Mục lục

I. Đặt vấn đề....1
1. Lí do nghiên cứu....1
2. Tình hình nghiên cứu liên quan....2
2.1. Ngoài nước....2
2.2. Trong nước....5
II. Nội dung dự định nghiên cứu....6
1. Mục tiêu nghiên cứu....6
1.1. Mục tiêu tổng quát....6
1.2. Mục tiêu cụ thể....6
2. Phương hướng nghiên cứu....6
2.1. Ý tưởng tiếp cận....6
2.2. Mô hình kiến trúc đề xuất....7
3. Ý nghĩa khoa học của đề tài....9
4. Tính mới, tính độc đáo và tính sáng tạo của đề tài....9
III. Dự kiến kế hoạch làm việc....9
1. Lịch trình làm việc....9
2. Phần báo cáo chuyên đề....9
2.1. Chuyên đề 1....9
2.2. Chuyên đề 2....10
2.3. Chuyên đề 3....10
2.4. Chuyên đề 4....11
IV. Tài liệu tham khảo....11

I. Đặt vấn đề

1. Lí do nghiên cứu

Hiện nay, với sự phát triển của phần mềm, cả về số lượng, chất lượng và độ phức tạp, vấn đề kiểm tra tính đúng đắn của chương trình nổi lên như một nhu cầu cấp thiết. Trong đó, *kiểm thử* (testing) là một khâu quan trọng và không thể thiếu. Kiểm thử là một qui trình bao gồm các bước từ xét duyệt và kiểm tra một cách thủ công [1] cho đến sinh ra những test case phù hợp dựa trên việc phân tích sự bao phủ [2]. Tuy nhiên, phương pháp này tồn tại lỗ hổng, do không thể sinh đầy đủ các test case cần thiết để đảm bảo chương trình không có lỗi. Mặt khác, phương pháp kiểm thử đòi hỏi thực thi đoạn code cần kiểm tra trong môi trường thực. Điều này trong thực tế khá khó khăn. Một giải pháp được đề xuất là xây dựng môi trường giả lập, thông qua các công cụ ảo hóa (ví dụ như Virtual Machine). Tuy nhiên, các công cụ này rất tốn kém và không phản ánh chính xác môi trường thực tế. Chính từ những vấn đề này của kiểm thử đã nảy sinh phương pháp *kiểm chứng phần mềm* (software verification), trong đó tính đúng của phần mềm sẽ được kiểm chứng dựa trên logic. Hướng tiếp cận này chủ yếu dựa vào các *phương pháp hình thức* (formal method). Ý tưởng chính của phương pháp hình thức là biểu diễn phần mềm và hệ thống thành các đối tượng toán học, từ đó suy luận và xác định các hình vi của chúng. Trong lĩnh vực áp dụng phương pháp hình thức để kiểm chứng phần mềm, có hai hướng tiếp cận chủ yếu: *chứng minh lí thuyết* (theorem proving) và *kiểm tra mô hình* (model checking) [3].

Các phương pháp hình thức có thể được sử dụng để kiểm tra phần mềm ở nhiều mức độ khác nhau, từ các *đặc tả* (specification), *thiết kế* (design), *mã nguồn* (source code) cho đến *mã thực thi* (binary code). Hiện nay, hướng nghiên cứu sử dụng phương pháp hình thức cho mã thực thi ngày càng được sử dụng rộng rãi trong giới học thuật lẫn cộng đồng doanh nghiệp do một số nguyên nhân sau. Thứ nhất, hầu hết các mã nguồn hiện nay thường đóng, việc kiểm tra source code là khó khả thi, đặc biệt là đối với các virus. Trên thực tế, phân tích virus là một ví dụ điển hình và rõ ràng của vấn đề phân tích mã thực thi không có mã nguồn. Thứ hai, mã thực thi phụ thuộc vào quá trình biên dịch của chương trình. Mà quá trình biên dịch có khả năng phát sinh lỗi do quá trình tối ưu hóa làm thay đổi hành vi của chương trình [4]. Chính vì nguyên nhân này, việc kiểm tra trên mã thực thi sẽ cho ra các kết quả đáng tin cậy vì máy tính sẽ thực thi binary code thay vì source code như là kết quả duy nhất và cuối cùng. Thứ ba, khi chương trình được phát triển từ nhiều ngôn ngữ khác nhau và được tích hợp nhiều module khác nhau hoặc khi phần mềm cần các module thư viện chỉ được cung cấp ở dạng mã nhị phân, vấn đề phân tích mã nguồn sẽ trở nên hết sức khó khăn do vấn đề thiếu hụt cấu trúc.

Tuy nhiên, trong phương pháp phân tích mã thực thi, cũng nổi lên rất nhiều những thách thức. Thách thức đầu tiên là độ phức tạp của mã thực thi. Những kiến trúc CISC (Complex Instruction Set Computer [33]) như x86 có rất nhiều các câu lệnh khác nhau để phục vụ các

tác vụ hệ thống. Chẳng hạn, trong kiến trúc x86, có hàng trăm câu lệnh và hàng ngàn khả năng kết hợp giữa các toán hạng [5]. Mặt khác, mỗi câu lệnh lại có ngữ nghĩa phức tạp. Một chương trình phân tích mã thực thi phải thỏa mãn điều kiện, biểu diễn một cách chính xác các câu lệnh này. Đây là một vấn đề khó khăn. Thách thức thứ hai đến từ vấn đề thiếu hụt các *cấu trúc ngữ nghĩa bậc cao* (high level semantic structure) trong mã nhị phân. Mã nhị phân không chứa các hàm và các kiểu dữ liệu. Mặt khác, chương trình phân tích mã nhị phân phải giải quyết các vấn đề về sự nhập nhằng giữa mã chương trình và dữ liệu, các *câu lệnh nhảy không trực tiếp* (indirect branches) và vấn đề các *câu lệnh lồng nhau* (overlapping instructions) [30]. Do những khó khăn trên, đã dẫn đến vấn đề bùng nổ độ phức tạp trong phân tích mã nhị phân của các chương trình lớn. Vì thế có rất ít các framework phân tích mã nhị phân có khả năng phân tích các chương trình ứng dụng với độ phức tạp cao. Đây là một nhu cầu bức thiết đặt ra trong thời điểm hiện tại và là nguyên nhân thúc đẩy của đề tài này.

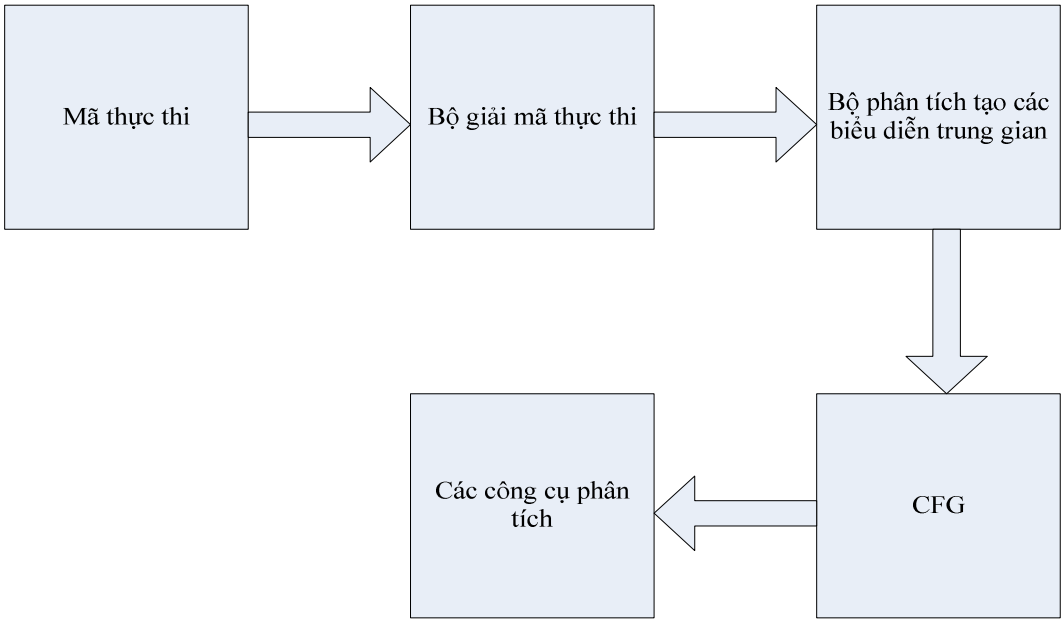
Như vậy, phân tích mã thực thi là một hướng nghiên cứu có ý nghĩa to lớn trong lĩnh vực phần mềm. Ngoài ra, do sự phát triển nhanh chóng của các kho ứng dụng trực tuyến (ví dụ như App Store của Apple hay Appstore của Amazon …), nhu cầu phân tích phần mềm dựa trên mã thực thi càng trở nên rất quan trọng. Yêu cầu này đòi hỏi phải xây dựng một công cụ tự động với độ chính xác cao. Đề tài nghiên cứu này sẽ tập trung vào yêu cầu phân tích mã thực thi để giải quyết các vấn đề trên cũng như góp phần giải quyết các vấn đề lí thuyết sẽ được trình bày ở các phần sau.

2. Tình hình nghiên cứu liên quan

2.1. Ngoài nước

Trong ngành công nghiệp phần mềm, xu hướng kiểm tra phần mềm dựa trên mã thực thi thay vì mã nguồn đang phát triển một cách mạnh mẽ. Ngày nay, trên thế giới có rất nhiều dự án đi theo hướng phát triển này [6][7][8][9], và nổi lên dự án BINCOA [21] được hỗ trợ bởi ANR. Bên cạnh BINCOA, có một số dự án nghiên cứu và thương mại với hướng tiếp cận khá tương tự, như IDA Pro [10], Jakstab [11], CodeSurfer/x86 [17][18]…. Các hướng tiếp cận này đều cố gắng đưa ra các framework để phân tích mã nhị phân. Các framework được đề xuất đều có các nguyên lí hoạt động chung như sau:

- Bước 1: *Biên dịch ngược* (disassembly) các mã thực thi của chương trình
- Bước 2: *Biểu diễn trung gian* (intermediate representation) kết quả bước 1
- Bước 3: Xây dựng *đồ thị luồng điều khiển* (Control Flow Graph - CFG) từ kết quả bước 2
- Bước 4: Phân tích chương trình dựa trên đồ thị luồng điều khiển được xây dựng từ bước 3

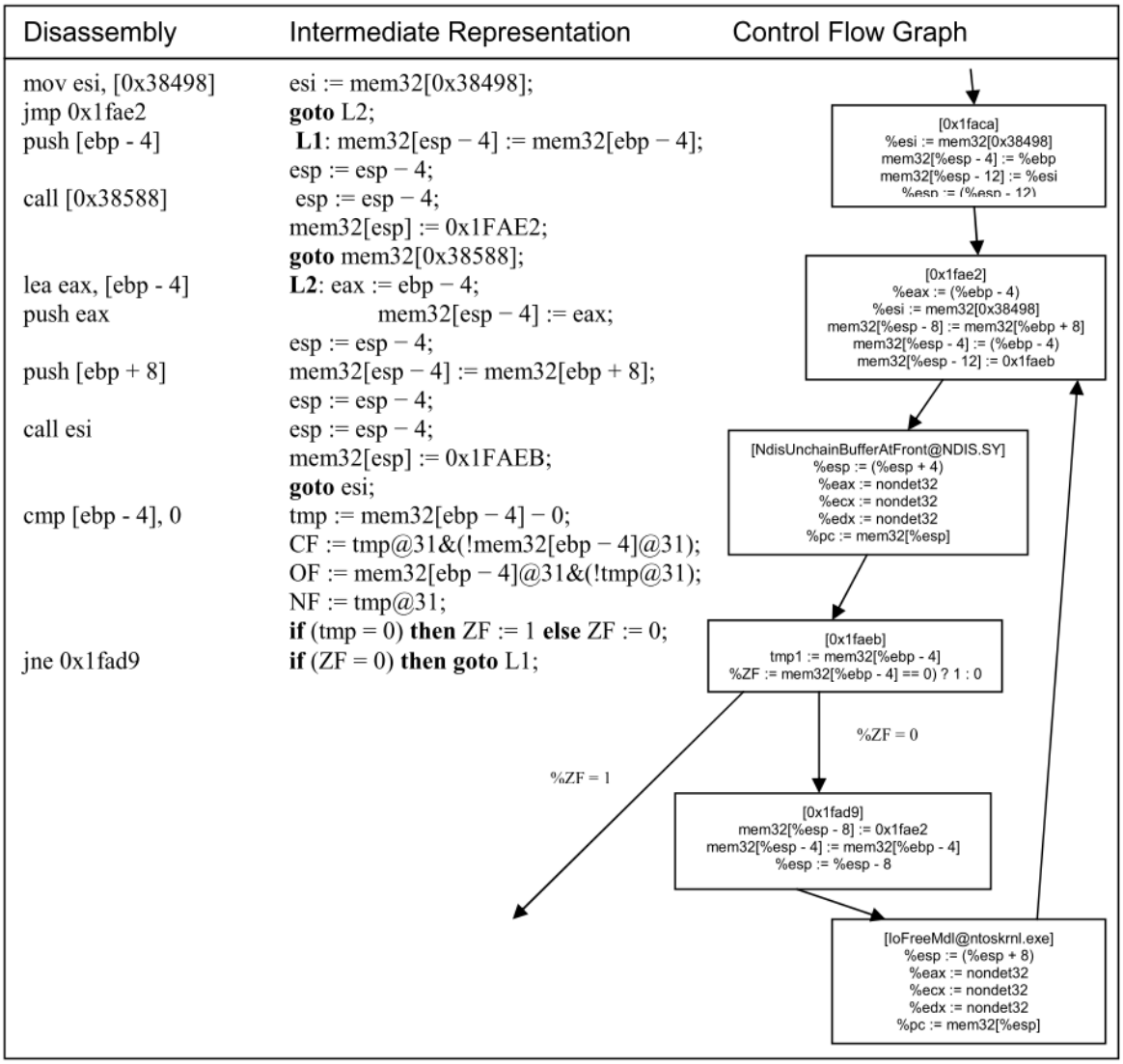


Hình 1: Sơ đồ tổng quát quá trình phân tích dựa trên mã thực thi

Có hai hướng tiếp cận chính trong kĩ thuật biên dịch ngược [22]: *phương pháp quét tuyến tính* (linear sweep) và *duyệt đệ quy* (recursive traversal). Ý tưởng căn bản của phương pháp quét tuyến tính là giải mã các byte thành câu lệnh assembly một cách tuần tự từ đầu đến cuối của mã thực thi. Phương pháp đơn giản này được hiện thực trong các công cụ, như GNU objdump [31]. Tuy nhiên, do vấn đề các câu lệnh lồng nhau, phương pháp này có rất nhiều hạn chế trong ứng dụng thực tế biên dịch ngược mã thực thi. Phương pháp duyệt đệ quy có ý tưởng chính là công cụ dịch bắt đầu từ điểm vào của chương trình, lần theo các nhánh và sử dụng kĩ thuật *tìm kiếm theo chiều sâu* (depth first search) để giải mã chương trình. Phương pháp này là ý tưởng cơ bản được áp dụng trong chương trình mang tính thương mại, như IDA Pro. Tuy nhiên, do sự xuất hiện của các câu lệnh nhảy không trực tiếp trong mã thực thi, sẽ có những vùng trong mã thực thi không được phát hiện bởi kĩ thuật này. Do đó, để giải quyết vấn đề trên, các công cụ biên dịch ngược sẽ bổ sung quá trình duyệt đệ quy bằng kĩ thuật heuristics để phát hiện các đoạn code tiềm ẩn trong mã thực thi của chương trình. Các heuristics này tận dụng những *mẫu biên dịch thường xảy ra* (compiler idioms) trong chương trình, chẳng hạn như các mẫu thường gặp trong quá trình tính toán các lệnh nhảy từ bảng các bước nhảy [23].

Sau khi hoàn tất quá trình biên dịch ngược, kết quả này sẽ được chuyển thành biểu diễn trung gian bằng cách sử dụng thư viện mô tả ngữ nghĩa ngôn ngữ assembly. Dựa trên biểu diễn trung gian thu được, các công cụ phân tích mã thực thi sẽ xây dựng nên các đồ thị luồng điều khiển tương ứng. Sơ đồ sau đây mô tả về các bước biên dịch ngược, xây dựng biểu diễn trung gian và xây dựng CFG tương ứng của công cụ Jakstab. Các CFG được sinh ra sẽ được sử dụng trong rất nhiều mục đích khác nhau, chẳng hạn như phát hiện các *phần mềm độc hại*

(malwares), *phân tích đa đường* (multipath analysis) và xác định các *hành vi độc hại tiềm ẩn* (dormant malicious behaviors) [32]. Mặt khác, CFG hỗ trợ vấn đề xây dựng phương pháp tự động để phân loại các phần mềm độc hại, bằng cách áp dụng *giải thuật so sánh cây* (tree difference algorithm [24]) trên các CFG kết quả.



Hình 2: Sơ đồ ví dụ về quá trình phân tích mã thực thi của Jakstab [30]

Hiện nay, các framework xử lý mã nhị phân đều chủ yếu dựa vào kỹ thuật *phân tích tĩnh* (static analysis) bằng các phương pháp hình thức để thực hiện các bước phân tích. BINCOA có thể xem như một ví dụ tiêu biểu của hướng tiếp cận này. Ý tưởng chính của BINCOA là tạo ra một framework nền tảng hỗ trợ vấn đề *phân tích mã thực thi* (binary code analysis), không chỉ ứng dụng trong lĩnh vực nghiên cứu mà còn trong công nghiệp. Về khía cạnh học thuật, BINCOA hỗ trợ phân tích mã thực thi theo 2 hướng nghiên cứu chính như sau: phân tích tĩnh,

áp dụng *suy diễn trừu tượng* (abstract interpretation [29]) của mã thực thi và *phân tích concolic* (concolic analysis), là kĩ thuật phân tích kết hợp hai hướng kiểm thử và *thực thi kí hiệu* (symbolic execution [13]). Về khía cạnh công nghiệp, BINCOA đã được áp dụng thực tế với vai trò là công cụ test ở qui mô lớn (CEA, EDF, Trusted Lab và SAGEM). Jakstab [14][15] là một framework phân tích mã nhị phân dựa trên kĩ thuật phân tích tĩnh để xây dựng đồ thị luồng dữ liệu. Jakstab dịch ngược mã nhị phân thành dạng biểu diễn trung gian và sử dụng kĩ thuật suy diễn trừu tượng kết hợp với *phân tích luồng dữ liệu* (data flow analysis) để giải quyết vấn đề vấn đề lệnh nhảy động. Framework này được viết bằng ngôn ngữ Java. Vì sử dụng kĩ thuật suy diễn trừu tượng, Jakstab có yếu điểm là sự không chính xác trong kết quả thu được. Công cụ phân tích mã nhị phân Codesurfer/X86 được giới thiệu bởi Balakrishnan và Reps. Ý tưởng chính của Codesurfer/X86 là thu thập thông tin về quan hệ giữa các giá trị và tính toán xấp xỉ các giá trị dựa trên miền trừu tượng của *tập hợp các giá trị* (value set analysis [18]). Codesurfer/X86 sử dụng công cụ IDA Pro trong vấn đề dịch ngược mã nhị phân. Vì thế, nhược điểm của Codesurfer/X86 chính là kĩ thuật diassembly dựa trên heuristics được sử dụng trong IDA Pro.

Những kết quả ở trên là động lực thúc đẩy đề tài xây dựng một công cụ phân tích mã nhị phân bằng phương pháp hình thức. Tuy nhiên, nếu chỉ áp dụng các kỹ thuật phân tích tĩnh, độ phức tạp phải trả là rất cao và dễ dàng bùng nổ khi xử lý các ứng dụng thực tế. Các framework được đề xuất như đã thảo luận bên trên đều gặp khó khăn khi phân tích các chương trình thực tế với hàng triệu dòng mã lệnh.

Để vượt qua khó khăn trên, đề tài đề xuất một hướng đi mới bằng phương pháp kết hợp *phân tích tĩnh* (static analysis) và kĩ thuật *kiểm tra động* (dynamic testing), như sẽ trình bày ở các phần tiếp theo.

2.2. Trong nước

Ở Việt Nam hiện tại, số lượng nhóm nghiên cứu về vấn đề kiểm tra hình thức là không nhiều, có thể kể đến nhóm nghiên cứu tại trường Đại học Bách Khoa Hà Nội do TS. Trương Anh Hoàng làm trưởng nhóm. Nhóm có một số công trình nghiên cứu về sử dụng *lập trình hướng khía cạnh* (aspect oriented programming) để kiểm tra giao thức tương tác giao diện trong chương trình Java [12] … Bên cạnh đó là nhóm nghiên cứu về *Phân tích và Kiểm tra Hệ thống* (System Analysis and VErification - SAVE) tại trường Đại học Bách Khoa TP.HCM dưới sự hướng dẫn của PGS.TS Quản Thành Thơ. SAVE là một trong số rất ít các nhóm nghiên cứu tập trung vào vấn đề kiểm chứng phần mềm sử dụng các phương pháp hình thức. Dựa trên ý tưởng ban đầu là kiểm chứng các bài tập lập trình của sinh viên, nhóm đã phát triển và đạt được những thành tựu nhất định trong lĩnh vực kiểm chứng phần mềm và đặc biệt là một số kết quả trong phân tích virus sử dụng các phương pháp hình thức. Trong bài báo này [25] của nhóm SAVE, tác giả đã đề xuất phương pháp sử dụng suy diễn trừu tượng để trừu

tượng hóa các *chữ kí* (signagure) của virus trong bộ nhớ nhằm quản lí vấn đề *làm rối rắm* (obfuscation) của *virus đa hình* (polymorphism virus).

II. Nội dung dự định nghiên cứu

1. Mục tiêu nghiên cứu

1.1. Mục tiêu tổng quát

Mục tiêu của đề tài: Nghiên cứu phương pháp kết hợp hai kĩ thuật phân tích tĩnh và kiểm tra động trong phân tích mã nhị phân để giải quyết các bài toán thực tế, ví dụ như phân tích mã độc.

1.2. Mục tiêu cụ thể

- Nghiên cứu các kĩ thuật được sử dụng trong phân tích tĩnh, tìm kiếm một mô hình phù hợp để kết hợp với kĩ thuật kiểm tra động.
- Xây dựng framework kết hợp cả hai kĩ thuật, phân tích tĩnh và kiểm tra động trong xử lí mã thực thi.
- Ứng dụng framework này trong bài toán sinh ra đồ thị luồng điều khiển của mã thực thi từ đó sinh ra một CFG chính xác và đầy đủ với tốc độ xử lý nhanh hơn các phương pháp hiện có.
- Từ CFG xây dựng được, tiếp tục giải quyết một số vấn đề nổi bật hiện nay của phân tích mã nhị phân, bao gồm việc phân tích tính an toàn của chương trình gốc và phát hiện mã độc có thể tồn tại trong chương trình này.

2. Phương hướng nghiên cứu

2.1. Ý tưởng tiếp cận

Đề tài được xây dựng dựa trên các nền tảng lí thuyết như sau:

- Kĩ thuật phân tích tĩnh
- Kĩ thuật kiểm tra động

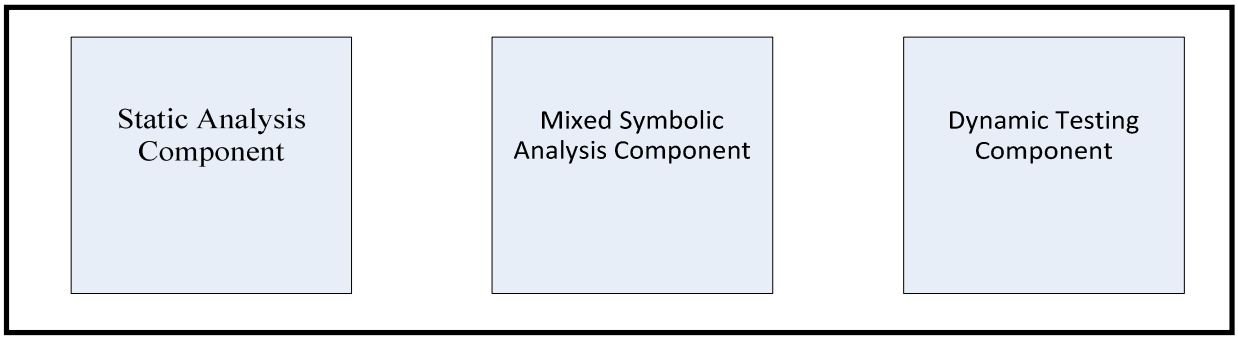
Kiểm tra động (dynamic testing) là kĩ thuật cho phép thu thập thông tin của chương trình thông qua quá trình thực thi đoạn code trên máy thực hoặc máy ảo trong một môi trường có kiểm soát. Tuy nhiên, dynamic testing có nhược điểm về mặt lí thuyết, do các phương pháp sinh test case như sử dụng Petri nets [34] hay phân tích sự bao phủ [2] … không thể vét cạn tất cả các trường hợp cần thiết để bảo đảm chương trình không có lỗi.

Phân tích tĩnh là kĩ thuật thu thập các thông tin của chương trình mà không cần phải thực thi đoạn code. Ưu điểm của phân tích tĩnh là kĩ thuật này cho phép nghiên cứu nhiều đường thực thi, điều này trái ngược với kiểm tra động. Có rất nhiều kĩ thuật được sử dụng trong phân tích tĩnh, chẳng hạn như suy diễn trừu tượng, phân tích luồng dữ liệu, thực thi kí hiệu…

Là một kĩ thuật cổ điển trong *kiểm tra hình thức* (formal verification), kĩ thuật phân tích tĩnh là một hướng tiếp cận khả quan và được ứng dụng nhiều trong bài toán kiểm tra mã thực thi. Tuy nhiên, thách thức của kĩ thuật phân tích tĩnh đến từ bài toán xác định địa chỉ đến trong các câu lệnh nhảy động. Ví dụ như, để xác định câu lệnh kế tiếp sau lệnh CALL EAX, chúng ta cần xác định chính xác giá trị của thanh ghi EAX trong thời điểm thực thi của chương trình. Đây là một nhiệm vụ rất khó khăn với điều kiện không thực thi chương trình. Mặt khác, sự nhập nhằng giữa code và dữ liệu trong phân tích tĩnh, những kĩ thuật được sử dụng trong các đoạn mã độc để chống lại kĩ thuật phân tích tĩnh như *đóng gói code* (code packing), mã hóa và obsfucation… cũng là những thách thức to lớn.

Do đó, đề tài này đề xuất một hướng đi mới kết hợp ưu điểm của hai kĩ thuật, phân tích tĩnh và kiểm tra động. Kĩ thuật phân tích tĩnh được áp dụng để phân tích mã thực thi. Khi gặp những địa chỉ không thể phân giải trong lệnh nhảy động hay những vùng chưa được xử lí trong mã thực thi của chương trình bởi phân tích tĩnh, kĩ thuật kiểm tra động sẽ được áp dụng để tìm lời giải. Hai phương pháp này kết hợp với nhau để xây dựng đồ thị luồng thực thi chính xác và đầy đủ của chương trình.

2.2. Mô hình kiến trúc đề xuất



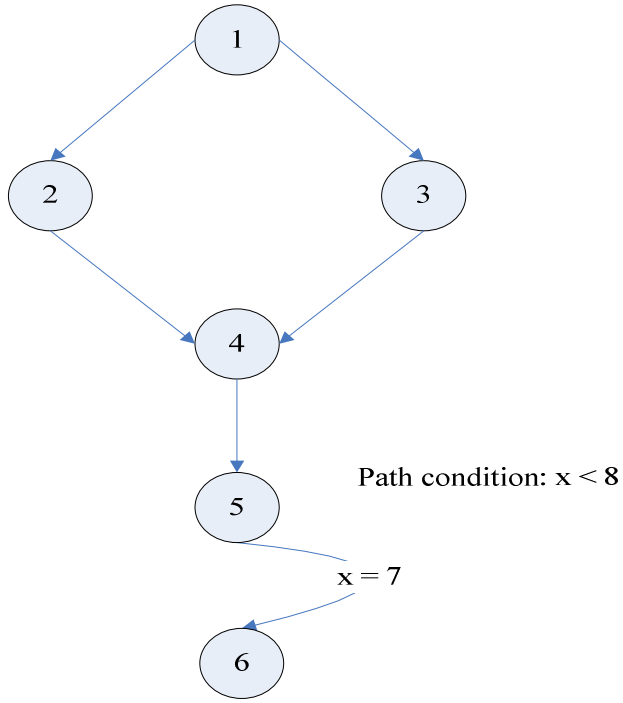
Hình 3: Sơ đồ mô hình kiến trúc đề xuất

Mô hình kiến trúc đề xuất của đề tài sẽ gồm 3 thành phần chính, *thành phần phân tích tĩnh* (static analysis component - SAC), *thành phần kiểm tra động* (dynamic testing component - DTC) và *thành phần phân tích kí hiệu hỗn hợp* (mixed symbolic analysis component - MSAC), kết hợp giữa phân tích tĩnh và động.

- SAC làm nhiệm vụ chuyển đổi câu lệnh assembly thành dạng biểu diễn trung gian và cung cấp những chức năng nền tảng cho quá trình phân tích tĩnh trên kết quả biểu diễn trung gian này, ví dụ như phân tích luồng dữ liệu, tối ưu hóa, thực thi kí hiệu …
- Vai trò của DTC là cung cấp nền tảng cơ bản cho phép hiện thực mã thực thi bằng cách sử dụng công cụ *mô phỏng* (emulator). Chương trình được chạy trên DTC và cho phép người dùng quan sát hành vi của những đoạn code được thực thi.

- MSAC sử dụng các chức năng được cung cấp bởi SAC và DTC, kết hợp với quá trình thực thi kí hiệu trên mã nhị phân của chương trình. Ý tưởng chính của phương pháp thực thi kí hiệu là sử dụng các kí hiệu như tham số đầu vào của chương trình thay vì các giá trị cụ thể. Đối với một đường thực thi của chương trình được xây dựng bằng kĩ thuật thực thi kí hiệu, MSAC sẽ xác định *điều kiện đường đi* (path condition) tương ứng. Bằng cách sử dụng các SMT (Satisfiability Modulo Theories) Solvers như Z3 [27], STP [28] hoặc các giải thuật heuristics [16][26] để giải các đường đi này, chương trình sẽ xác định được đường đi đó có khả thi hay không, hoặc với những giá trị nào sẽ dẫn đến đường đi tương ứng. Kết hợp với DTC để tiến hành thực thi các đường đi này, MSAC cho phép phát hiện những vùng chưa được xử lí trong mã nhị phân của chương trình.

Ví dụ dưới đây mô tả cho sự kết hợp các thành phần SAC, DTC và MSAC trong phân tích mã nhị phân.



Hình 4: Sơ đồ ví dụ về sự kết hợp các thành phần

Mã thực thi của chương trình sẽ được xử lí bởi SAC để sinh ra CFG tương ứng (giả sử là các node 1,2,3,4 và 5 với sơ đồ như hình vẽ). Do vấn đề các câu lệnh nhảy động trong mã nhị phân, quá trình phân tích tĩnh sẽ không thể thực thi trong toàn bộ mã chương trình. Sẽ có những vùng trong mã chương trình chưa được xử lí. Khi đó, kĩ thuật thực thi kí hiệu trong MSAC kết hợp với SMT Solvers sẽ được áp dụng để sinh ra các điều kiện đường đi tương ứng, node 5 với path condition là (x < 8). DTC sẽ tiến hành quá trình thực thi động (x = 7).

Và các đường thực thi sẽ được kiểm tra để phát hiện những vùng chưa được xử lí trong mã chương trình (node 6). Các bước này sẽ được lặp lại để bảo đảm phân tích đầy đủ mã thực thi của chương trình.

3. Ý nghĩa khoa học của đề tài

Đề tài đề xuất một hướng đi mới trong việc kiểm tra mã thực thi bằng cách xây dựng framework kết hợp phân tích tĩnh và kiểm tra động. Framework này sẽ tạo ra kết quả có độ chính xác cao và có khả năng mở rộng. Khi được hoàn thành, framework áp dụng phương pháp này có thể được sử dụng để kiểm tra mã thực thi một cách độc lập hoặc kết hợp với một số phương pháp kiểm tra hình thức khác như theorem proving hoặc model checking.

4. Tính mới, tính độc đáo và tính sáng tạo của đề tài

Tính độc đáo trong đề tài này là sự kết hợp giữa kĩ thuật phân tích tĩnh và kiểm tra động. Phân tích tĩnh được áp dụng để phân tích mã thực thi. Khi gặp những địa chỉ không thể phân giải trong lệnh nhảy động, kĩ thuật kiểm tra động sẽ được áp dụng để tìm lời giải. Hai phương pháp này kết hợp với nhau để xây dựng đồ thị luồng điều khiển của chương trình. Ngoài ra, tính sáng tạo của đề tài còn thể hiện ở việc phân tích các điểm mạnh và điểm yếu của từng kĩ thuật hiện tại. Từ đó, loại bỏ các điểm yếu và kết hợp tốt các đặc tính của từng kĩ thuật để tạo ra những kết quả phù hợp.

III. Dự kiến kế hoạch làm việc

1. Lịch trình làm việc

	2013				2014				2015				2016			
	I	II	III	IV	I	II	III	IV	I	II	III	IV	I	II	III	IV
Chuyên đề 1																
Chuyên đề 2																
Chuyên đề 3																
Chuyên đề 4																
Luận văn																

2. Phần báo cáo chuyên đề

2.1. Chuyên đề 1

- Nội dung: biểu diễn trung gian trong phân tích mã nhị phân.
- Thời gian báo cáo chuyên đề: tháng 7/2014.

- Mục tiêu: xây dựng framework hỗ trợ biên dịch ngược mã nhị phân trong kiến trúc x86 thành biểu diễn trung gian.
- Công việc cần thực hiện:
    - Tìm hiểu về tập lệnh trong kiến trúc x86.
    - Tìm hiểu các kĩ thuật dịch ngược mã nhị phân.
    - Xây dựng mã biểu diễn trung gian.
- Kết quả mong đợi:
    - Framework này có thể xây dựng biểu diễn trung gian từ mã nhị phân một cách nhanh chóng và chính xác.

2.2. Chuyên đề 2

- Nội dung: xây dựng framework phân tích tĩnh.
- Thời gian báo cáo chuyên đề: tháng 2/2015.
- Mục tiêu: xây dựng một framework kết hợp biểu diễn trung gian mã nhị phân và kĩ thuật phân tích tĩnh.
- Công việc cần thực hiện:
    - Tìm hiểu về các kĩ thuật sử dụng trong phân tích tĩnh.
    - Kết hợp với biểu diễn trung gian.
- Kết quả mong đợi:
    - Xây dựng một framework phân tích tĩnh hỗ trợ đầy đủ tập lệnh của X86 để xây dựng nên CFG của chương trình.

2.3. Chuyên đề 3

- Nội dung: tìm hiểu về các kĩ thuật chống lại phân tích tĩnh trong phân tích mã nhị phân.
- Thời gian báo cáo chuyên đề: tháng 11/2015.
- Mục tiêu: tìm hiểu về nguyên tắc hoạt động, ưu và nhược điểm của các kĩ thuật chống phân tích tĩnh.
- Công việc cần thực hiện:
    - Tìm hiểu về các kĩ thuật code packing, mã hóa và obfuscation.
- Kết quả mong đợi:
    - Hỗ trợ cho vấn đề xây dựng framework phân tích tĩnh xử lí các khó khăn gây ra bởi các kĩ thuật này.

2.4. Chuyên đề 4

- Nội dung: tìm hiểu về phương pháp phân loại mã độc.
- Thời gian báo cáo chuyên đề: tháng 5/2016.
- Mục tiêu: tìm hiểu về nguyên tắc, kĩ thuật, những ưu khuyết điểm trong phương pháp phân loại mã độc.
- Công việc cần thực hiện:
    - Tìm hiểu về giải thuật tree difference algorithm.
- Kết quả mong đợi:
    - Ứng dụng các nguyên tắc trên trong bài toán sử dụng CFG sinh ra từ mã nhị phân để phân loại virus.

IV. Tài liệu tham khảo

[1] K.E. Wiegers, *Peer Reviews in Software*, Addison Wesley, London, UK, 2002.

[2] A. Spillner, T. Linz, H. Schaefer, *Software Testing Foundations*, dpunkt, 2006.

[3] E. M. Clarke, J. M. Wing et. Al, "Formal methods: State of the art and future directions", In *ACM Survey*, 28: 626-643, 1996.

[4] G. Balakrishnan, T. Reps, D. Melski, T. Teitelbaum. "WYSINWYX:What You See Is Not What You eXecute", In *Journal ACM Transactions on Programming Languages and Systems* , Lecture Notes in Computer Science, Springer, October 2005, pp. 202–213.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2009.

[6] Bor-Yuh Chang, Matthew Harren and George Necula, "Analysis of Low-Level Code Using Cooperating Decompilers", In *13th Int. Static Analysis Symp (SAS 2006)*, Ed. by Kwangkeun Yi. Vol. 4134. LNCS. Springer, 2006, pp. 318–335.

[7] Junghee Lim, Akash Lal and Thomas W. Reps, "Symbolic Analysis via Semantic Reinterpretation", In *Proc. 16th Int. Workshop Model Checking Software (SPIN 2009)* ,Ed. by Corina S. Păsăreanu, Vol. 5578. LNCS. Springer, 2009, pp. 148–168.

[8] M. Cova, V. Felmetsger, G. Banks, G. Vigna, "Static detection of vulnerabilities in x86 executables", In *ACSAC 2006.*

[9] J. Bergeron, M. Debbabi, M.M. Erhioui, B. Ktari, "Static analysis of binary code to isolate malicious behaviors", In *WETICE 1999.*

[10] IDA Pro, available at: http://www.hex-rays.com/idapro/

[11] Jakstab, available at: http://www.jakstab.org/

[12] Anh-Hoang Truong, Thanh-Binh Trinh, Dang Van Hung, Viet-Ha Nguyen, Nguyen Thi Thu Trang, Pham Dinh Hung, "Checking Interface Interaction Protocols Using Aspect-oriented Programming", In *SEFM'08*, Cape Town, South Africa, November 10-14, 2008.

[13] James C. King, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, "Symbolic execution and program testing", In *Communications of the ACM* Volume 19 Issue 7, July 1976, Pages 385–394.

[14] Johannes Kinder, Helmut Veith, "Jakstab: A Static Analysis Platform for Binaries", In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, vol. 5123, Lecture Notes in Computer Science, Springer, July 2008, pp. 423–427.

[15] Johannes Kinder, Helmut Veith, Florian Zuleger, "An Abstract Interpretation–Based Framework for Control Flow Reconstruction from Binaries", In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, vol. 5403, Lecture Notes in Computer Science, Springer, January 2009, pp. 214–228.

[16] Michael A., Colón, Sriram Sankaranarayanan, Henny B. Sipma, "Linear Invariant Generation Using Non–linear Constraint Solving", In *Computer Aided Verification (CAV)*, LNCS 2725, Springer–Verlag, pp 420–433, 2003, Lecture Notes in Computer Science, Springer, January 2009, pp. 214–228.

[17] T. Reps, J. Lim, A. Thakur, G. Balakrishnan and A. Lal, "There's plenty of room at the bottom: Analyzing and verifying machine code (Invited tutorial)", In *Proc. Computer Aided Verification*, July 2010.

[18] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables", In *Proc. Int. Conf. on Compiler Construction*, Springer-Verlag, New York, NY, 2004, 5-23.

[20] Sen, Koushik; Darko Marinov, Gul Agha (2005), "CUTE: a concolic unit testing engine for C", In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY: ACM. pp. 263–272.

[21] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, Aymeric Vincent, "The BINCOA Framework for Binary Code Analysis", In *23rd International Conference, CAV 2011*, Snowbird, UT, USA.

[22] Cullen Linn and Saumya K. Debray, "Obfuscation of executable code to improve resistance to static disassembly", In *Proc. 10th ACM Conf. Computer and Communications Security (CCS 2003)*. Ed. by Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger. ACM, 2003, pp. 290–299.

[23] Laune C. Harris and Barton P. Miller, “Practical analysis of stripped binary code”, In *SIGARCH Comput*. Archit. News 33.5 (2005), pp. 63–68.

[24] M. Hashimoto, A. Mori, “Diff/TS: A tool for fine-grained structural change analysis”, In *Proceedings of the 15th Working Conference on Reverse Engineering, WCRE (2008).*

[25] Binh T. Nguyen, Binh T. Ngo and Tho T. Quan, ”A Memory-based Abstraction Approach to Handle Obfuscation in Polymorphic Virus”, In *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), Postgrad Symposium*, pp. 158-161 ,Hong Kong, IEEE Press, ISBN 978-0-7695-4922-4.

[26] Sriram Sankaranarayanan, Henny Sipma and Zohar Manna, "Constraint-based Linear-Relations Analysis", In *Static Analysis Symposium, 2004.*

[27] Leonardo de Moura, Nikolaj Bjorner, “Z3: An Efficient SMT Solver”, In *14th International Conference, TACAS 2008.*

[28] Vijay Ganesh, David L. Dill, “A Decision Procedure for Bit-Vectors and Arrays”, In *19th International Conference, CAV 2007,* Berlin, Germany.

[29] Patrick Cousot, Radhia Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”, In *Conference Record of the Sixth Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 238—252, Los Angeles, California, 1977. ACM Press, New York.

[30] Johannes Kinder (2010), *Static Analysis of x86 Executables*, Phd Thesis, Technische Universitat Darmstadt.

[31] GNU objdump, avaiable at, http://www.gnu.org/software/binutils/

[32] Tomonori Izumida, Kokichi Futatsugi, Akira Mori, “A Generic Binary Analysis Method for Malware”, In *International Workshop on Security - IWSEC* , vol. 6434, pp. 199-216, 2010.

[33] Tanenbaum, Andrew S. (2006), *Structured Computer Organization*, Pearson Education, Inc. Upper Saddle River, NJ.

[34] Desel, J., Oberweis, A., Zimmer, T., Zimmermann, “Validation of Information System Models: Petri Nets and Test Case Generation”, In *Proc of SMC 1997*, pp.3401–3406 (1997).