thể khai thác bậc hai, gây ra bởi điểm có thể khai thác đầu tiên trong dòng 14.

Exploitation path (**Đường dẫn khai thác):** đường dẫn được thực hiện bởi một lần khai thác thành công, ví dụ: 9->10->11->13->14->15 trong ví dụ.

Stitching points (**Khâu các điểm):** các hướng dẫn đặc biệt trong đường dẫn phân kỳ và đường dẫn sự cố, có thể được ghép lại với nhau để tạo ra đường dẫn khai thác, ví dụ: dòng 11 và dòng 13 trong ví dụ. Trong thực tế, có thể có nhiều đường phụ giữa hai điểm nối để khám phá.

User space (**Không gian người dùng**): phần bộ nhớ chứa các tiến trình không có đặc quyền do người dùng điều hành. Nó tách biệt hoàn toàn với không gian nhân (**kernel**), phần bộ nhớ nơi các tiến trình nhân của hệ điều hành đặc quyền được thực thi

Dynamic analysis **(Phân tích động**): kiểm tra và đánh giá chương trình bằng cách thực thi dữ liệu trong thời gian thực, mục tiêu là tìm lỗi trong một chương trình khi chương trình đang chạy, thay vì liên tục kiểm tra mã ngoại tuyến.

PoC (**Proof-of-Concept**): là bằng chứng cho thấy lỗ hổng tồn tại và có thể bị khai thác, việc hiện thực hóa một phương pháp hoặc 1 ý tưởng nhằm chứng minh tính khả thi (đưa ra bằng chứng về khái niệm) hay giải pháp và tiềm năng của nó trong thực tế. (nó dùng trên : software + network)

Weird Machine (**Cỗ máy kỳ lạ**): khung lý thuyết để hiểu sự tồn tại của việc khai thác các lỗ hổng bảo mật. Khai thác tồn tại theo kinh nghiệm, nhưng không được nghiên cứu từ góc độ lý thuyết trước khi xuất hiện khuôn khổ của những cỗ máy kỳ lạ. Trong bảo mật máy tính, nó là một thao tác tính toán trong đó việc thực thi mã bổ sung có thể xảy ra bên ngoài đặc điểm kỹ thuật ban đầu của chương trình, có liên quan chặt chẽ với khái niệm về các hướng dẫn kỳ lạ, là các khối xây dựng của một khai thác dựa trên dữ liệu đầu vào được tạo thủ công

Heuristic(Tiếng Hy Lạp nghĩa là “**tìm**”, “**khám phá**”): Một tính từ chỉ các kỹ thuật dựa trên kinh nghiệm giúp giải quyết vấn đề, học hỏi và khám phá, một quy tắc thông thường nhằm tăng khả năng giải quyết một số vấn đề. Trong khoa học máy tính, thuật toán Heuristic, hay đơn giản là Heuristic, là thuật toán có thể đưa ra giải pháp có thể chấp nhận được cho một vấn đề trong nhiều tình huống thực tế, theo kiểu của một Heuristic chung, nhưng đối với không có bằng chứng chính thức về tính đúng đắn của nó.

**Cách thực hiện: Tìm hiểu lý thuyết về ngữ cảnh, mô hình/phương pháp/cách thực nghiệm được sử dụng trong tài liệu của đề tài; và thực hiện demo về một trong các nội dung được đề cập trong đề tài**

Ngữ cảnh:. Hướng đến việc tự động tạo khai thác lỗ hổng

Do sự thành công của các giải pháp phát hiện lỗ hổng tự động (ví dụ: fuzzing), ngày càng có nhiều lỗ hổng được tìm thấy trong các ứng dụng trong thực tế, cùng với các đầu vào bằng chứng khái niệm (PoC). Do đó, ngày càng có nhiều nguồn nhân lực được dành cho việc đánh giá các lỗ hổng, ví dụ: xác định nguyên nhân gốc rễ và khắc phục chúng. Do đó, nó kêu gọi các giải pháp tự động đánh giá mức độ nghiêm trọng và mức độ ưu tiên của các lỗ hổng.

Đánh giá lỗ hổng, đặc biệt là đánh giá khả năng khai thác, rất quan trọng đối với cả người bảo vệ và kẻ tấn công. Những kẻ tấn công có thể cô lập các lỗ hổng có thể khai thác và viết các khai thác để khởi động các cuộc tấn công. Mặt khác, những người bảo vệ có thể ưu tiên các lỗ hổng có thể khai thác để khắc phục trước và phân bổ tài nguyên phù hợp. Hơn nữa, những người bảo vệ có thể học hỏi từ các lần khai thác để tạo chữ ký IDS (Hệ thống phát hiện xâm nhập) để chặn các cuộc tấn công trong tương lai.

Một cách đơn giản để đánh giá lỗ hổng bảo mật là *phân tích trạng thái chương trình tại điểm xảy ra sự cố* , nghĩa là hướng dẫn dẫn đến sự cố chương trình hoặc vi phạm bảo mật, có thể bị bắt bởi trình khử trùng (ví dụ: AddressSanitizer. Ví dụ: công cụ có thể khai thác của Microsoft kiểm tra tất cả các hướng dẫn trong khối cơ bản của điểm xảy ra sự cố và tìm kiếm các mẫu có thể khai thác đã biết, ví dụ: hướng dẫn chuyển điều khiển với các mục tiêu bị nhiễm độc. HCSIFTER thực hiện thêm một bước để khôi phục dữ liệu bị hỏng do tràn bộ nhớ, cho phép chương trình thực thi nhiều mã hơn sau điểm sự cố và do đó cung cấp các đánh giá đáng tin cậy hơn. Tuy nhiên, các giải pháp này dựa trên kinh nghiệm để xác định khả năng khai thác lỗ hổng và do đó đôi khi không chính xác. Hơn nữa, họ không thể cung cấp đầu vào khai thác để chứng minh khả năng khai thác.

Các nguyên mẫu cho việc tự đông tạo ra khai thác đã được để xuất trong đó có AEG và Mayhem cung cấp các hệ thống đầu cuối để khám phá các lỗ hổng và tự động tạo các khai thác khi có thể, tương ứng cho mã nguồn và mã nhị phân, cũng như Q và CRAX có thể tạo ra các khai thác cho các tệp nhị phân được cung cấp bởi đầu vào PoC -> nhưng những giải pháp này không đủ và chỉ giải quyết được một số vấn đề nhỏ, đặc biệt là rất ít giải pháp khai thác được lỗ hổng dựa trên HEAP

Đối với nhân hệ điều hành có độ phức tạp và khả năng mở rộng cao hơn, nó không phù hợp để tạo khai thác hoàn toàn tự động do các kỹ thuật phân tích chương trình tiên tiến nhất có nhiều hạn chế

Exploitability assessment of vulnerabilities is important for both defenders and attackers. The ultimate way to assess the exploitability is crafting a working exploit. However, it usually takes tremendous hours and significant manual efforts. To address this issue, automated techniques can be adopted. Existing solutions usually explore in depth the *crashing paths*, i.e., paths taken by proof-of-concept (PoC) inputs triggering vulnerabilities, and assess exploitability by finding *exploitable states* along the paths. However, exploitable states do not always exist in crashing paths. Moreover, existing solutions heavily rely on symbolic execution and are not scalable in path exploration and exploit generation.

Đánh giá khả năng khai thác của các lỗ hổng là một việc quan trọng đối với cả người bảo vệ và kẻ tấn công. Cách cuối cùng để đánh giá khả năng khai thác là tạo ra một khai thác đang hoạt động. Tuy nhiên, nó thường mất rất nhiều thời gian và nỗ lực cho việc làm thủ công. Để giải quyết vấn đề này, các kỹ thuật tự động có thể được áp dụng. Các giải pháp hiện tại thường khám phá sâu *các* ***đường dẫn sự cố*** (crashing path), tức là các đường dẫn được thực hiện bởi các đầu vào kiểm tra tính khả thi hay bằng chứng khái niệm (PoC) để kích hoạt các lỗ hổng và đánh giá khả năng khai thác bằng cách tìm *các* ***trạng thái có thể khai thác*** dọc theo các đường dẫn đó. Tuy nhiên, các trạng thái có thể khai thác không phải lúc nào cũng tồn tại trong các đường dẫn sự cố (crashing paths). Hơn nữa, các giải pháp hiện tại phụ thuộc rất nhiều vào việc thực thi tượng trưng (symbolic) và không thể mở rộng trong quá trình khám phá đường dẫn và tạo khai thác.

In this paper, we propose a novel solution to generate exploit for userspace programs or facilitate the process of crafting a kernel UAF exploit. Technically, we utilize oriented fuzzing to explore diverging paths from vulnerability point. For userspace programs, we adopt a control-flow stitching solution to stitch crashing paths and diverging paths together to generate exploit. For kernel UAF, we leverage a lightweight symbolic execution to identify, analyze and evaluate the system calls valuable and useful for exploiting vulnerabilities.

Trong bài báo này, người ta đề xuất một giải pháp mới để tạo khai thác cho các chương trình không gian người dùng hoặc tạo điều kiện thuận lợi cho quá trình tạo khai thác nhân UAF (kernel **Use-After-Free**). Về mặt kỹ thuật, họ sử dụng phương pháp oriented fuzzing (**làm mờ có định hướng**) để khám phá các đường dẫn khác nhau từ điểm dễ bị tổn thương (**vulnerability point**). Đối với các chương trình không gian người dùng, họ áp dụng giải pháp ghép luồng điều khiển để ghép các đường dẫn sự cố và các đường dẫn phân kỳ (**crashing paths + diverging paths**) lại với nhau để tạo ra khai thác. Đối với nhân UAF, họ tận dụng một thực thi tượng trưng nhẹ để xác định, phân tích và đánh giá các cuộc gọi hệ thống có giá trị và hữu ích để khai thác các lỗ hổng.

We have developed a prototype system and evaluated it on a set of 19 CTF (capture the flag) programs and 15 realworld Linux kernel UAF vulnerabilities. Experiment results showed it could generate exploit for most of the userspace test set, and it could also facilitate security mitigation bypassing and exploitability evaluation for kernel test set.

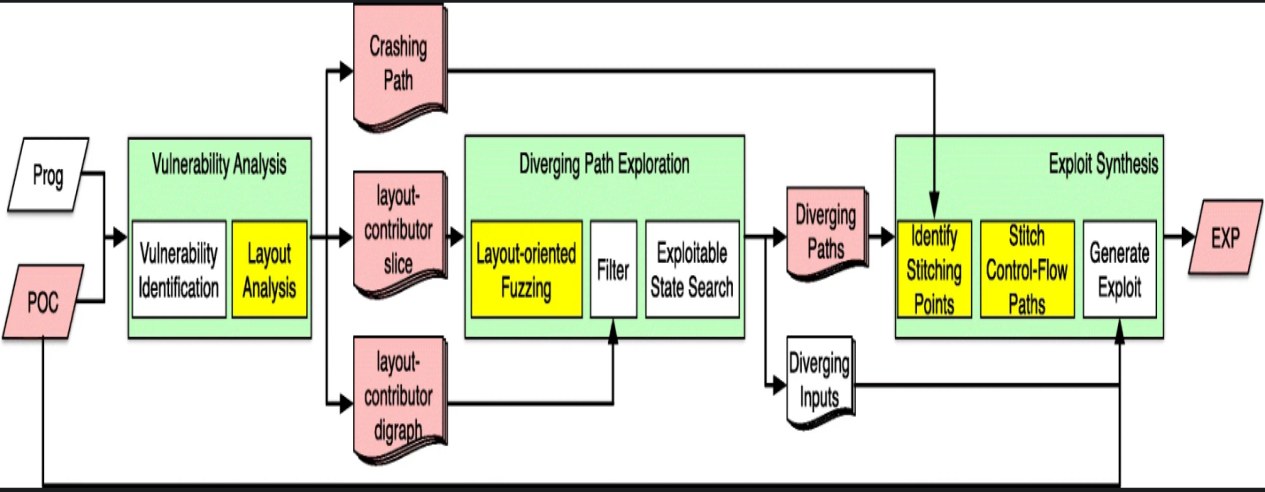
Họ đã phát triển một hệ thống nguyên mẫu và đánh giá nó trên một bộ gồm 19 chương trình CTF (flags) và 15 lỗ hổng UAF của nhân Linux trong thế giới thực. Kết quả thử nghiệm cho thấy nó có thể tạo ra khai thác cho hầu hết bộ thử nghiệm không gian người dùng và nó cũng có thể tạo điều kiện bỏ qua giảm thiểu bảo mật và đánh giá khả năng khai thác cho bộ thử nghiệm nhân

**-> Kết quả:** Họ đã xây dựng một  khung Revery, có thể tạo ra các khai thác chiếm quyền điều khiển luồng điều khiển đang hoạt động cho các chương trình không gian người dùng. Họ cũng xây dựng một khung FUZE , có thể đánh giá khả năng khai thác các lỗ hổng Use-after-free của kernel.

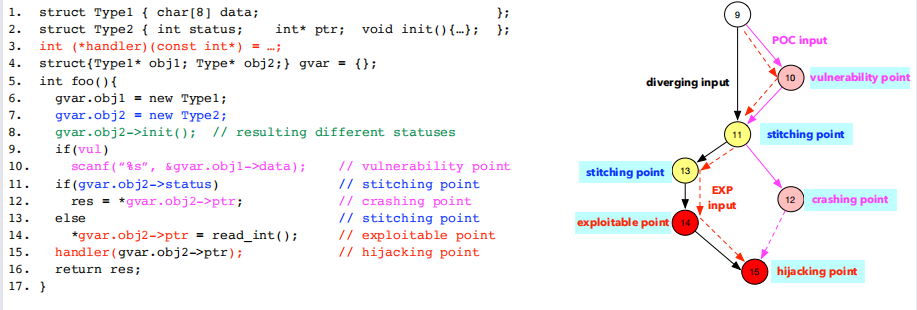
Mô hình:

a novel solution Revery, to solve the *exploit derivability* issue and assess the exploitability of heap-based vulnerabilities. At the high level, Revery analyzes the vulnerability in detail, utilizes the vulnerability information to guide a fuzzer rather than symbolic execution to explore diverging paths and search for exploitable states, then synthesizes exploitation paths by stitching the crashing path and diverging path, and finally generates inputs to trigger both the vulnerability and exploitable states. As shown in Fig. [2](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig2), it has three major components.

Giải pháp mới Revery: để giải quyết vấn đề *về* ***khả năng khai thác*** và đánh giá khả năng khai thác của các lỗ hổng dựa trên **heap**. Ở cấp độ cao, Revery phân tích lỗ hổng một cách chi tiết, sử dụng thông tin lỗ hổng để hướng dẫn một **fuzzer** thay vì thực thi tượng trưng (symbolic) để khám phá các đường dẫn phân kỳ và tìm kiếm các trạng thái có thể khai thác, sau đó tổng hợp các đường dẫn khai thác bằng cách ghép đường dẫn sự cố và đường dẫn phân kỳ, và cuối cùng tạo ra đầu vào để kích hoạt cả trạng thái dễ bị tổn thương và có thể khai thác. Như thể hiện trong Hình  [2](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig2) , nó có ba thành phần chính.



Hinh 2: Vulnerbility Analysis, Diverging Path Exploitation, Exploit Synthesis

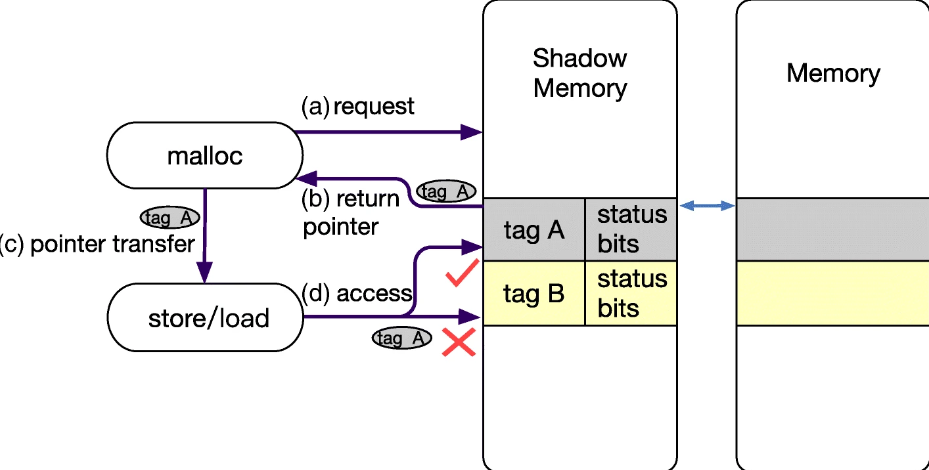


Hình 1: Ví dụ về heap overflow

- Vulnerability Analysis (include: Vulnerability identification, Layout analysis)

Revery first analyzes the vulnerability in detail, similar to existing AEG solutions. It uses dynamic analysis to test target application with the provided PoC input. More specifically, it tracks the states of each pointer and memory object, and catches security violations along the crashing path. It could thus identify the vulnerability point, e.g., line 10 in Fig. [1](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig1).

More importantly, it identifies *exceptional objects* corrupted by the vulnerability, e.g., obj2 in the example. Revery also identifies the exceptional object’s indexing objects, which could be used to locate the exceptional object, e.g., the global variable gvar in the example. Moreover, it retrieves *layout-contributor* instructions from the execution trace, which create the exceptional and indexing objects and set up their point-to relationships, e.g., line 7 in the example. These objects and contributor instructions are used to construct a layout-contributor digraph.



Trước tiên , **Revery** phân tích lỗ hổng một cách chi tiết, tương tự như các giải pháp AEG hiện có. Nó sử dụng phân tích động để kiểm tra ứng dụng mục tiêu với đầu vào PoC được cung cấp. Cụ thể hơn, nó theo dõi trạng thái của từng con trỏ và đối tượng bộ nhớ, đồng thời phát hiện các vi phạm bảo mật dọc theo đường dẫn sự cố. Do đó, nó có thể xác định điểm dễ bị tổn thương, ví dụ: dòng 10 trong Hình  [1](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig1) .

Quan trọng hơn, nó xác định *các đối tượng đặc biệt* bị lỗ hổng làm hỏng, ví dụ: **obj2**trong ví dụ. **Revery** cũng xác định các đối tượng lập chỉ mục của đối tượng đặc biệt, có thể được sử dụng để định vị đối tượng đặc biệt, ví dụ: biến toàn cục **gvar** trong ví dụ. Ngoài ra, nó truy xuất các hướng dẫn *của* ***trình đóng góp bố cục*** (layout-contributor) từ dấu vết thực thi, tạo ra các đối tượng đặc biệt và lập chỉ mục đối tượng, cùng thiết lập các mối quan hệ điểm - tới của chúng, ví dụ: dòng 7 trong ví dụ. Các đối tượng này và hướng dẫn của người đóng góp được sử dụng để xây dựng sơ đồ bố cục-người đóng góp (layout-contributor digraph).

Gồm:

+ Thẻ nhớ (memory tags)

Mỗi đối tượng heap và con trỏ được gắn với một thẻ nhớ, cho biết nguồn gốc của nó. Thẻ này sẽ được tạo duy nhất khi một đối tượng được tạo và truyền tới các con trỏ của đối tượng và các con trỏ liên quan khác dưới dạng nhãn hiệu (nhãn phân tích). Ngoài ra, mỗi đối tượng heap được liên kết với một trạng thái, nghĩa là chưa được khởi tạo, bận hoặc rảnh , đại diện cho ba trạng thái trong vòng đời của nó, nghĩa là được phân bổ nhưng không được khởi tạo, được khởi tạo và đang được sử dụng hoặc được giải phóng. Điều đáng chú ý là, một vùng bộ nhớ được giải phóng có thể được phân bổ cho các đối tượng mới và trạng thái bộ nhớ cũng như thẻ của nó sẽ thay đổi tương ứng.

Trong một số trường hợp góc, nhà phát triển có thể sử dụng con trỏ của một đối tượng để lấy con trỏ của đối tượng khác, bằng một phép toán số học. Nó sẽ truyền sai thẻ của đối tượng đầu tiên sang con trỏ thứ hai. May mắn thay, điều này hiếm khi xảy ra đối với các đối tượng heap, vì độ lệch giữa các đối tượng heap không cố định. Ngoại lệ duy nhất là các chức năng quản lý heap, có thể kiểm tra các đối tượng lân cận theo cách này, bất kể các đối tượng này có nội dung như thế nào. Vì vậy, Revery sẽ vô hiệu hóa việc truyền và xác thực thẻ cho các chức năng đặc biệt này. Điều đáng chú ý là, tối ưu hóa này chỉ dành cho dẫn xuất con trỏ chéo đối tượng. Revery hỗ trợ điều khiển con trỏ trong đối tượng như bình thường.

+ Quy tắc bảo mật (security rules)

Đối với mỗi hướng dẫn truy cập bộ nhớ heap (tải và lưu trữ), chúng ta có thể lấy thẻ của con trỏ tag\_ptr và thẻ của vùng bộ nhớ đích tag\_obj, cùng với trạng thái status\_obj . Việc truy cập bộ nhớ không được vi phạm các quy tắc bảo mật sau:

**V1: truy cập các đối tượng dự định** : Các hướng dẫn chỉ nên truy cập các đối tượng dự định, tức là tag\_obj và tag\_ptr phải khớp.

**V2: đọc các đối tượng bận** : Hướng dẫn tải không được truy cập vào bộ nhớ được giải phóng hoặc chưa được khởi tạo, nghĩa là status\_obj phải bận .

**V3: ghi các đối tượng còn sống** : Lệnh lưu trữ không được truy cập vào bộ nhớ đã giải phóng, nghĩa là status\_obj phải bận hoặc chưa được khởi tạo .

Bất kỳ vi phạm các quy tắc này sẽ gây ra một lỗ hổng. Ví dụ: truy cập bộ nhớ bị tràn bộ đệm sẽ vi phạm quy tắc V1. Một lỗ hổng biến chưa được khởi tạo sẽ vi phạm quy tắc V2. Lỗ hổng sử dụng Use-After-Free (UAF) có thể vi phạm quy tắc V1, V2 hoặc V3. Nếu bộ nhớ của đối tượng được giải phóng chưa bị chiếm bởi các đối tượng khác, thì quyền truy cập đọc vào đối tượng đó sẽ vi phạm V2 và quyền ghi vào đối tượng đó sẽ vi phạm V3. Nếu bộ nhớ của đối tượng được giải phóng bị chiếm dụng, thì thẻ của nó sẽ thay đổi và mọi quyền truy cập vào nó thông qua con trỏ lơ lửng ban đầu sẽ vi phạm quy tắc V1.

### phân tích bố cục

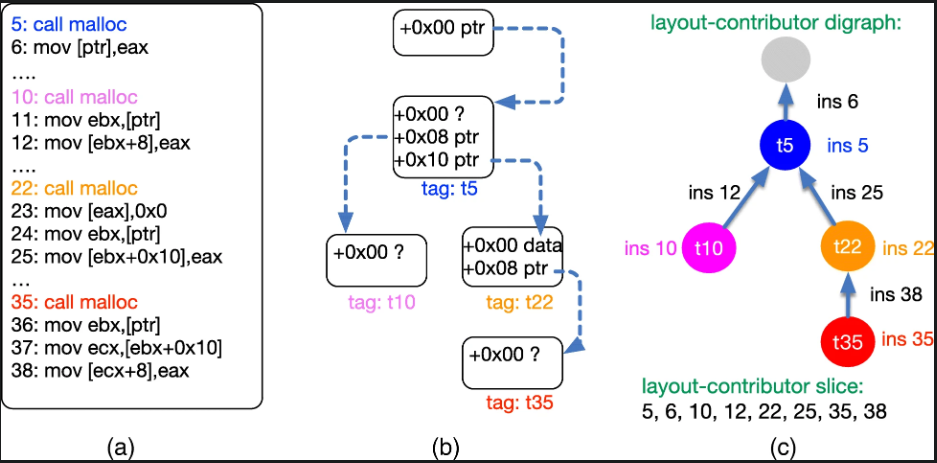
Revery phân tích thêm bố cục đối tượng để mô tả trạng thái dễ bị tổn thương và truy xuất các hướng dẫn góp phần vào trạng thái.

#### **Bố cục đối tượng liên quan đến lỗ hổng (Vulnerability-related Object Layout)**

Mỗi lỗ hổng dựa trên heap (bao gồm tràn heap và UAF) có liên quan đến một *đối tượng ngoại lệ* có nội dung bị (hoặc sẽ) bị lỗ hổng làm hỏng. Các hoạt động tiếp theo trên các đối tượng này có thể dẫn cỗ *máy kỳ lạ (weird machine)* đến trạng thái có thể khai thác được.

Giả sử điểm dễ bị tổn thương sử dụng con trỏ có thẻ tag\_ptr để truy cập đối tượng mục tiêu có thẻ tag\_obj . Nếu đó là quyền truy cập ghi, đối tượng có thẻ tag\_obj là đối tượng đặc biệt, sẽ bị hỏng bởi quyền truy cập ghi này. Nếu đó là quyền truy cập đọc và lỗ hổng này là UAF, thì đối tượng có thẻ tag\_ptr là đối tượng đặc biệt, đối tượng này sẽ bị hỏng do phân bổ đối tượng mới chiếm cùng bộ nhớ. Revery hiện không hỗ trợ tốt các loại vi phạm quyền truy cập đọc khác.

Hơn nữa, Revery cũng theo dõi tất cả các đối tượng lập chỉ mục có thể được sử dụng để định vị các đối tượng đặc biệt. Các đối tượng đặc biệt và các đối tượng lập chỉ mục này được kết nối với mối quan hệ điểm tới. Kết quả là Revery có thể nhận được một sơ đồ của các đối tượng, được ký hiệu là *sơ đồ bố cục* . Sơ đồ bố cục này mô tả trạng thái dễ bị tổn thương ở một mức độ nào đó. Hình  [4](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig4) b cho thấy một sơ đồ bố cục ví dụ.



1. Sample execution
2. Layout digraph
3. Layout-contributor

#### **Mã liên quan đến lỗ hổng(Vulnerability-related code)**

Như đã nói ở trên, *cỗ máy kỳ lạ* phải vào các trạng thái ban đầu cụ thể, bao gồm cả trạng thái dễ bị tấn công. Vì vậy, cần phải chuẩn bị một bố cục đối tượng tương tự như của lỗ hổng, cả trong các đường phân kỳ và đường khai thác. Vì vậy, các hướng dẫn góp phần vào bố cục là rất quan trọng.

Có thể thấy rằng, hai loại hoạt động sau đây có thể đóng góp vào bố cục đối tượng: (1) cấp phát bộ nhớ để tạo đối tượng mới và (2) lưu trữ các hoạt động gán trường của đối tượng với một con trỏ tới đối tượng khác. Do đó, Revery có thể truy xuất tất cả các hoạt động của trình đóng góp như vậy, thao tác này hoạt động trên các đối tượng trong sơ đồ bố cục và tạo một *sơ đồ kết hợp bố cục* (**layout-contributor digraph**).

Cụ thể hơn, mỗi nút trong sơ đồ này là một đối tượng đặc biệt hoặc một đối tượng lập chỉ mục, với một thuộc tính của hướng dẫn tạo đối tượng và thẻ bộ nhớ. Mỗi cạnh trong sơ đồ mô tả mối quan hệ điểm tới giữa hai đối tượng, với một thuộc tính của lệnh gán con trỏ. Với một đối tượng mục tiêu đặc biệt, chúng ta có thể sử dụng kỹ thuật cắt lát ngược để xây dựng sơ đồ này. Hình  [4](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig4) c cho thấy một sơ đồ bố trí-người đóng góp ví dụ. Sơ đồ ghép này có dạng đơn giản hơn, được gọi là *lát cắt bố cục-người đóng góp* (**layout-contributor slice**), là một chuỗi các hướng dẫn của người đóng góp theo thứ tự thực hiện.

- Diverging Path Exploitation

Revery searches for exploitable states in diverging paths, to solve the *exploit desirability* issue. Rather than using symbolic execution, it employs fuzzing.

First, it employs a novel *layout-oriented fuzzing* solution to explore diverging paths. To facilitate exploit generation, only diverging paths with memory layouts similar as the PoC input’s will be explored. So, it drives a fuzzer to explore paths close to the crashing path, in a similar way as directed fuzzing solutions ([Böhme et al. 2017](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "ref-CR9" \o "Böhme, M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2329–2344.. ACM.)). But instead of using the full crashing path, it uses the aforementioned *layout-contributor* instructions as the fuzzer’s guidance. The fuzzer could thus produce diverging inputs to exercise the diverging paths (e.g., 9->11->13->14 in the figure) with proper memory layouts.

Then, Revery searches for exploitable states in the diverging paths. Several heuristics are used to identify exploitable states. For example, if a memory store operation’s destination is controlled by the corrupted object, e.g., line 14, it is an exploitable state.

Furthermore, Revery also searches for hijacking points in these diverging paths. Hijacking points sometimes are not obvious. So Revery uses some heuristics to infer hijacking points. For example, line 15 in the figure is a second-order hijacking point, which could be enabled if line 14 overwrites the global function pointer.

Revery tìm kiếm các trạng thái có thể khai thác trong các đường dẫn khác nhau, để giải quyết vấn đề *về khả năng khai thác* . Thay vì sử dụng thực thi tượng trưng, nó sử dụng fuzzing.

Đầu tiên, nó sử dụng một giải pháp *làm mờ định hướng bố cục* (**layout-oriented fuzzing**) mới để khám phá các đường phân kỳ. Để tạo điều kiện thuận lợi cho việc tạo khai thác, chỉ các đường dẫn phân kỳ có bố cục bộ nhớ tương tự như đầu vào PoC mới được khám phá. Vì vậy, nó điều khiển một bộ làm mờ để khám phá các đường dẫn gần với đường dẫn sự cố, theo cách tương tự như các giải pháp làm mờ có hướng. Nhưng thay vì sử dụng toàn bộ đường dẫn sự cố, nó sử dụng các hướng dẫn *của trình đóng góp bố* cục (layout-contributor) đã nói ở trên làm hướng dẫn của bộ làm mờ. Do đó, bộ làm mờ có thể tạo ra các đầu vào phân kỳ để thực hiện các đường dẫn phân kỳ (ví dụ: 9->11->13->14 trong hình) với cách bố trí bộ nhớ phù hợp.

Sau đó, Revery tìm kiếm các trạng thái có thể khai thác trong các đường dẫn phân kỳ. Một số Heuristic được sử dụng để xác định các trạng thái có thể khai thác. Ví dụ: nếu đích của hoạt động lưu trữ bộ nhớ được kiểm soát bởi đối tượng bị hỏng, ví dụ: dòng 14, thì đó là trạng thái có thể khai thác.

Hơn nữa, Revery cũng tìm kiếm các điểm chiếm quyền điều khiển trong các đường dẫn phân kỳ này. Hijacking point đôi khi không rõ ràng. Vì vậy , Revery sử dụng một số kinh nghiệm để suy ra các điểm chiếm quyền điều khiển. Ví dụ: dòng 15 trong hình là điểm chiếm quyền điều khiển cấp hai, điểm này có thể được kích hoạt nếu dòng 14 ghi đè lên con trỏ hàm toàn cục.

### lựa chọn thay thế

Các giải pháp tạo khai thác tự động hiện có, ví dụ: AEG và Mayhem, chủ yếu dựa vào việc thực thi biểu tượng để khám phá đường dẫn sự cố hoặc đường dẫn có thể truy cập từ điểm dễ bị tổn thương, nhằm tìm kiếm các trạng thái có thể khai thác dọc theo con đường khám phá. Tuy nhiên, việc thực thi biểu tượng có một số thách thức nghiêm trọng và không phù hợp để khám phá đường dẫn hoặc tìm kiếm trạng thái có thể khai thác.

Đầu tiên, nó không thể mở rộng trong khám phá đường dẫn. Nó bị vấn đề bùng nổ đường dẫn do các nhánh và vòng lặp trong chương trình gây ra. Ngay cả khi phân tích một đường dẫn, nó cũng tiêu tốn quá nhiều tài nguyên. Hơn nữa, các ràng buộc tượng trưng thường quá phức tạp để giải quyết.

Thứ hai, việc thực thi tượng trưng có thể bị mù đối với một số trạng thái có thể khai thác. Nó phải cụ thể hóa một số giá trị tượng trưng trong quá trình khám phá, bằng cách thêm các ràng buộc bổ sung của việc gán giá trị cụ thể hóa. Không thể thử tất cả các giá trị được cụ thể hóa ứng cử viên, do đó bỏ lỡ một số giá trị nhất định và gây ra tình trạng mù quáng đối với một số trạng thái có thể khai thác.

Ví dụ, nó sẽ cụ thể hóa các đối số tượng trưng của *cấp phát bộ nhớ* trong một đường dẫn, để mô hình hóa các trạng thái bộ nhớ và khám phá các đường dẫn phụ sau. Có khả năng chỉ một số lượng nhỏ phân bổ có thể gây ra trạng thái có thể khai thác. Vì vậy, việc phân bổ bộ nhớ cụ thể hóa có thể dẫn đến trạng thái không thể khai thác.

Hơn nữa, nó cũng sẽ cụ thể hóa *các chỉ mục tượng trưng* trong các thao tác truy cập bộ nhớ, vì nếu không thì kết quả của các thao tác là không xác định. Tương tự, nó cũng có thể dẫn đến các trạng thái không thể khai thác được.

### Fuzzing định hướng bố cục (Layout-oriented Fuzzing)

Revery chỉ sử dụng fuzzing để khám phá các đường dẫn khác nhau và tìm kiếm các trạng thái có thể khai thác. Như đã trình bày trong phần phát hiện lỗ hổng, làm mờ hiệu quả hơn so với thực thi tượng trưng trong việc khám phá các đường dẫn và trạng thái chương trình. Vì vậy, có khả năng fuzzing cũng có thể giúp tìm ra các đường dẫn khác nhau và các trạng thái có thể khai thác nhanh hơn.

Revery sử dụng một giải pháp làm mờ định hướng bố cục mới được hướng dẫn bởi sơ đồ của người đóng góp bố cục, để khám phá các đường dẫn phân kỳ tạo ra các bố cục bộ nhớ tương tự như lỗ hổng bảo mật.

#### **Thiết kế**

Revery mở rộng **AFL fuzzer** hướng dẫn vùng phủ sóng phổ biến để thực hiện fuzzing. Thay vì chỉ dựa vào phạm vi mã để hướng dẫn khám phá đường dẫn, Revery sử dụng sơ đồ đóng góp bố cục làm hướng dẫn để điều chỉnh hướng khám phá và đột biến.

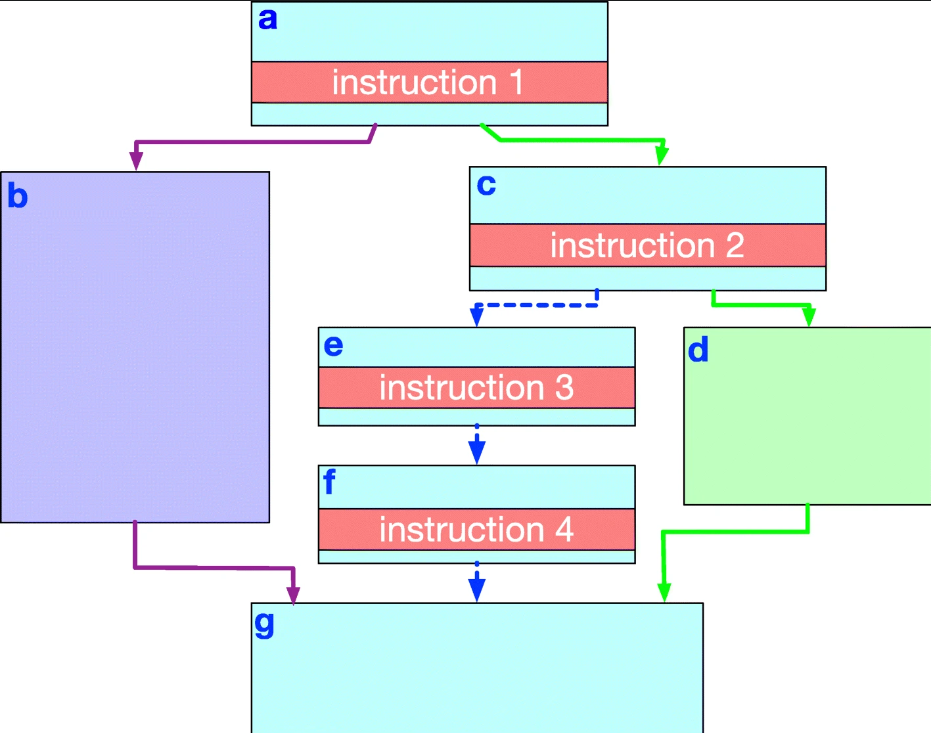
Tương tự như làm mờ theo hướng (directed fuzzing), Revery điều khiển bộ làm mờ để khám phá các đường dẫn gần với đường dẫn sự cố. Nó chỉ nhằm mục đích khớp các hướng dẫn trong phần trình đóng góp bố cục và bỏ qua các hướng dẫn khác trong đường dẫn sự cố. Các lựa chọn thiết kế được thực hiện từ ba trực giác sau đây.

Để đơn giản, họ giới thiệu một số thuật ngữ. Đưa ra một đầu vào *I a* , nó có thể đạt được một số hướng dẫn của người đóng góp bố cục (có thể không theo thứ tự như phần hướng dẫn). Danh sách đầy đủ các hướng dẫn như vậy được ký hiệu là *L a* và dãy con chung dài nhất (LCS) của nó với lát hướng dẫn mục tiêu được ký hiệu là *P a* .

**Trực giác 1:** Một đầu vào chạm vào tất cả các hướng dẫn của trình đóng góp bố cục, theo cùng thứ tự với lát hướng dẫn, có thể tạo ra một bố cục bộ nhớ tương tự như lỗ hổng. Các hướng dẫn của người đóng góp bố cục chịu trách nhiệm tạo đối tượng đặc biệt của lỗ hổng bảo mật và các đối tượng lập chỉ mục của nó, cũng như thiết lập mối quan hệ điểm tới giữa chúng. Vì vậy, một đầu vào chạm vào toàn bộ lát trình đóng góp bố cục có thể có thể tạo ra các bố cục bộ nhớ tương tự.

**Trực giác 2:** Một đầu vào chạm vào chuỗi con dài hơn của phần hướng dẫn có nhiều khả năng dẫn đến các đầu vào chạm vào toàn bộ phần.

Nói cách khác, nếu LCS *P a* của đầu vào *I a* dài hơn LCS *P b của I b* thì đầu vào *I a* tốt hơn *I b* . Như được hiển thị trong Hình  [5](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig5) , giả sử lát cắt mục tiêu nằm trong đường dẫn a=>c=>e=>f, thì một đầu vào thực hiện đường dẫn a=>c=>d tốt hơn các đầu vào khác thực hiện a=>b. Các đột biến tiếp theo trên đầu vào này có thể dẫn đến các đầu vào chạm vào lát hướng dẫn đầy đủ nhanh hơn.

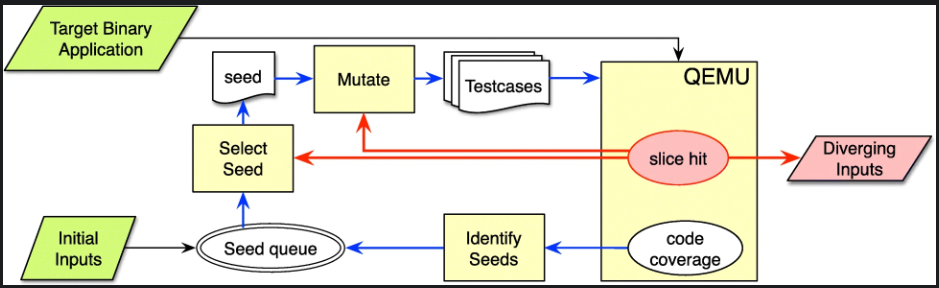


**Trực giác 3:** Các đầu vào chạm vào ít hướng dẫn trình đóng góp bố cục hơn có nhiều khả năng gây ra ít rắc rối hơn cho việc tạo khai thác tiếp theo.

Nói cách khác, đối với hai đầu vào *I a* và *I b* , nếu LCS *P a* và *P b* của chúng có cùng độ dài, nhưng danh sách lệnh của bộ đóng góp bố cục *L a* dài hơn *L b* , thì đầu vào *I b* tốt hơn *I một* . Trong trường hợp này, đầu vào *I a* có nhiều hướng dẫn cộng tác viên trùng lặp hoặc không theo thứ tự hơn *I b* , điều này có thể gây ra việc tạo đối tượng dư thừa hoặc xây dựng bố cục, khiến bố cục bộ nhớ quá phức tạp để khai thác

#### **Chi tiết triển khai**

Revery mở rộng **AFL fuzzer** phổ biến. Như được hiển thị trong Hình  [6](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig6) , AFL áp dụng một vòng lặp liên tục để khám phá các đường dẫn. Nó (1) giữ một hàng các trường hợp thử nghiệm tốt, tức là hạt giống; và (2) chọn một hạt giống từ hàng đợi; và sau đó (3) biến đổi hạt giống để nhận được một loạt các trường hợp thử nghiệm mới, sau đó (4) chạy chương trình nhị phân đích với các trường hợp thử nghiệm được tạo trong **QEMU** và theo dõi mức độ phù hợp, sau đó (5) xác định các hạt giống dựa trên thông tin mức độ phù hợp. Revery sửa đổi AFL theo hai khía cạnh sau



**Theo dõi Slice Hit Count:**Revery thêm một bộ đệm HIT bổ sung trong bộ nhớ dùng chung giữa QEMU và trình điều khiển bộ làm mờ, ngoài bitmap hiện có được sử dụng để theo dõi phạm vi mã. HIT[0] được sử dụng để theo dõi số lần chạm vào lát cắt, trong khi HIT[i] được sử dụng để theo dõi xem hướng dẫn *thứ i* trong lát cắt hướng dẫn có được chạm vào hay không.

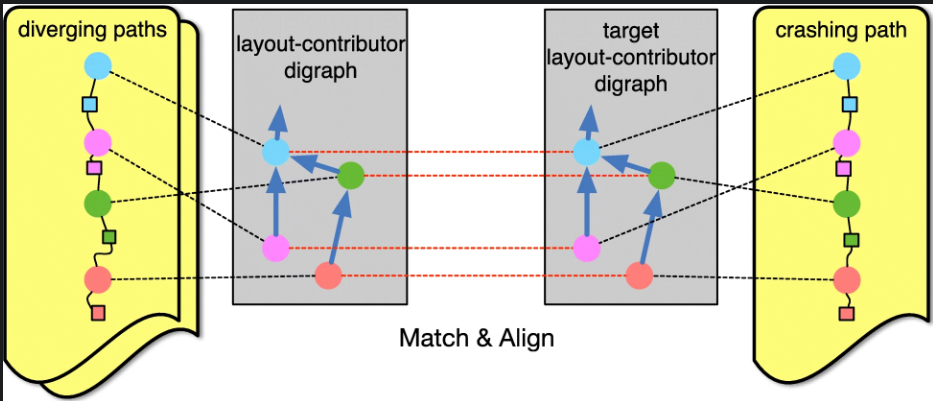
Cụ thể hơn, mỗi khi một lệnh của trình đóng góp bố cục được thực thi, QEMU sẽ tăng số lần chạm vào lát cắt HIT[0] . Nếu lệnh này là lệnh thứ *n* (n>=1) trong lát hướng dẫn, thì QEMU sẽ đặt HIT[n] khi và chỉ khi HIT[n-1] đã được đặt. Bằng cách này, trình điều khiển fuzzer có thể nhận được số lần chạm vào lát cắt trong HIT[0] và LCS của lát cắt hướng dẫn trong HIT[1:N] .

**Điều chỉnh các hướng làm mờ:**Revery sửa đổi trình điều khiển bộ làm mờ để sử dụng thông tin lần truy cập lát cắt đã thu thập. Về cơ bản, nó thay đổi một chút thuật toán lựa chọn hạt giống. Khi chọn một hạt giống từ hàng đợi để đột biến, trước tiên, nó ưu tiên các hạt có LCS dài hơn, như đã thảo luận trong Trực giác 2. Sau đó, trong số các hạt có LCS có cùng độ dài, nó ưu tiên các hạt có số lần chạm lát ít hơn, như được đề xuất trong Trực giác 3. Cuối cùng, nó ưu tiên các hạt có kích thước nhỏ hơn và thời gian thực hiện nhanh hơn, giống như chính sách mặc định của AFL.

### Lọc đầu vào phân kỳ

Với tính năng làm mờ định hướng bố cục, Revery có thể tìm thấy các đầu vào phân kỳ có thể kích hoạt lát cắt trình đóng góp bố cục giống như đầu vào PoC. Tuy nhiên, không giống như sơ đồ của người đóng góp bố cục, các ràng buộc luồng dữ liệu bị thiếu trong lát cắt của người đóng góp bố cục. Vì vậy, các đầu vào phân kỳ đôi khi không khớp với sơ đồ của bộ đóng góp bố cục đích được tạo từ đường dẫn sự cố. Do đó, Revery thực hiện thêm một bước để cô lập các đầu vào phân kỳ có thể khớp với biểu đồ ghép nối của trình đóng góp bố cục đích.

Nói chung, trước tiên, nó căn chỉnh đường dẫn phân kỳ với đường dẫn sự cố và định vị các hướng dẫn chịu trách nhiệm tạo đối tượng đặc biệt. Sau đó, nó xây dựng một sơ đồ đóng góp bố cục mới của đối tượng đặc biệt từ đường dẫn phân kỳ bằng cách cắt ngược, theo cách tương tự như đường dẫn va chạm. Cuối cùng, nó đối sánh sơ đồ ghép mới này với sơ đồ ghép đích, bằng cách so sánh thẻ nhớ của mỗi nút và địa chỉ của lệnh tạo của nó trong hai sơ đồ. Hình  [7](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig7) cho thấy một ví dụ về cách thức hoạt động của trận đấu.



Nếu hai chữ ghép này không khớp, thì đầu vào phân kỳ này sẽ bị loại bỏ. Mặt khác, đầu vào phân kỳ được giữ lại. Hơn nữa, các nút của hai chữ ghép này (nghĩa là các đối tượng heap) sẽ được căn chỉnh tương ứng, cũng như các thẻ bộ nhớ của tất cả các nút. Vì vậy, chúng ta có thể suy ra bản sao của từng đối tượng giữa đường dẫn phân kỳ và đường dẫn sụp đổ, cho phép phân tích chung hơn nữa trên hai đường dẫn này.

### Tìm kiếm trạng thái có thể khai thác

Ngay cả khi các đường dẫn phân kỳ có bố cục tương tự như lỗ hổng, thì không phải tất cả chúng đều có thể khai thác được. Revery tiếp tục loại bỏ các đường dẫn phân kỳ không có trạng thái có thể khai thác.

#### **trạng thái có thể khai thác**

Đối tượng đặc biệt có thể ảnh hưởng đến các đối tượng khác và đôi khi sẽ được sử dụng trực tiếp hoặc gián tiếp trong một số hoạt động nhạy cảm. Trạng thái chương trình do các hoạt động nhạy cảm này được biểu thị là trạng thái có thể khai thác.

Trong bài báo này, họ chủ yếu xem xét hai loại hoạt động nhạy cảm (có thể khai thác), tức là *ghi bộ nhớ* và *gọi gián tiếp* . Ví dụ: nếu địa chỉ đích của quá trình ghi bộ nhớ bị ảnh hưởng bởi đối tượng ngoại lệ, thì kẻ tấn công có thể kiểm soát nơi ghi và gây ra AAW (ghi địa chỉ tùy ý), tức là trạng thái có thể khai thác thường được sử dụng trong thực tế. Nếu những kẻ tấn công có thể ảnh hưởng đến mục tiêu của các lệnh gọi gián tiếp, bao gồm các lệnh gọi chức năng ảo và lệnh *jmp* gián tiếp , v.v., thì chúng có thể chiếm đoạt luồng điều khiển. Ngoài ra, Revery cung cấp một mẫu để các chuyên gia mở rộng định nghĩa về các điểm có thể khai thác, ví dụ: các hoạt động khởi động cuộc tấn công hủy liên kết.

#### **Tìm kiếm trạng thái có thể khai thác**

Do đó, vấn đề này trở thành việc xác định các lệnh nhạy cảm có toán hạng bị ảnh hưởng bởi các đối tượng đặc biệt. Phân tích vết bẩn là một giải pháp đơn giản.

Revery đánh dấu mỗi thao tác tạo đối tượng là một nguồn nhiễm độc và gắn một nhãn nhiễm độc duy nhất cho hoạt động đó. Mỗi hoạt động truyền tất cả các nhãn mờ của toán hạng nguồn đến đích. Tại mỗi lệnh nhạy cảm (nghĩa là ghi bộ nhớ hoặc lệnh gọi hàm), nhãn vết của địa chỉ đích sẽ được kiểm tra xem chúng có chứa nhãn vết của đối tượng ngoại lệ hay không. Nếu có, thì hướng dẫn nhạy cảm này có thể khai thác được.

Exploit Synthesis (Khai thác tổng hợp)

PoC **stitching**

Once an exploitable state (together with a diverging input) in a diverging path is found, Revery will try to synthesize a new input to trigger both the vulnerability and the exploitable state. In general, it first finds the stitching points in the crashing path (e.g., line 11) and in the diverging path (e.g., line 13), with some specific data flow analysis.

Then it utilizes a lightweight symbolic execution to explore potential sub-paths between these two stitching points (e.g., 11->13), and stitch the crashing path with the diverging path to synthesize an exploitation path (e.g., 9->10->11->13->14->15), and finally generate inputs to exercise the exploitation paths. Several optimizations are deployed to make the symbolic execution lightweight.

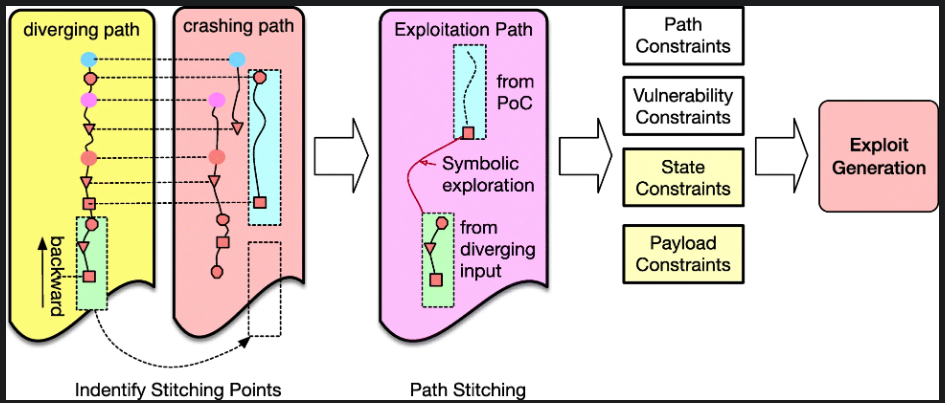
Therefore, Revery could produce *EXP inputs* able to trigger both vulnerabilities and exploitable states. It could help experts to quickly generate working exploits. In certain cases, Revery is able to directly generate exploits. For example, Revery could generate an exploit input to hijack the control flow, by utilizing the exploitable state at line 14 to overwrite the global function pointer handler

Sau khi tìm thấy trạng thái có thể khai thác (cùng với đầu vào phân kỳ) trong đường dẫn phân kỳ, Revery sẽ cố gắng tổng hợp đầu vào mới để kích hoạt cả lỗ hổng và trạng thái có thể khai thác. Nói chung, trước tiên nó tìm các điểm nối trong đường dẫn sự cố (ví dụ: dòng 11) và trong đường dẫn phân kỳ (ví dụ: dòng 13), với một số phân tích luồng dữ liệu cụ thể.

Sau đó, nó sử dụng một thực thi ký hiệu nhẹ để khám phá các đường dẫn con tiềm năng giữa hai điểm ghép này (ví dụ: 11->13 ) và ghép đường dẫn sự cố với đường dẫn phân kỳ để tổng hợp đường dẫn khai thác (ví dụ: 9->10-> 11->13->14->15 ), và cuối cùng tạo đầu vào để thực hiện các đường dẫn khai thác. Một số tối ưu hóa được triển khai để làm cho việc thực thi tượng trưng trở nên nhẹ nhàng.

Do đó, Revery có thể tạo ra *đầu vào EXP* có thể kích hoạt cả lỗ hổng và trạng thái có thể khai thác. Nó có thể giúp các chuyên gia nhanh chóng tạo ra các khai thác đang hoạt động. Trong một số trường hợp nhất định, Revery có thể trực tiếp tạo ra các khai thác. Ví dụ: Revery có thể tạo đầu vào khai thác để chiếm đoạt luồng điều khiển, bằng cách sử dụng trạng thái có thể khai thác ở dòng 14 để ghi đè lên trình xử lý con trỏ hàm toàn cầu .

Hình  [8](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig8) cho thấy quy trình chung của quá trình tổng hợp khai thác. Trong thực tế, trước tiên, nó xác định các điểm ghép nối, sau đó khám phá các đường dẫn phụ giữa các điểm ghép nối và đường dẫn khai thác tổng hợp, cuối cùng giải quyết các ràng buộc liên quan để tạo ra các khai thác đang hoạt động.



### Xác định các điểm khâu

Trước tiên, họ giới thiệu cách Revery xác định các điểm nối trong cả đường dẫn va chạm và đường dẫn phân kỳ.

#### **Khâu các điểm trong đường dẫn sự cố**

Để khai thác thành công chương trình nạn nhân, lỗ hổng của nó phải được kích hoạt trước và một số đối tượng đặc biệt bị hỏng. Do đó, Revery chọn các vị trí nơi các đối tượng đặc biệt bị hỏng trong đường dẫn sự cố làm điểm ghép nối.

Như đã đề cập trong phần “ [Bố cục đối tượng liên quan đến lỗ hổng](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Sec15) ”, trong đường dẫn sự cố, mỗi lần vi phạm quyền ghi sẽ làm hỏng một đối tượng đặc biệt và do đó, nó là một điểm ghép ứng cử viên. Đối với mỗi lần vi phạm quyền truy cập đọc trong lỗ hổng UAF, đối tượng đặc biệt là đối tượng đã được giải phóng nhưng vẫn được trỏ bởi con trỏ lơ lửng. Vùng bộ nhớ của đối tượng đặc biệt này sẽ bị chiếm bởi một cấp phát bộ nhớ khác. Revery lấy thao tác cấp phát bộ nhớ mới làm điểm ghép ứng cử viên.

Vì có thể có nhiều vi phạm trong một đường dẫn đến sự cố, nên cũng có thể có nhiều điểm nối. Revery sẽ cố gắng nối từng cái với đường phân kỳ.

#### **Khâu các điểm trong các đường dẫn phân kỳ**

Để khai thác thành công chương trình nạn nhân, các thao tác có thể khai thác phải được thực hiện trên các đối tượng ngoại lệ hoặc đối tượng thế chấp bị hỏng.

**Điểm khâu tốt là gì?**Mỗi hướng dẫn có thể được sử dụng như điểm khâu. Nhưng không phải tất cả chúng đều tốt. Một điểm khâu thích hợp phải đáp ứng một số tiêu chí:

*Không quá gần các điểm vào.*Nếu không, nhiều hoạt động trùng lặp như đường dẫn sự cố sẽ được thực hiện. Do các hoạt động trùng lặp (ví dụ: khởi tạo đối tượng) sẽ không xảy ra trong luồng điều khiển hợp lệ, nên không thể tìm thấy đường dẫn để kết nối điểm ghép nối này với đối tác của nó trong đường dẫn sự cố.

*Không quá gần các điểm có thể khai thác.*Mặt khác, cần có một đường dẫn dài hơn để kết nối điểm ghép nối này với điểm đối chiếu của nó, đòi hỏi nhiều nỗ lực thực hiện tượng trưng hơn. Điểm nối có thể được đặt trước một số thao tác nhất định, ví dụ: khởi tạo toán hạng của các điểm có thể khai thác, để tiết kiệm các nỗ lực thực thi tượng trưng.

*Phụ thuộc dữ liệu tối thiểu.*Luồng dữ liệu sau điểm ghép trong đường dẫn phân kỳ phải có ít giao điểm với luồng dữ liệu trước điểm ghép trong đường dẫn sự cố.

**Làm thế nào để tìm điểm khâu?**Ở cấp độ cao, Revery so khớp sự phụ thuộc dữ liệu của đường dẫn phân kỳ với đường dẫn gặp sự cố và xác định sự khác biệt. Sau đó, nó sử dụng hướng dẫn gây ra sự khác biệt trong đường phân kỳ làm điểm nối.

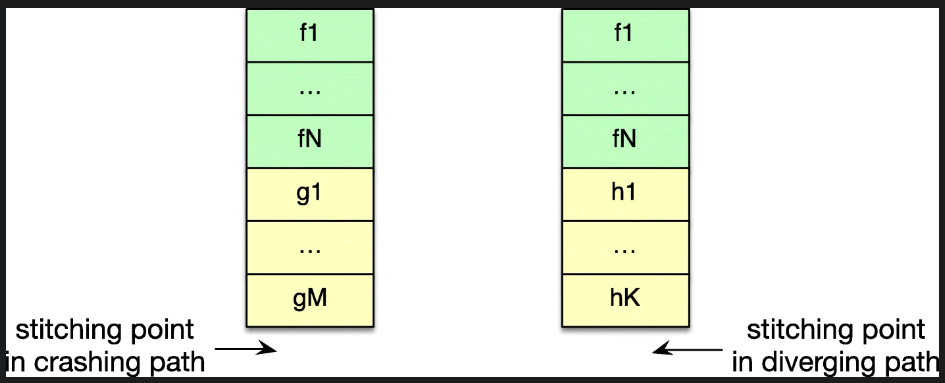
Đầu tiên, Revery xây dựng sơ đồ trình đóng góp bố cục của toán hạng của thao tác có thể khai thác trong đường dẫn phân kỳ. Sau đó, nó so khớp sơ đồ này với sơ đồ của đối tượng đặc biệt trong đường dẫn va chạm. Nếu cái trước là biểu đồ con của cái sau, điều đó có nghĩa là đường dẫn sự cố đã thiết lập tất cả các phụ thuộc dữ liệu cho hoạt động có thể khai thác. Sau đó, hướng dẫn trong đường dẫn phân kỳ, ngay sau lần truy cập ghi cuối cùng vào toán hạng của hoạt động có thể khai thác, được chọn làm điểm ghép.

Mặt khác, có các nút hoặc cạnh khác nhau trong sơ đồ của đường phân kỳ, tức là đường phân kỳ đã xen kẽ sự phụ thuộc của các hoạt động có thể khai thác. Trong trường hợp này, Revery chọn hướng dẫn sớm nhất (tạo hoặc ghi đối tượng) trong đường dẫn phân kỳ gây ra sự khác biệt trong sơ đồ, làm điểm ghép.

### Khâu đường dẫn luồng điều khiển

Để ghép đường dẫn sự cố và đường dẫn phân kỳ lại với nhau, Revery khám phá các đường dẫn con tiềm năng kết nối các điểm ghép nối trong các đường dẫn này. Nói chung, nó dựa vào việc thực thi biểu tượng để khám phá các đường dẫn. Tuy nhiên, Revery sử dụng một số kinh nghiệm để hướng dẫn thực thi biểu tượng một cách hiệu quả.

Trước hết, Revery sử dụng ngăn xếp cuộc gọi chức năng để hướng dẫn khám phá đường dẫn. Nó kiểm tra ngăn xếp cuộc gọi tại hai điểm ghép tương ứng và tìm thấy sự khác biệt. Hình  [9](https://link.springer.com/article/10.1186/s42400-019-0028-9" \l "Fig9) cho thấy hai ngăn xếp cuộc gọi ví dụ. Những khác biệt trong ngăn xếp cuộc gọi này cho biết hướng khám phá lộ trình. Các lời gọi hàm trong đường dẫn sự cố (ví dụ: g1, g2,..., gM trong hình) phải được trả về từng cái một, trong khi các lời gọi hàm trong đường dẫn phân kỳ (ví dụ: h1, h2,..., hK trong hình) nên được gọi từng cái một sau đó.



Nói cách khác, khi khám phá các đường dẫn tiềm năng, Revery sẽ thêm lần lượt lệnh trả về của hàm gM,..., g2, g1 làm lệnh đích, sau đó thêm điểm vào của hàm h1, h2,..., hK như hướng dẫn mục tiêu từng cái một. Các hướng dẫn đích này là các điểm thống trị giữa hai điểm ghép. Sau đó, Revery sẽ khám phá các đường dẫn phụ tiềm năng giữa các hướng dẫn mục tiêu trung gian này.

Revery tiếp tục giảm nhẹ quá trình khám phá đường dẫn phụ bằng cách sử dụng lại các đường dẫn hiện có. Ví dụ: nếu đã có một đường dẫn phụ kết nối hai đích trung gian trong đường dẫn phân kỳ hoặc đường dẫn gặp sự cố, Revery sẽ sử dụng lại đường dẫn phụ này. Revery cũng thực hiện một thuật toán xác định vòng lặp đơn giản và tìm một đường dẫn phụ để thoát khỏi vòng lặp càng sớm càng tốt, nhằm giảm bớt gánh nặng thực thi tượng trưng. Đôi khi, đường dẫn phụ được sử dụng lại sẽ khiến đường dẫn tổng thể không thể giải quyết được, Revery sẽ cố gắng xóa các đường dẫn phụ này và tìm kiếm các đường dẫn phụ thay thế.

Bằng cách này, Revery giảm đáng kể gánh nặng thực thi biểu tượng khi khám phá các đường dẫn phụ để kết nối các điểm nối.

### thế hệ khai thác

Sau khi tìm thấy một đường dẫn phụ kết nối hai điểm ghép, một đường dẫn khai thác ứng cử viên sẽ được xây dựng. Revery cũng có thể giải quyết các ràng buộc về lỗ hổng, các ràng buộc về đường dẫn và các ràng buộc khai thác để tạo các mẫu khai thác cuối cùng. Tuy nhiên, nó là bất cập.

#### **Các ràng buộc trạng thái có thể khai thác**

Chỉ giải quyết các ràng buộc của đường dẫn khai thác có thể không kích hoạt trạng thái có thể khai thác giống như đường dẫn phân kỳ. Do đó, Revery thêm một số ràng buộc dữ liệu bổ sung vào đường dẫn khai thác, đảm bảo trạng thái chương trình vẫn có thể khai thác được.

Đầu tiên, kích thước cấp phát bộ nhớ trong đường dẫn khai thác phải giống với đường dẫn phân kỳ để kích hoạt các trạng thái có thể khai thác như trong đường dẫn phân kỳ. Revery ghi lại kích thước cụ thể của tất cả các phân bổ bộ nhớ khi phân tích đường dẫn phân kỳ. Trong đường dẫn khai thác, nếu cấp phát bộ nhớ trong đường dẫn phân kỳ có kích thước tượng trưng, thì Revery sẽ thêm một ràng buộc để đảm bảo kích thước này bằng với giá trị cụ thể trong đường dẫn phân kỳ.

Thứ hai, Revery sẽ căn chỉnh sơ đồ của đường dẫn sự cố với đường dẫn phân kỳ. Một số địa chỉ tượng trưng trong đường dẫn phân kỳ giống như các đối tác của chúng trong đường dẫn sự cố về mặt logic. Vì vậy, trong lộ trình khai thác được ghép nối, các ràng buộc bổ sung phải được đưa ra để khẳng định sự bình đẳng giữa các địa chỉ tượng trưng này.

#### **ràng buộc tải trọng**

Với các ràng buộc về trạng thái có thể khai thác đã nói ở trên, cùng với các ràng buộc về lỗ hổng và đường dẫn, Revery có thể tạo *các đầu vào* ***EXP*** để kích hoạt cả trạng thái có thể khai thác và các lỗ hổng. Những đầu vào này có thể giúp các chuyên gia bảo mật xây dựng một khai thác đầy đủ.

Trong một số trường hợp nhất định, Revery có thể trực tiếp tạo ra các khai thác đang hoạt động. Tại điểm có thể khai thác, Revery có thể xây dựng các ràng buộc tải trọng có thể dẫn đến việc chiếm đoạt luồng điều khiển. Nếu điểm có thể khai thác là một lệnh gọi hàm (ví dụ: *lệnh gọi* gián tiếp hoặc lệnh jmp ) và mục tiêu của nó là một giá trị tượng trưng, thì Revery sẽ thêm một ràng buộc bổ sung để đặt mục tiêu thành giá trị do kẻ tấn công kiểm soát. Nếu trạng thái có thể khai thác là quyền truy cập ghi và cả địa chỉ đích và nội dung cần ghi đều mang tính biểu tượng, thì Revery sẽ thêm một ràng buộc bổ sung để ghi đè lên một địa chỉ đã biết (ví dụ: các mục nhập Bảng bù đắp toàn cầu hoặc con trỏ hàm toàn cục) với giá trị do kẻ tấn công kiểm soát.

Bằng cách này, Revery có thể tạo ra các khai thác để chiếm đoạt luồng điều khiển trong một số trường hợp nhất định. Tuy nhiên, nó không phải lúc nào cũng đảm bảo thành công.

**FUZZING**

Các bước kiểm thử fuzzing:

Bước 1: Nhận dạng hệ thống mục tiêu (**Recognition of the target system.**)

Bước 2: Công nhận các yếu tố đầu vào (**Recognition of the inputs**.)

Bước 3: Tạo dữ liệu mờ (**Fuzzed data Generation.**)

Bước 4: Thực hiện kiểm tra bằng các sử dụng dữ liệu mờ (**Test Execution using fuzzy data.**)

Bước 5: Giám sát hành vi hệ thống (**System behavior Monitoring**.)

Bước 6: Ghi nhật ký lỗi (**[Logging of defects](https://www.educba.com/what-is-defect/)**.)

