HastFuzz: 선행 분석 결과 기반의 실행 흐름 유도를 통한 퍼징 방법론

조승현†, 이광진†, 최광준, 진호용, 나윤종, 박기웅*

세종대학교 정보보호학과

HastFuzz : Precedence Result Analysis Based Control Flow Induction Fuzzing Scheme

Seung-Hyeon Cho†, Kwang-Jin Lee†, Kwang-Jun Choi, Ho-Yong Jin, Yoon-Jong Na, Ki-Woong Park*

Department of Computer and Information Security, Sejong University.

요 약

퍼징은 소프트웨어에 무작위 데이터를 입력하여 크래시 유발을 유도하고 이를 통해 취약점을 찾기 위한 소프트웨어 테스트 기법으로 활발하게 사용되고 있다. 그러나 기존의 퍼징 방법론에서는 코드분석과 퍼징의 과정을 분리하고 있기 때문에 이미 발견된 취약점이나 소스 코드 분석 등 선행 분석 결과를 통해 도출된 정보를 퍼징에서 활용할 수 없다는 한계가 있었다. 본 논문에서는 선행 분석 결과를 통해 도출된 정보를 퍼징에 활용할 수 있는 HastFuzz를 제안한다. HastFuzz는 공개된 자바스크립트 소스 코드의 AST(Abstract Syntax Tree) 정보에서 AST 노드 간의 상하 관계를 사전에 학습하여 해석기의 문법 검사를 통과하는 AST를 생성한다. 이렇게 생성된 AST는 휴리스틱 기법을 활용한 부분 재생성 과정을 통해 코드 커버리지를 극대화 한다. 이러한 접근 방법은 선행 분석 결과를 통해 도출한 정보를 퍼징에서도 활용할 수 있도록 하며 코드 커버리지 확대 과정에 방향성을 유도할 수 있다. 이를 통해 경계값 조건에 더욱 쉽게 접근할 수 있도록 퍼징을 유도할 수 있으며, 효율적인 퍼징을 가능하게 한다.

I. 서론

퍼징은 소프트웨어 테스트 기법 중 하나로서, 프로그램에 유효하지 않거나 예상치 못한 입력을 생성하고 실행하는 방식으로 버그를 찾아내는 기법이다. 이러한 소프트웨어 버그는 치명적인 취약점으로 이어질 수 있으므로 이를 미리 찾아내어 사전에 차단하는 것은 매우 중요하다. 따라서 많은 보안 연구자들은 다양한 퍼징 기법[1, 2, 3, 4, 5, 6, 7]을 활용하여 소프트웨어 버그를 더욱 효율적으로 찾아내고자 하였다.

웹 브라우저의 경우 자바스크립트를 해석하고 실행하는 자바스크립트 엔진에서 많은 버그가 발생하였기 때문에 자바스크립트 엔진을 대상으로 한 퍼징에 대해 다양한 연구[1, 3, 6, 7]가 선행되었다. 그러나 선행된 연구에서 제시된 퍼징 기법들은 코드 분석과 퍼징을 분리하고 있으므로 이미 발견된 취약점이나 소스 코드 분석 등 경험으로 도출한 정보를 퍼징에서 활용할 수 없다. 더불어 퍼징을 수행하면서 경계값 도달 여부를 확인할 수 없기 때문에 경계값에 대해 충분한 퍼징을 수행할 수 없는 한계점이 있다.

본 논문에서 제시하는 퍼징 방법론은 AST를 확장하여 스크립트 해석기의 문법 검사를 통과할 수 있는 자바스크립트 소스 코드를 생성한

본 연구는 한국연구재단 지원사업(2017R1C1B2003957) 및 2018년도 과학기술정보통신부의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2018-0-00420, API 호출 단위 자원 할당 및 사용량 계량이 가능한 서버리스 클라우드 컴퓨팅 기술 개발)

† 주저자: 세종대학교 정보보호학과 학부과정 (me@shc.me, y2sm2n@gmail.com)

* 교신저자: 세종대학교 정보보호학과 교수 (woongbak@sejong.ac.kr)

다. 또한, 이를 실행하는 자바스크립트 엔진의 코드 커버리지를 극대화하여, 더 높은 확률로 취약점에 도달하는 것을 목표로 한다. 이를 위해 코드 커버리지 극대화 과정에 휴리스틱 기법을 활용하여 경험에 의해 도출한 정보를 퍼징에서도 활용할 수 있도록 하는 방법과 이러한 유도로 인해 얻게 되는 효과에 대해 설명하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존에 선행되었던 자바스크립트 퍼저들의 특징과 한계 및 코드 커버리지에 대해 살펴본다. 3장에서는 기존 퍼저들의 한계점을 개선한 새로운 퍼징 방법론에 대해 제시하고, 마지막으로 4장에서 결론과 향후 연구 방향에 관하여 서술한다.

II. 관련 연구

본 장에서는 자바스크립트 엔진을 대상으로 퍼징을 수행하며 해석기의 문법 검사를 효과적으로 통과하는 방법이 제안된 연구 중 대표적인 두 퍼저에 대해 소개하고, 코드 커버리지를 넓히기 위해 수행되었던 연구를 소개한다.

2.1 기존의 자바스크립트 퍼저

Jsfunfuzz[7]는 대표적인 생성 방식의 퍼저로서, Firefox 브라우저의 자바스크립트 엔진을 테스트하기 위해 개발되었다. Jsfunfuzz는 자바스크립트 언어의 문법 규칙을 내장하고 있으며, 이를 바탕으로 해석기의 문법 검사를 통과하는 올바른 구조의 자바스크립트 소스 코드를 생성한다. 이를 통해 해석기를 통과할 수 있지만, 직접 정의한 문법 관계를 바탕으로 코드를 생성하기 때문에 이를 일반화하여 자바스크립트를 제외한 다른 언어에 적용하기에는 다소 어려움이 있다는 한계가 있다. 또한, JSfunfuzz는 생성된 자바스크립트 소스 코드의 실행 결과를 이후 진행되는 퍼징에 활용할 수 없으며, 무작위로 생성하기 때문에 연속된 조건이 만족해야 해야 도달 가능한 루틴에 대해서는 매우 낮은 확률에 의존할 수밖에 없게 된다. 이처럼 JSfunfuzz는 경계값 문제를 해결하지 못하고 있다.

이러한 JSfunfuzz의 일반화 문제와 경계값 문제를 개선하기 위해 Langfuzz[6]는 사용자의 입력을 변조하는 방법으로 경계값 문제를 해결하였으며, 특정 언어에 구속되지 않으면서도 일반적인 해결책을 찾기 위해 노력하였다. Langfuzz는 특정 언어의 소스 코드가 주어졌을 경우, 이를 파편화하고 학습하여 특정 언어의 문법을 스스로 알아가도록 설계되었다. 또한, 크래시를 발생시켰던 코드는 이후에도 크래시를 발생시킬 확률이 높다는 가정하에 다시 한번 학습을 진행한다. 이처럼 Langfuzz는 학습한 정보들을 바탕으로 스스로 코드를 생성하고 변조하면서 입력값을 생성한다. 이를 통해 일반화된 퍼저를 제작하였지만 복잡한 코드를 생성할 경우 무의미한 코드가 될 확률이 높아지기 때문에 작은 코드를 생성하는 한계가 있으며, 좁은 범위의 경계값에 대해서만 퍼징을 수행하게 된다.

2.2 기존의 코드 커버리지 퍼저

AFL(American Fuzzy Lop)[8]은 유전 알고리즘을 적용하여 코드 커버리지 확대를 지향하는 대표적인 퍼저이다. 퍼징을 수행할 대상의 소스 코드의 모든 분기점마다 진입 여부를 확인할 수 있는 코드를 삽입하고, 퍼징을 수행하며 프로그램의 실행을 추적하는 방법으로 퍼징 대상의 수행 흐름을 수집한다. AFL은 변조 작업을 수행하면서, 전반적인 코드 흐름이 유일하더라도, 새로운 상태 전이를 발생하지 않는 입력값은 버리는 과정을 통해 코드 커버리지를 확대한다. 이러한 확대 방식은 실제로 코드 커버리지를 확대할 수는 있지만, 경계값에 도달하였는지를 확인할 수 없으므로 경계값에 대해 충분한 퍼징을 수행할 수 없다는 한계점이 있다.

2.3 정리

기존의 연구를 통해 그동안 문법 기반 퍼징[3, 4, 6, 7]과 코드 커버리지와 관련된 연구[8, 9]가 활발하게 진행됐음을 확인할 수 있었다. 하지만 대다수 자바스크립트 퍼저에 대한 연구는 해석기의 문법 검사를 통과하는 방법론에 초점을 두어 연구가 진행되어 왔기 때문에 자바스크립트 엔진의 내부 로직에 대한 퍼징에는

다소 미흡한 점이 있었으며, 별도의 소스 코드 분석 행위가 퍼징에 영향을 줄 수 없었다. 따라서 본 연구에서는 휴리스틱 기법을 활용한 부분 재생성 과정을 통해 코드 커버리지를 극대화함으로써, 선행 분석 결과에 의해 도출한 정보를 퍼징에서도 활용할 수 있도록 하여 코드 커버리지 확대 과정에 방향성을 유도하는 방법을 제안한다.

III. HastFuzz 작동 원리

퍼징의 경우 임의적인 입력을 생성하기 때문에 경계값 조건을 찾기 어렵다는 단점이 있었다. 이러한 문제를 해결하기 위해 LangFuzz를 비롯한 대부분의 퍼저는 기존의 사용자 입력에 변화를 일으키는 방식의 접근을 제안하였다. 그러나 이러한 접근은 소프트웨어에 대한 분석과 퍼징의 과정을 분리하기 때문에, 기존의 알려진 취약점이나 소스 코드 분석을 통해 도출한 정보를 퍼징에서 활용할 수 없다는 한계가 있었다.

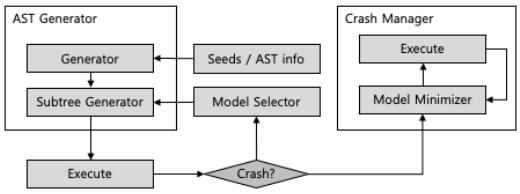

그림 1 HastFuzz 동작 순서

이에 본 연구에서 제안하는 HastFuzz는 통계적인 AST 생성으로 경계값 조건 문제를 해결하고, 코드 커버리지에 기반한 우선순위에 따라 선택된 모델을 부분 재생성하며 퍼징을 수행한다. 이때, 부분 재생성과 모델 선택 과정에 휴리스틱 기법을 적용하여 특정 코드에 대해 더욱 충분한 퍼징을 수행하도록 유도한다. [그림 1]은 제안하는 퍼징 방법론을 도식화한 것으로 이를 통해 소스 코드 분석 등을 비롯하여 선행 분석을 통해 도출한 정보를 퍼징에 적용할 수 있다.

3.1 AST 생성

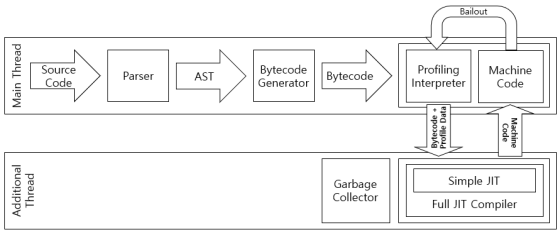

그림 2 ChakraCore의 JavaScript 엔진 구조

브라우저의 자바스크립트 엔진에 대한 퍼징을 효율적으로 수행하기 위해서는 먼저 자바스크립트 엔진의 구조를 파악할 필요가 있다. 자바스크립트 코드가 엔진에서 실행되기 위해서는 [그림 2]와 같은 실행 과정[9]을 거친다. 자바스크립트 엔진의 해석기는 내장된 문법 검사를 통해 입력된 자바스크립트 소스 코드의 실행 가능 여부를 판단하기 때문에, 생성되는 모든 자바스크립트 소스 코드는 이를 만족해야 한다. 이에 본 연구는 이를 효과적으로 만족시키기 위해 AST 확장을 통한 소스 코드 생성 방법을 제안한다.

3.1.1 AST 노드

AST는 소스 코드를 트리 형태로 표현한 구조이며, 이를 바탕으로 실제 실행에 필요한 바이트 코드를 생성한다. AST는 노드와 간선 형태로 표현되며, 각 노드는 종류에 따라 표현식과 설정해야 하는 속성이 결정된다. 각 속성에는 할당 가능한 자식 노드의 유형이 문법 규칙에 의해 정해져 있으며, 생성된 AST는 자바스크립트 소스 코드로 직렬화하는 과정에서 표현식을 참조한다.

3.1.2 AST 노드 관계 정의

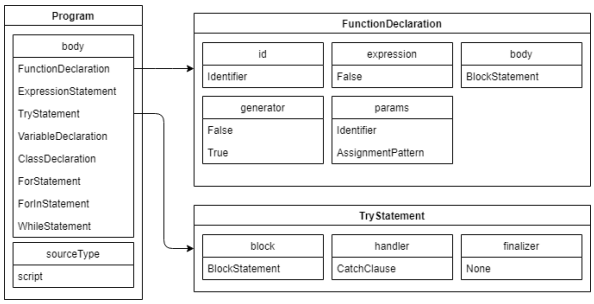

그림 3 AST에서 노드 간의 관계 표현 예시

문법 규칙에 따른 AST 노드의 상하 관계는 [그림 3]과 같이 나타낼 수 있다. AST의 최상

위 노드로 생성되는 'Program' 유형의 노드에는 'body', 'sourceType' 속성이 있으며, 그 중 'body' 속성에는 8종류의 노드가 할당될 수 있음을 나타내고 있다. 이러한 AST 노드 간의 상하 관계 정보는 AST Parser를 이용하여 이미 작성된 자바스크립트 코드를 통해 학습할 수 있다.

HastFuzz는 미리 정의된 AST 노드 관계 정보를 토대로 자식 노드를 선택 생성하고, 자식 노드 또한 이를 반복하여 재귀적으로 트리를 확장해나간다. 이러한 방법을 통해 올바른 문법 구조를 가진 AST를 생성할 수 있다.

3.2 코드 커버리지 극대화

HastFuzz는 AST를 자바스크립트 소스 코드로 직렬화하여 자바스크립트 엔진의 입력으로 실행하여 코드 커버리지를 측정한다. 측정된 코드 커버리지와 AST 쌍을 모델이라 하며, 이러한 모델의 AST에 부분적인 재생성을 하여 새로운 코드 커버리지가 측정되는 모델을 발견해 나간다. HastFuzz는 이러한 과정을 반복하여 코드 커버리지를 극대화 할 수 있다.

3.2.1 코드 커버리지 해시값

HastFuzz는 AST를 자바스크립트 엔진의 입력으로 실행한다. 이때 실행된 모든 분기 주소와 분기별 실행 횟수를 코드 커버리지 정보로 측정한다. 후술할 연산을 효율적으로 처리하기 위해서는 커버리지 정보를 압축할 필요가 있다. 따라서 모든 데이터는 해시함수로 압축하여 관리하며 이러한 압축된 코드 커버리지를 코드 커버리지 해시값으로 정의한다.

3.2.2 AST 부분 재생성

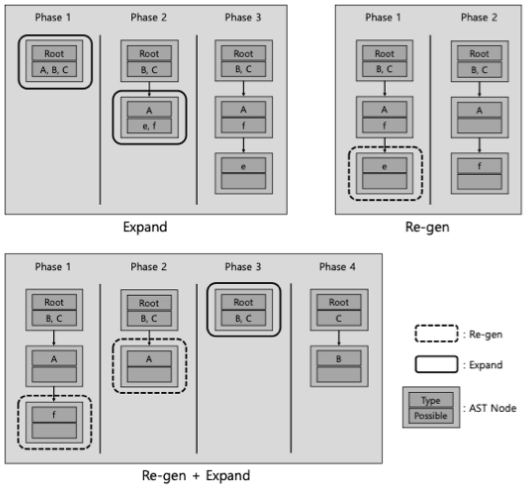

그림 4 상황별 AST 부분 재생성 절차

HastFuzz는 기존 모델의 AST를 부분적으로 재생성하여 새로운 코드 커버리지 해시값이 측정되는 자식 모델을 찾는다. 이때 AST의 반복적인 재생성에도 새로운 AST 구조를 만들어 내는 것을 보장하기 위하여 [그림 4] 같이 각 AST 노드는 할당되었던 자식 노드 유형을 저장하여 노드 확장 단계에서 기존의 노드를 할당하지 않도록 한다.

3.2.3 모델 관리

새로운 코드 커버리지를 효과적으로 발견하기 위해서는 재생성할 모델의 우선순위를 정해야 한다. 본 연구에서는 발견횟수가 가장 적은 모델을 우선 선택하도록 하여 퍼징을 진행하였다.

$$P_1(M) = COUNT(M)$$

퍼징을 수행한 모델M의 우선순위는 위 함수와 같이 동일 모델의 개수로, 최솟값 우선순위 큐에 의해 관리를 하게 된다.

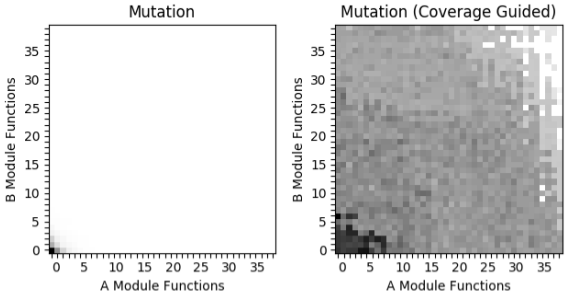

그림 5 코드 커버리지 히스토그램 1

[그림 5]은 간소화된 두 모듈에 대해 퍼징을 16만 회 수행하였을 때 최종적으로 도달한 코드 커버리지를 나타낸 것으로, 각 40개의 경계값 조건의 실행 여부를 측정하였다. 좌측의 히스토그램은 모델 관리 없이 단일 모델에 대해 퍼징을 수행하였다. 우측의 히스토그램은 발견 횟수가 가장 적은 모델을 우선 선택하여 퍼징을 수행하였다. 두 히스토그램을 통해 코드 커버리지 기반의 모델 관리로 더욱 넓은 범위의 코드에 도달할 수 있음을 알 수 있다.

3.3 휴리스틱 기법의 적용

모델의 우선순위 산정과 AST 노드 확장 단계에 휴리스틱 기법을 활용하여 커버리지 극대화 과정에서 특정 실행 분기로 방향성을 부여할 수 있다. 대부분의 분석가들은 이미 알려진 취약점이나 소스 코드 분석을 통해 특정 기능과 상황에서 취약점이 자주 발생할 수 있다는 것을 경험적으로 유추할 수 있다. 이러한 경험에 의한 정보를 퍼징에 반영하여 커버리지 극대화 과정에 방향성을 부여하는 방법을 제안한다.

3.3.1 코드 커버리지 휴리스틱

모델 선택 단계에서, 자바스크립트 엔진의 특정 루틴을 실행한 모델에 높은 우선순위를 부여한다. 이를 통해 특정 경계값 조건에서 더 많은 퍼징을 수행하도록 유도할 수 있다. 이때 측정되는 루틴은 중첩될 수 있다.

$$H_{cov}(M) = \sum_{k=1}^{n} isTriggered(F_k)$$

$$P_2(M) = COUNT(M) - H_{cov}(M)^2$$

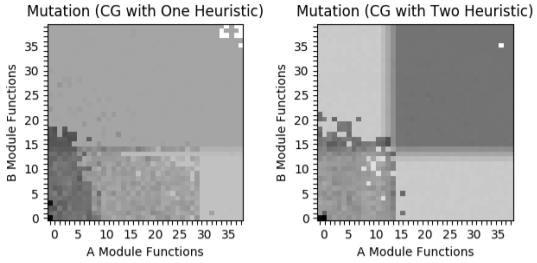

그림 6 코드 커버리지 히스토그램 2

[그림 6]은 간소화된 두 모듈에 대해 퍼징을 12만 회 수행하였을 때 최종적으로 도달한 코드 커버리지를 나타낸 것으로, 좌측 히스토그램은 총 4개(B모듈 4개 분기)의 커버리지 측정을, 우측 히스토그램은 총 9개(A모듈 5개 분기, B 모듈 4개 분기)의 커버리지 측정을 우선순위 산정에 활용한 결과이다.

3.3.2 AST 휴리스틱

노드 확장 및 재생성 단계에서 AST 노드의 속성 할당 시에 참조하는 AST 관계 정보에 가중치를 부여하는 방법을 통해 특정 기능과 조건에 확률적으로 더 적은 횟수의 퍼징으로 도달하도록 유도를 할 수 있다. 이는 코드 커버리지 휴리스틱과 함께 활용할 경우 좀 더 가시적인 결과를 확인할 수 있다.

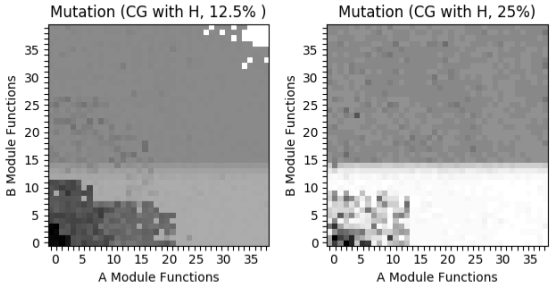

그림 7 코드 커버리지 히스토그램 3

[그림 7]은 간소화된 두 모듈에 대해 퍼징을 12만 회 수행하였을 때 최종적으로 도달한 코드 커버리지를 나타낸 것이다. 우측 히스토그램에 해당하는 테스트에서 선택된 모델에 대해 퍼징을 수행할 때, B 모듈의 코드 커버리지에 영향을 줄 수 있는 도메인을 우선 선택하도록 가중치를 부여하였다. 이로 인해 코드 커버리지 휴리스틱을 통해 유도하려던 특정 경계값 조건에 좀 더 적은 횟수의 퍼징으로 도달하게 된 것을 확인할 수 있었다.

IV. 결론 및 추후 연구

본 논문에서는 휴리스틱을 이용하여 코드 커버리지 확장에 방향성을 가지도록 유도하는 퍼징 방법론을 제안하였다. 기존의 자바스크립트 퍼저들은 해석기의 문법 검사를 통과하기 힘들거나, 이를 통과하더라도 퍼징 결과값을 활용하지 않는 등의 비효율적인 부분이 존재하였다. 또한, 임의의 코드를 생성하여 테스트를 수행하는 접근은 대상 소프트웨어에 대한 분석 행위

와 퍼징 과정을 분리하기 때문에 기존에 알려진 취약점이나 소스 코드 분석을 통해 도출한 정보를 퍼징에서 활용할 수 없다는 한계가 있었다.

이러한 한계점들을 개선하기 위해 본 논문에서 제안한 HastFuzz는 노드 간의 상하 관계를 만족하는 AST를 생성함으로써 해석기의 문법 검사를 통과하고, 휴리스틱 기법을 활용하여 코드 커버리지 극대화에 방향성을 부여하는 접근은 분리되어 있었던 소프트웨어 분석과 퍼징의 과정을 결합하여 취약점 탐색의 효율을 높여줄 것으로 기대한다.

하지만 HastFuzz는 코드를 생성하고 실행하는 과정의 실행 속도 문제와 코드 커버리지를 측정하는 과정에서 오버헤드가 발생할 수 있다는 한계가 있다. 이를 위해, 실행 속도 개선 방안에 대한 추가 연구와 코드 커버리지 확장을 위한 휴리스틱 기법의 효율적인 적용 방안 대한 추가 연구를 진행할 계획이다.

[참고문헌]

[1] TGuo, Tao, et al. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. Informatics and Applications (ICIA), 2013 Second International Conference on. IEEE, 2013.

[2] Dewey, Kyle, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014.

[3] Godefroid, Patrice, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. ACM Sigplan Notices. Vol. 43. No. 6. ACM, 2008.

[4] Godefroid, Patrice, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. Queue 10.1 (2012): 20.

[5] Rebert, Alexandre, et al. Optimizing Seed Selection for Fuzzing. USENIX Security Symposium. 2014.

[6] Holler, Christian, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. USENIX Security Symposium. 2012.

[7] RUDERMAN, J. Introducing jsfunfuzz. Blog Entry. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[8] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[9] ChakraCore Architecture-Overview, https://github.com/Microsoft/ChakraCore/wiki/Architecture-Overview