하드웨어를 활용한 커버리지 기반 Fuzzing 기법 비교 연구

박종현 [1], 오현영 [2*]
[1] 가천대학교 AI · 인공지능학부 석사과정
[2] 가천대학교 AI · 소프트웨어학부 교수

hynmail@naver.com, hyoh@ gachon.ac.kr

Comparative Study of Hardware-Based Coverage-guided Fuzzing Techniques

Jonghyun-Park, Hyunyoung Oh
Dept. of AI · Software, Gachon University

요 약

본 논문은 보안적 취약점을 찾는 기능인 Fuzzing 이 가진 문제점 중에서 소프트웨어 기반 커버리지 기반 Fuzzing 이 지닌 문제점인 속도 및 오버헤드의 문제의 원인이 무엇인지 살펴보고 이에 대한 하드웨어 기반 Fuzzing 을 통한 각각의 해결 접근 방법이 무엇이 있는지 각 하드웨어 해결책의 차이점이 무엇인지 또한 공통적으로 가지고 있는 남아있는 문제점이 무엇이며 이를 해결하기 위한 추후 발전 방향성은 어떤 방향성으로 나아가야 하는지에 대해 언급한다.

1. 서론 및 배경

최근 다수의 소프트웨어 및 하드웨어의 보안 취약점은, 입력 값에 대한 불충분한 유효성 검사 또는 비정상 입력 처리 실패로 인해 발생했다. 예를 들어, 버퍼 오버플로우(Buffer Overflow), 포인터 참조 오류(Null Pointer Dereference) 등이 대표적인 사례이다. 이러한 취약점들을 공격자가 발견하여 악용하게 될 경우 시스템의 신뢰성이 무너지고, 심각한 보안 위반을 유발할 수 있다. 그러나 기존의 테스트 및 검증 방법은 사전에 주어진 정보에 의존하기 때문에, 알 수 없는 미지의 취약점을 충분히 탐지하지 못한다는 한계점이 있다.

이런 배경에서 Fuzzing 기법이 개발자들에게 주목받고 있다. Fuzzing 은 개발자가 소프트웨어의 보안 취약점을 탐지하기 위해 사용하는 자동화된 테스트 기법으로, 프로그램에 다양한 입력을 주입하여 예외나 충돌을 유발하고 이를 분석함으로써 취약점을 탐색한다. 초기의 무작위 기반 Fuzzing 은 구현이 간단하다는 장점이 있지만, 비체계적으로 생성된 입력으로 인해 대부분 의미 없는 경로만을 탐색하고, 복잡한 분기 조건을 갖는 경로에는 도달하기 어려워 탐색 효율이 낮다. 이를 보완하기 위해 등장한 커버리지 기반 Fuzzing 은 실행 경로 정보를 피드백으로 활용하여 더 많은 코드 경로를 탐색할 수 있는 입력을 선택적으로 생성하는 방식으로, 기존 방식의 무작위성을 개선하며 Fuzzing 의 성능을 향상시켜왔다.

그러나 커버리지 정보를 수집하기 위해 매 실행마다 tracing 을 반복해야 하며, 이로 인해 높은 오버헤드와 낮은 실행 속도라는 구조적 한계가 여전히 존재한다. Ding et al[1]은 소프트웨어 기반 커버리지 기반 Fuzzing 에서 tracing 오버헤드가 전체 실행의 90% 이상을 차지하며, 대부분의 테스트가 무의미한 경로만 탐색하게 된다고 지적하였다. 이에따라 최근에는 커버리지 피드백의 효율성과 성능을 동시에 확보하기 위한 방안으로, 하드웨어 기반 Fuzzing 이 새로운 대안으로 주목받고 있다.

본 논문은 하드웨어 기반 Fuzzing 기법인 SNAP[1], GenFuzz[2], Fuzz_E[3]를 대상으로, 각 기법의 구조와 동작 방식, 성능을 중심으로 비교·분석한다. 커버리지 확보 방식, 구현 복잡도, 탐색 속도 등의 측면에서 차이를 평가하고, 하드웨어 기반 Fuzzing 기술의 활용 가능성과 향후 확장 방향을 제시한다.

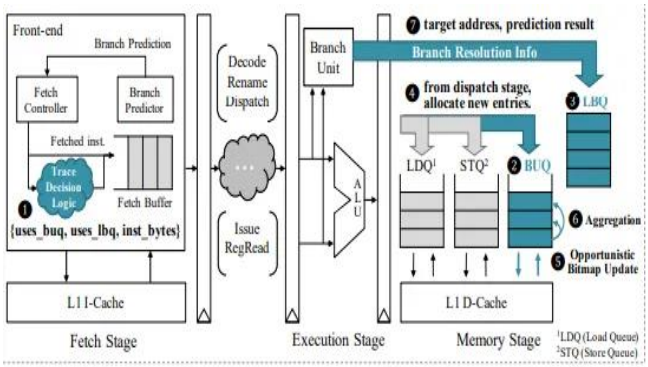
2. 하드웨어 기반 Fuzzing 기법 소개

2.1 SNAP

SNAP[1]은 커버리지 기반의 여러 문제점을 하드웨어를 기반으로 두어 해결하여 커버리지 기반 Fuzzing 을 조금 더 원활하게 수행하고자 설계된 하드웨어 Fuzzing 이다. 커버리지 기반의 Fuzzing 은 그 자체가 큰 오버헤드를 발생시키고 실행 속도를 늦춰 Fuzzing 의 효과를 떨어뜨리며 그 결과적으로 컴퓨팅 리소스가 낭비되어 수만 대의 머신으로 확장되는 지속적인

* 교신저자

Fuzzing 서비스가 더욱 확대될 수 있다. 예를 들어 유명한 Fuzzing 중 하나 인 AFL 도 소스 코드 계측으로 인해 70%의 오버 헤드를 겪으며 심지어 바이너리 전용 프로그램의 경우 QEMU 모드에서 무려 1300%의 오버헤드를 겪는다. 이러한 커버리지 기반의 문제를 해결하기 위해 SNAP 은 커버리지 기반 Fuzzing 을 위해 아래 그림 1 인 하드웨어 가속 tracing architecture 를 설계하여 이를 해결했다.



(그림 1) SNAP 의 하드웨어 구조도 SNAP 구조도[1] 참고

우선 1 Trace Decision Logic(1) 과정은 Fetch 단계에서 각 명령어을 검사해 tracing 대상인지 판별하고 추적 태그(uses_buq, uses_lbq)를 붙인다. (2)그 후 태그 된 instruction 중에서 커버리지 추적 대상이 commit 된다면 이를 BUQ 에 삽입한다. (4)BUQ 에 삽입된 명령어를 FSM 로 비트맵을 안전하게 업데이트한다. 이 과정은 BUQ 의 메모리 병렬 접근 문제 방지를 위한 구조이며 s_init 은 업데이트할 비트맵 주소를 계산한다. 이후 s_load 과정에서 해당 위치에서 현재 값을 읽고 s_store 값을 1 증가시키고 저장한다. 마지막으로 s_done 을 사용해서 완료 처리하고 큐에서 제거한다. (5)비트맵 업데이트는 CPU 의 자원을 사용할 때만 수행되며, 우선 순위가 가장 낮다. (6)동일한 주소에 대한 연속 업데이트를 하나로 병합하여 store 의 수를 줄이는 과정이다. (7)한편 분기 명령(branch)가 commit 될 경우 해당 분기 정보는 LBQ 에 삽입한다. 이후 분기 명령의 실제 타겟 주소 예측 결과를 LBQ 로 전달한다.

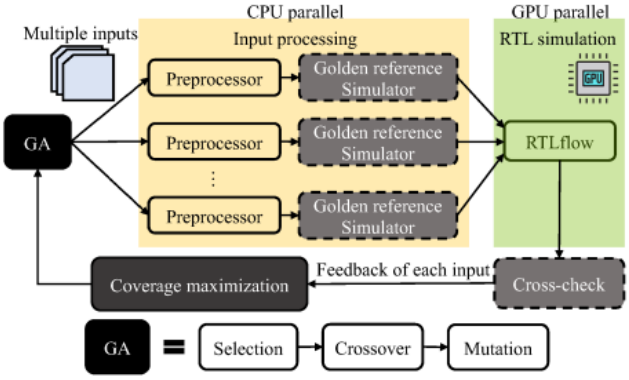
각각 BUQ, LBQ 의 과정은 퍼저(Fuzzer)에게 정보를 제공한다. BUQ 를 통한 커버리지 비트맵 정보는 코드의 어디를 실행하였는지에 대한 정보를 제공해주고 LBQ 를 통한 실행 분기 히스토리 정보는 어떻게 실행 경로를 따라갔는지에 대한 정보를 제공해준다. 이 두가지 중요한 정보가 따로 소스 코드나 바이너리 삽입 없이, 하드웨어에서 수집되는 것이 SNAP 의 핵심 강점이다.

SNAP 의 하드웨어 기반 설계 성능은 소프트웨어 기반 설계 중 하나인 AFL 과 실험의 결과를 비교했을 때 같은 메모리 크기인 64KB 에서 거의 3.14%의 오버헤드를 발생시켜 AFL 이 599.77%에 비해 상당히 우수한 오버헤드 제어 성능을 보인다는 것이 나타났다. 메모리의 크기를 늘려도 여전히 SNAP 이 더욱 빠른 모습을 보이며 하드웨어 설계인 SNAP 이 소프트웨어에 비해 더욱 우수하다는 것을 보여준다. 이를 통해 SNAP[1]이 실시간 Fuzzing 에 조금 더 용이하다는 것을 보여준다. 또한 Preserving Trace 부분에서 충돌율도 AFL 과 마찬가지로 8.94%의 충돌율을 보여준다. 이는 SNAP 이 하드웨어 기반임에도 불구하고 AFL 과 실행 경로 구분의 정확성 측면에서 소프트웨어 Fuzzing 과 동등한 성능을 지닌다.

SNAP[1]은 하드웨어로 설계한 Fuzzing 임에도 불구하고 소프트웨어와 동일한 정확도를 보이며 소프트웨어 Fuzzing 의 문제점인 오버헤드와 속도를 개선한 점에서 매우 훌륭하다. 하지만 실험이 단일 코어 환경 기준이고 멀티 코어 환경에서의 실험이 진행되지 않은 점과 구체적인 RTL 구현 방법에 대한 언급이 없다는 아쉬운 점이 남아있다.

2.2 GenFuzz

GenFuzz[2]는 기존의 커버리지 기반의 Fuzzing 의 문제점 중 하나인 무작위 입력 값 생성으로 인한 Fuzzing 의 효율성 및 속도 저하에 대해 주목하고 해결하고자 한 연구이다. GenFuzz[2]는 Genetic Algorithm(GA)과 하드웨어의 병렬 구조를 설계하여 소프트웨어와 하드웨어의 기능을 합하여 Fuzzing 의 문제를 해결하기 위한 연구이다. 즉 완전한 하드웨어 만을 사용한 Fuzzing 보다는 하드웨어와 소프트웨어의 결합을 통해 속도를 향상시킨 Fuzzing 알고리즘으로 해석된다.



(그림 2) GenFuzz[2]-의 전체 프로세스 구조도

GenFuzz[2]는 GPU 기반 RTL 시뮬레이션과 Genetic Algorithm(GA)을 통합하여 하드웨어 Fuzzing 을 병렬 구조로 수행한다. 그림 2 는 GenFuzz[2]의 전체 구조도다. 우선 (1) GA 단계에서는 이전 반복에서 수집된 입력들의 적합 값을 기반으로 유전 연산(selection, crossover, mutation)을 수행하여 새로운 입력을 생성한다. selection 은 Roulette-Wheel Selection 방식으로 이루어지며, 적합성이 높을수록 선택 확률이 높다. 선택된 부모 입력은 one-point crossover 를 통해 gene 을 교환하며, 이후 insert, delete, replace 중 하나의 변이 연산이 적용되어 자손이 생성된다. 입력은 명령 단위의 gene 시퀀스로 표현되며 variable-length 구조를 가지므로 탐색 공간을 유연하게 구성할 수 있다.

그 후 (2) 입력 처리 단계에서는 GA 로 생성된 입력을 자극으로 컴파일하고 시뮬레이션 실행 파일을 준비한다. 이 과정은 CPU 코어 단위로 병렬 수행되

며, I/O 및 컴파일 시간이 지연 요인이 될 수 있다. 이후 (3) RTL 시뮬레이션 단계에서는 입력들을 GPU 기반 RTL 시뮬레이터인 RTLflow 로 병렬 실행한다. 이 과정에서 각 입력은 RTL 디자인 위에서 동시에 시뮬레이션 되며, 기존의 단일 입력 방식보다 높은 처리량을 달성한다. 시뮬레이션이 완료되면 (4) cross-check 단계에서 reference simulator 와의 결과를 비교하여 정합성을 확인한다. 마지막으로 (5) coverage-maximization 단계에서는 각 입력의 커버리지 맵을 기반으로 delta coverage 와 progressive coverage 를 계산하고, 이를 정규화하여 fitness 값을 산출한다. 이 값은 다음 반복의 GA 선택 기준으로 다시 활용된다.

GenFuzz[2]는 기존 Fuzzing 기법 대비 시뮬레이션 속도에서 큰 향상을 보인다. GPU 기반의 RTLflow 를 도입함으로써, BoomCore1 기준으로 100% 커버리지 도달 시간이 기존 DIFUZZRTL 의 172,800 초에서 2,160 초로 줄어들며 최대 80 배까지 속도 개선이 이루어진다. 동일한 instruction 수를 기준으로 비교할 때에도 GenFuzz 는 최대 2.1 배 더 많은 커버리지를 탐색하며, fuzzing 성능을 빠르게 수렴시킨다. 또한 GA 는 평균적으로 20 회 반복 이내에 수렴하여 입력 생성과 평가의 연산 부담을 제한된 시간 내에 마무리할 수 있도록 한다.

하지만 GenFuzz[2]에도 단점은 존재한다. 입력 개수가 많아질 경우 입력 처리와 coverage 분석 과정에서 오버헤드가 누적될 수 있으며, 특히 coverage point 가 많을 경우 GA 의 알고리즘 계산 복잡도는 O(n × cov_size)로 선형 증가하게 된다. 또한 RTLflow 기반 병렬 시뮬레이션은 GPU 자원 활용에는 강점을 가지나, 복잡한 RTL 설계나 GPU 자원 부족 상황에서는 기대만큼의 병렬 이점을 얻지 못할 수 있다. cross-check 는 병렬화 되지 않으며, GA 가 지역 최적의 결과 값에 수렴할 경우 coverage 확장에 한계가 생길 수 있다는 구조적 약점도 함께 존재한다.

2.3 Fuzz_E

앞서 소개한 두 가지 방식은 하드웨어를 기반으로 Fuzzing 을 사용하여 소프트웨어 코드 내의 결함을 살펴보는 방식이라면 Fuzz_E[3]는 하드웨어 설계에 대한 Fuzzing 을 실행하는 방법이다. Fuzz_E 는 RTL 접근 없이 완전한 블랙 박스 상태에서도 하드웨어 Fuzzing 을 가능하게 하는 새로운 기법으로, 내부 회로 신호 대신 FPGA 내부 전압 변동(voltage fluctuation)을 활용하여 커버리지 정보를 간접적으로 추정한다. 이를 위해 전압 센서를 구성하는 TDC(Time-to-Digital Converter)를 FPGA 에 삽입하여, 각 입력에 대해 512 클럭 사이클 동안 전압의 변화를 정수 값으로 기록한다. 전압 변화는 회로 내 활성화된 경로와 모듈에 따라 달라지므로, 이 전압 trace 를 통해 입력이 얼마나 새로운 회로 경로를 활성화했는지 판단할 수 있다. Fuzz_E 는 이 전압 trace 를 클러스터링 하여 피드백 메커니즘으로 활용하고, 기존과 다른 트레이스는 흥미로운 입력으로 간주하여 Fuzzing 입력군에 추가한다.

이 방식은 기존 회색박스 기반의 RFUZZ 와 달리 RTL 코드에 접근할 필요가 없으며, 복잡하고 암호화된 IP 블록을 포함하는 대규모 SoC 에도 적용 가능하다는 장점이 있다. 또한, RFUZZ 와 같은 방법이 회로 크기에 따라 높은 오버헤드를 유발하는 반면, Fuzz_E 는 전압 센서만 사용하는 방식이므로 오버헤드가 일정하며 대규모 설계에 더욱 적합하다. 실험 결과에서도 Fuzz_E 는 랜덤 또는 피드백 없는 Fuzzing 방식보다 더 나은 커버리지를 확보했으며, 특히 RocketTile 과 같은 대규모 디자인에서 효과가 두드러졌다.

한편 이 논문은 실질적인 설계 적용이나 시스템 레벨에서의 통합 관점이 부족하다는 점에서 한계를 가진다. Fuzz_E 의 전압 기반 커버리지 추정 방식은 이론적으로 유효함을 보여주지만, 구체적인 하드웨어 아키텍처, 모듈 간 연결 방식, 센서 배치 전략 등 실제 설계자 입장에서 필요한 구현 세부사항은 거의 제시되어 있지 않다. 또한, 피드백 계산에서 사용하는 클러스터링 알고리즘의 연산 복잡도에 대한 분석은 부재하며, 이러한 연산이 Fuzzing 의 실시간성 또는 실용성에 어떤 영향을 미치는지에 대한 논의도 부족하다. 가장 큰 문제는 Fuzz_E 가 도입한 전압 기반 피드백이 실제 버그 탐지 성능과 얼마나 밀접하게 연관되어 있는지를 입증하는 실험이 미흡하다는 점이다. 커버리지 증가만을 성과로 강조하며, 발견된 실제 취약점이나 결함에 대한 사례 분석이 결여되어 있어 실질적 유효성 검증이 제한적이다. Fuzz_E 는 비공개 구조와 구현 세부 부족 등 한계가 있었으며, 이러한 문제를 개선한 FuzzWiz[4]가 제안되었다. FuzzWiz[4]는 하드웨어를 소프트웨어 모델로 추상화하여 다양한 퍼저를 적용 가능하게 하였고, 퍼저 간 커버리지를 정량적으로 비교할 수 있는 프레임워크를 제공한다.

3 하드웨어 기반 Fuzzing 의 비교 분석

기준	SNAP	GenFuzz	Fuzz_E
활용 하드웨어	SoC	GPU	FPGA 실험
하드웨어 활용 범위	Coverage trace	병렬 시뮬레이션 기반 입력 진화	회로 신호 기반 블랙박스 검증
Fuzzing 대상	SW	SW	HW
오픈 소스	X	O	X

(표 1) 하드웨어 Fuzzing 의 성능 비교 분석

세 가지 하드웨어 기반 Fuzzing 기법은 적용 대상, 하드웨어 구현 방식, Fuzzing 의 초점, 소스 공개 여부 등에서 뚜렷한 차이를 보인다.

먼저, SNAP[1]은 기존 소프트웨어 기반 커버리지 추적의 성능 병목을 해결하기 위해, 하드웨어 수준에서 직접 커버리지 정보를 수집할 수 있도록 설계된 전용 트레이서이다. 소프트웨어 도구나 코드 삽입 없이 CPU 파이프라인 내부에서 추적을 수행하며, 모든 Fuzzing 과정은 순수 하드웨어 기반으로 실행된다. 특히 RISC-V BOOM 프로세서 기반으로 구현되어, 소스 코드의 유무와 관계없이 다양한 바이너리 프로그램에 적용 가능

하다는 점에서 기존 방식과 뚜렷한 차별성을 가진다. GenFuzz[2]는 GPU 병렬 구조를 활용하여, 다수의 입력 집합을 동시에 평가할 수 있는 병렬 시뮬레이션 기반 입력 진화 시스템을 구현하였다. 이 기법은 Genetic Algorithm 을 활용한 소프트웨어적 Fuzzing 전략과 GPU 기반 하드웨어 병렬처리 구조를 결합함으로써, 소프트웨어 Fuzzing 의 성능 병목을 개선하고 처리량을 대폭 향상시켰다. 또한 해당 알고리즘은 GitHub 를 통해 공개된 오픈 소스로, 실제 적용 및 확장 가능성 측면에서도 강점을 지닌다. 마지막으로, Fuzz_E[3]는 하드웨어 구조 자체를 Fuzzing 의 대상으로 삼는 회로 수준의 블랙박스 검증 기법이다. 기존의 소프트웨어 Fuzzing 처럼 코드 기반 분석이 아닌, 전기적 신호 및 트랜잭션 레벨의 반응 분석을 통해 회로 결함을 탐지한다는 점에서 독자적인 구조를 가진다. 이 방식은 실험을 위해 FPGA 기반의 하드웨어 환경을 설계하고 적용해야 하며, 분석 대상 자체가 하드웨어라는 점에서 기존 Fuzzing 과 개념적 차이를 가진다. 오픈소스 여부에서는 SNAP[1]과 Fuzz_E[3]가 비공개 구조인 반면, GenFuzz[2]는 알고리즘 전반을 공개하고 있어 활용 접근성에서도 차이를 보인다.

4. 한계점 및 향후 연구

SNAP, GenFuzz, Fuzz_E 는 하드웨어 기반 Fuzzing 의 성능 향상과 커버리지 개선에 기여했지만, 각 기법은 기술적 제약도 지닌다. SNAP 은 SoC 내 하드웨어 트레이서를 통해 실시간 커버리지 수집이 가능하지만, RISC-V BOOM 아키텍처에만 적용되어 다양한 ISA 환경에서의 범용성이 부족하며, 다중 코어 환경 확장도 미비하다. GenFuzz[2]는 GPU 병렬 시뮬레이션을 통해 입력 생성을 가속화했으나, 단일 GPU 환경에만 한정되고, 멀티 스레딩이나 분산 구조에 대한 검증 및 하드웨어 연동 구조 설명이 부족하다. Fuzz_E[3]는 회로 단위 Fuzzing 이라는 새로운 접근을 제시하지만, FPGA 환경에 종속되어 있고, 신호 해석 정확도나 센서 구성에 대한 구체적 논의가 부족하다.

따라서 SNAP 은 다양한 아키텍처에서 활용 가능한 모듈형 트레이서 IP 및 표준 인터페이스 개발이 필요하다. GenFuzz[2] 병렬 평가 구조를 멀티-GPU 및 분산 환경으로 확장하고, 유전 알고리즘 기반 입력 생성의 하드웨어 연동 방식을 구체화해야 한다. Fuzz_E[3]는 신호 기반 커버리지 해석의 신뢰성을 높이기 위한 센서 배치 최적화와 FPGA 외 환경에서도 적용 가능한 회로 분석 모듈 개발이 요구된다. 또한 SNAP[1]의 커버리지 수집, GenFuzz[2]의 병렬 처리, Fuzz_E[3]의 회로 중심 분석을 결합한 하이브리드 Fuzzing 구조 설계도 향후 연구로 제안할 수 있다.

5. 결론

본 논문에서는 SNAP[1], GenFuzz[2], Fuzz_E[3] 세 가지 하드웨어 기반 Fuzzing 기법을 비교·분석하고, 각 기법의 구조, 장단점, 적용 대상에 대해 고찰하였다. 이들은 서로 다른 하드웨어 자원(CPU, GPU, RTL)을 활용하여, 기존 소프트웨어 기반 Fuzzing 의 커버리지 한계와 성능 병목을 보완하고자 한다는 공통점을 가진다. 그러나 구현 복잡도, 디버깅 어려움, 실제 환경 적용의 한계 등 구조적 제약도 존재한다. 향후 연구는 이러한 문제를 개선하고, 하드웨어 기반 Fuzzing 의 실용성과 확장성을 높이는 방향으로 나아가야 할 것이다.

특히 SNAP[1]은 하드웨어만으로 Fuzzing 을 수행할 수 있는 구조적 완성도와 낮은 오버헤드를 통해 성능 측면에서 강점을 보인다. 예를 들어, 64KB 입력 환경에서 SNAP[1]은 3.14%의 오버헤드를 기록하며, 소프트웨어 기반 AFL 대비 약 190 배의 실행 효율을 달성하였다. GenFuzz[2]는 GPU 병렬 구조와 오픈소스 기반 확장성을 통해 실용성과 활용도가 높다. Fuzz_E[3]는 회로 자체를 대상으로 하는 독자적인 접근이지만, FPGA 환경에 국한되어 있고 구조 설명이 부족한 점은 한계로 남는다. 세 기법은 각각의 기술적 강점을 바탕으로 하드웨어 기반 Fuzzing 기술의 발전에 중요한 기여를 할 수 있을 것으로 기대된다.

사사문구

이 논문은 2025 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구 결과 임 (No. RS-2024-00337414, SW 공급망 운영환경에서 역공학 한계를 넘어서는 자동화된 마이크로 보안 패치 기술 개발).

참고문헌

[1] Ding, R., Kim, Y., Sang, F., Xu, W., Saileshwar, G., & Kim, T., "Hardware Support to Improve Fuzzing Performance and Precision," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[2] Zhang, J., Dong, Z., & Liu, Z., "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[3] Qian, Z., Zhang, T., & Jin, Y., "Fuzz Wars: The Voltage Awakens – Voltage-Guided Blackbox Fuzzing on FPGAs," *Proceedings of the 30th USENIX Security Symposium*, 2021.

[4] Vaibhav Sharma, Benjamin Tan, Daniel Holcomb, “FuzzWiz: A Hardware Fuzzing Framework for Evaluating and Comparing Fuzzers Using Software Abstractions,” *arXiv preprint arXiv:2410.17732*, 2024.