FuCE: Fuzzing+Concolic Execution guided Trojan Detection in Synthesizable Hardware Designs

MUKTA DEBNATH*, Indian Statistical Institute, India
ANIMESH BASAK CHOWDHURY*, New York University, USA
DEBASRI SAHA, A.K. Chowdhury School of IT,University of Calcutta, India
SUSMITA SUR-KOLAY, Indian Statistical Institute, India

High-level synthesis (HLS) is the next emerging trend for designing complex customized architectures for applications such as Machine Learning, Video Processing. It provides a higher level of abstraction and freedom to hardware engineers to perform hardware software co-design. However, it opens up a new gateway to attackers to insert hardware trojans. Such trojans are semantically more meaningful and stealthy, compared to gate-level trojans and therefore are hard-to-detect using state-of-the-art gate-level trojan detection techniques. Although recent works [19, 20] have proposed detection mechanisms to uncover such stealthy trojans in high-level synthesis (HLS) designs, these techniques are either specially curated for existing trojan benchmarks or may run into scalability issues for large designs. In this work, we leverage the power of greybox fuzzing combined with concolic execution to explore deeper segments of design and uncover stealthy trojans. Experimental results show that our proposed framework is able to automatically detect trojans faster with fewer test cases, while attaining notable branch coverage, without any manual pre-processing analysis.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; **Hardware reliability**; **Test-pattern generation and fault simulation**; *Software tools for EDA*; • **Security and privacy → Malicious design modifications**.

Additional Key Words and Phrases: Hardware Trojan, High-level Synthesis, Greybox fuzzing, Symbolic Execution

ACM Reference Format:
Mukta Debnath, Animesh Basak Chowdhury, Debasri Saha, and Susmita Sur-Kolay. 2022. FuCE: Fuzzing+Concolic Execution guided Trojan Detection in Synthesizable Hardware Designs. 1, 1 (January 2022), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent times, hardware designers are increasingly using commercial-off-the-shelf (COTS) third-party intellectual property (3PIP) for designing complex architectures [40]. For ease of automated design space exploration and functional verification, designers are adopting HLS framework like SystemC, synthesizable C/C++. As the designs have become increasingly more complex, chip designers build complex system-on-chips using 3PIPs of required functionalities and integrate them in-house. Post integration, these system-on-chips are outsourced to off-shore fabrication facilities. This typical chip design flow has enabled 3PIP vendors to maliciously inject stealthy bugs at a higher abstraction level

*Equal contribution while at Indian Statistical Institute

Authors' addresses: Mukta Debnath, mukta_t@isical.ac.in, Indian Statistical Institute, Kolkata, India; Animesh Basak Chowdhury, abc586@nyu.edu, New York University, New York, USA; Debasri Saha, debasri_cu@yahoo.in, A.K. Chowdhury School of IT,University of Calcutta, Kolkata, India; Susmita Sur-Kolay, ssk@isical.ac.in, Indian Statistical Institute, Kolkata, India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.
Manuscript submitted to ACM

that can later be exploited by the attacker to cause malfunction. Thus, verifying the security aspects of such 3PIPs is primarily important for the in-house integrator apart from ensuring correct functionality.

Hardware Trojans at high-level synthesized designs have been recently explored in [34], where the attacker injects malicious functionality either by leaking crucial information or corrupt final output of design-under-test (DUT). Although there exists a plethora of work to detect hardware trojans at register-transfer level (RTL) or gate-level designs, these detection mechanisms are biased towards certain types of gate-level trojans. The trojans inserted at a higher abstraction level are semantically meaningful and stealthy, but difficult for the defender to precisely detect any abnormal functionality. An attacker can manifest a trojan by simply adding an RTL statement that uses hardware blocks designed for other functionalities. This motivates performing security testing on high-level synthesized designs.

In this article, we propose a scalable trojan detection framework based on fuzzing and concolic execution (FuCE): combines greybox fuzzing [39] and concolic execution [29] in a synergistic way to alleviate the downsides of those two approaches in standalone mode. We show that prior state-of-the-art trojan detection works are heavily confined to the type and functionality of trojans, and fail on subtly modified trojan behavior. Our primary contributions in this paper are two-fold:

(1) a hybrid test generation approach combining the best of both worlds: greybox fuzzing and concolic testing — our proposed framework complements both the techniques extenuating the problems associated with standalone approaches;
(2) to the best of our knowledge, ours is the first work combining fuzzing with concolic testing to reach deeper segments of HLS designs without hitting the scalability bottleneck.

The rest of the paper is organized as follows: Section 2 outlines the background and the prior related works in the area of trojan detection. Section 3 describes the current limitations and challenges using state-of-art techniques. In Section 4, we propose our FuCE framework and show the efficacy of results in Section 5. Section 6 presents an empirical analysis of the results and concluding remarks appear in Section 7.

2 BACKGROUND

2.1 Overview of Hardware Trojan

For decades, the silicon chip was considered as the root-of-trust of a complex system. The assumption was that the hardware blocks and modules are trustworthy and functions are exactly as specified in the design documentation. However, at the turn of this millennium, researchers have extended the attack surface to the underlying hardware, and showed that it can be tampered to gain privileged information and/or launch denial-of-service attacks. This has disrupted the root-of-trust assumption placed on hardware. The core philosophy of hardware trojan is to insert a malicious logic in the design and bypass the functionality verification of the design. The malicious logic is activated by the attacker's designed input. Since the last decade, hardware trojan design and detection have been extensively studied in the field of hardware security. Security researchers have proposed numerous trojan threat models and novel ways of detecting them. In [37], the authors have shown hardware trojans can be inserted in various levels of the chip design life-cycle. Thus, security testing of a design becomes extremely important before being passed on to the next stage.

2.2 Threat model

We have outlined our threat model in Figure 1. We assume in-house designers and engineers are trusted entities who are primarily responsible for developing complex system-on-chip (SoC) modules. A number of third-party hardware

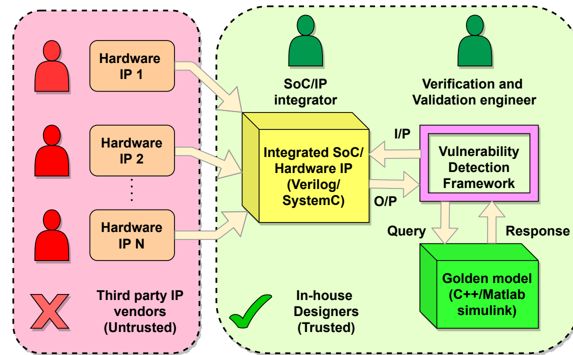

Fig. 1. Threat model showing untrusted third-party IPs are used for developing customized complex system-on-chip modules

IPs are procured from third-party vendors for developing such complex SoC design. However, 3PIPs are untrusted (not developed in-house) and may have hidden backdoors and vulnerabilities. Therefore, it is important for the in-house designers to locate the presence of malicious functionalities in such 3PIPs. For validation purpose, we assume that the in-house designers have access to functionally correct cycle accurate behavioural model of the SoC design (in the form of C++/Simulink model). Our threat model is in line with several earlier works such as [19, 20].

2.3 Security Testing

In the software community, security testing is one of the mandated steps adopted by the practitioners to analyze and predict the behaviour of the system with unforeseen inputs. This has helped develop robust software that are immune against a variety of attacks like buffer-overflow [12], divide by zero [8], arithmetic overflow [17]. We describe next two well-known security testing methodologies that are widely used for the same:

2.3.1 Greybox fuzzing. Fuzz testing is a well known technique in software domain for detecting bugs. Greybox fuzzing [39] involves instrumenting the code segments at the point-of-interest, and generate interesting test vectors using evolutionary algorithms. Instrumentation injects markers in the design code which post compilation and execution can track whether a test-case has reached the marker location. A fitness function is used in order to evaluate the quality of a test-vector. Typically, greybox fuzzing is used to improve branch-pair coverage [39] of the design, therefore the codes are annotated at every basic block. A test-vector is regarded as interesting, if it reaches a previously unexplored basic block, or hits it for a unique number of times. The fuzz engine maintains a history-table for every basic block covered so far, and retains interesting test vectors for further mutation/crossover. The test generation process completes once the user-defined coverage goal is achieved. There exists a plethora of works [10, 31, 36] that have improved the performance of greybox fuzzing by augmenting various program analysis techniques. Popular coverage-guided greybox fuzzing (CGF) engine like American fuzzy lop (AFL) [39] have been able to detect countless hidden vulnerabilities in well-tested softwares.

2.3.2 Concolic Testing. Concolic testing [29] is a scalable way of performing symbolic execution on a program with certain inputs considered concrete, and the rest are symbolic inputs. Symbolic execution in general suffers from scalability issues since the number of path constraints generated, are exponential in terms of the number of conditional statements. In order to avoid costly computations, concolic execution executes the program along the path dictated by

concrete input and fork execution at branch points. The path constraints generated in concolic execution have reduced the number of clauses and variables, thereby making it easier for solvers and can penetrate deep into complex program checks. *Driller* [31] and symbolic executer (S2E) [9] are examples of engines adopting this approach.

2.4 Testing for Hardware Trojan detection

Testing based trojan detection is a well studied problem in the recent past. In earlier works [4, 5, 7, 15, 28], authors have assumed that the trojan netlist contain a *rare* logic value and/or switching activity at certain nodes. Therefore, the test generation was tuned to excite such nodes in a netlist. Later, researchers used concolic testing approaches to detect trojans in behavioural level RTL designs [2, 3, 13, 24]. The objective of performing concolic testing is to penetrate into deeper conditional statements of a HDL program to expose the trojan behaviour. Although the techniques seem to work well on trojan benchmarks [30], employing concolic testing without an efficient search heuristic and a target of interest to cover, results in multiple SAT solver calls taking considerable time for test vector generation. With recent success of coverage-guided greybox fuzzing in software domain, it has been recently adopted in [16, 19, 32] for detecting trojans in hardware design. Concolic test generation for trojan detection in high level designs have only been proposed recently in [20, 21]. We summarize works related to ours in Table 1.

Table 1. Works on Test-based Trojan Detection in Hardware designs

Work	Abstraction level	Technique used	Benchmarks	Golden-model available?
Chakraborty *et al.* [7]	Gate level	Guided ATPG	ISCAS85, ISCAS89	✓
Banga *et al.* [4]	Gate level	Novel DFT	ISCAS89	✓
Saha *et al.* [28]	Gate level	Genetic algorithm + SAT formulation	ISCAS85, ISCAS89	✓
Chowdhury *et al.* [5]	Gate level	ATPG binning + SAT formulation	ISCAS85, ISCAS89, ITC99	✓
Huang *et al.* [15]	Gate level	Guided ATPG	ISCAS85, ISCAS89	✓
Liu *et al.* [25]	Gate level	Genetic algorithm + SMT formulation	TrustHub [33]	✓
Ahmed *et al.* [3]	Register transfer level	Concolic testing	TrustHub [33]	✓
Ahmed *et al.* [2]	Register transfer level	Greedy concolic testing	TrustHub [33]	✓
Cruz *et al.* [13]	Register transfer level	ATPG + Model checking	TrustHub [33]	✓
Liu *et al.* [24]	Register transfer level	Parallelism + concolic testing	TrustHub [33]	✓
Pan *et al.* [27]	Register transfer level	Reinforcement learning	TrustHub [33]	✓
Le *et al.* [19]	HLS/SystemC	Guided greybox fuzzing	S3C [34]	✓
Bin *et al.* [20]	HLS/SystemC	Selective symbolic execution	S3C [34]	✓
FuCE (Ours)	HLS/SystemC	Greybox fuzzing + Concolic Execution	S3C [34]	✓

2.5 Trojan detection in high-level design

With high level synthesis becoming the new trend for designing customized hardware accelerators, only few works [26, 34] have studied hardware trojan and security vulnerabilities in HLS designs and proposed preliminary countermeasures to detect them. In prior work[19], Trojan inserted in high-level design is called synthesizable hardware Trojan (SHT) as it gets manifested as malicious backdoor in the hardware design. Therefore, the problem of Trojan detection in low level RTL design can be appropriately abstracted as finding SHT in high-level design. Till date, [19] and [20] have systematically addressed the problem of Trojan detection in HLS designs. In [19], the authors have tuned the software fuzzer AFL [39] for S3C Trojan benchmark characteristics and showed that their technique outperforms the vanilla AFL. The modified AFL called AFL-SHT (AFL-SHT), introduces program-aware mutation strategy to generate meaningful test vectors. In [20], the authors identify the additional overhead of concolic testing from usage of software libraries, and

have restricted the search space within the conditional statements of design. They named their automated prototype as SCT-HTD (SCT-HTD), based on S2E [9]. In the next section, we focus on studying the Trojan characteristic embedded in high-level synthesized design and evaluate the efficacy of existing detection techniques.

3 LIMITATIONS AND CHALLENGES

We present our motivating case-study on a real-time finite state machine implementation. We use a system controller design mimicking a typical hardware functionality and highlight limitations of existing techniques to discover the trojan behavior.

3.1 Motivating case-study

Listing 1. Motivating example

```
1  int controller(int stateA, int stateB)
2  {
3    unsigned long long cycle=0,input = 101 ;
4    bool switchA = FALSE;
5
6    stateA+=input ; stateB-=input ;
7    if(stateA != 23978 || stateB != 5687)
8      exit(0) ;
9
10   while(input){
11       if(stateA == 23978 && stateB == 5678 && switchA == FALSE)  // Branch 1
12       {
13         switchA = TRUE ; cycle++ ;
14         swap(stateA,stateB) ;
15         if(cycle >= 2^20-1)  //Trojan
16         {
17             switchA = FALSE ;
18             swap(stateA,stateB) ;
19         }
20       }
21       else if(switchA == TRUE)  // Branch 2
22       {
23         stateA += 100 ; stateB -= 100 ;
24         if(stateA == 23978 && stateB == 5678)  // Branch 3
25             switchA = FALSE ;
26       }
27       scanf("\n%d",input) ;
28   }
29 }
```

In Listing 1, we have a code-snippet of a typical controller accepting state information `stateA` and `stateB`. The controller first checks if the `stateA` and `stateB` are set to the values 23978 and 5678, respectively (line 7). Once the guard condition is satisfied, it enters a while loop, and check for the values of `stateA`, `stateB` and `switchA`. The loop traversed for the first time, satisfies *Branch* 1 (line 11) and swaps the values of `stateA` and `stateB`, setting `switchA` to TRUE. In subsequent iterations, *Branch* 2 is always satisfied as `switchA` = TRUE (line 21), resulting in updating the values of `stateA` and `stateB`. It also checks whether `stateA` and `stateB` have reached the pre-defined values (line 24).

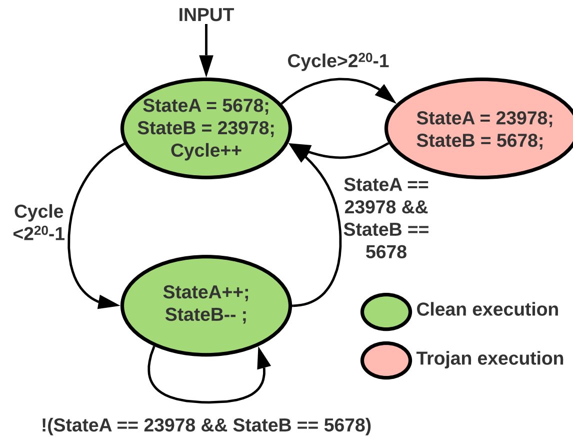

Fig. 2. State-transition diagram of motivating example [1]

If so, then `switchA` is set to FALSE. At the end, the controller accepts an input from the user for performing further action.

We insert a Trojan code at line 15 where we compare the current value of `cycle` with a very large number. The attacker intends to skip the functionality of *Branch 2*, and provide a false impression to the user that the controller is working by accepting inputs, however skipping *Branch 2* and not performing any operations. We explain it with the help of a state-transition diagram in Figure 2. Here, we observe that as soon as the `cycle` reaches the value of $2^{20} - 1$, `switchA` is maliciously set to FALSE and the values of `stateA` and `stateB` are swapped instead of the expected updates in *Branch 2*. Thus, the loop body repeatedly executes *Branch 1* and maliciously increments `cycle` after accepting input from the user. The Trojan resembles closely to the "ticking time-bomb" or "sequential Trojan" behaviour where the system malfunctions after running for sufficiently large number of cycles. From test and verification perspective, simulating the design for large number of cycles in an exhaustive manner is a hard problem.

3.2 Evaluating AFL-SHT on this example

Le *et al.*[19] envisioned hardware trojans in high-level SystemC designs as Synthesizable Hardware Trojan (SHT). They proposed fuzzing-based test generation AFL-SHT using AFL as the backend engine. Initially, the authors evaluated the trojan detection capability of vanilla AFL on *S3C* benchmarks and identified the pitfalls of AFL in detecting SHT. They presented three major modifications in the mutation block of fuzz-engine: 1) pump mutation 2) format-aware trimming, and 3) design-aware *interesting number* generation to tune test-generation for trojan detection.

We evaluated our motivating example with the authors' version of AFL-SHT and found that it was unable to generate appropriate values of `input`, `stateA` and `stateB` to activate the trojan behaviour. We observe that in-spite of aiding the fuzz-engine with coarse-grain information about the *interesting numbers*, it was unable to explore beyond line 7. This clearly shows the incapability of customized fuzz engine to generate `stateA`= 23978 and `stateB`= 5678 and satisfy the branch constraints. Similarly, we modified the trigger conditions of trojans in *S3C* benchmarks and found that AFL-SHT was unable to detect trojans while the performance were slightly better than vanilla AFL. This indicates AFL-SHT is unable to utilize the pump mutation and using interesting numbers from benchmarks in an effective manner. We

conclude our evaluation of AFL-SHT with a simple takeaway message: fuzzing needs additional aid to explore code segments guarded by complex conditional checks.

3.3 Evaluating SCT-HTD on this example

A growing body of work in high-level synthesized designs have led to renewed interest in concolic testing for SystemC [20–22]. Recent work SCT-HTD [20] has proposed a scalable, selective and systematic exploration approach for concolic testing of SystemC designs to uncover stealthy trojans. The authors identified a crucial insight that concolic engine do not distinguish in-built between library codes and design codes, and therefore gets stuck in exploration of undesirable library codes. The underlying assumption is: library codes and pragmas are maintained by SystemC specifications and therefore are trusted elements. Thus, exploring different paths in library codes does not hold much relevance and hence the authors have selectively restricted the state-space exploration within the design code. Additionally, while exploring the state-space, they prioritize states hitting uncovered conditions compared to states having already explored conditions. The authors have evaluated their approach on *S3C* benchmarks and showed improvement in terms of number of inputs generated for trojan detection.

A direct advantage of using concolic based approach for test-generation is systematic exploration of state-space. It maintains a history of states previously visited and prioritize the bandwidth towards exploring complex conditional checks. We run selective concolic engine (re-implementing the concepts of SCT-HTD) and run it on our motivating example. We discovered that SCT-HTD failed to trigger the Trojan condition particularly because of two reasons: 1) SCT-HTD forked two states for every iteration of the while loop and soon running into out of memory on a machine having 16 GB RAM, 2) SCT-HTD repeatedly invokes SAT-engine to generate test-input for satisfying the condition at line 11 when the condition at line 21 is True. On a high-level, it shows that concolic engines require an additional aid for intelligent exploration of search space leaving less memory footprint.

3.4 Lessons learnt

Post evaluation of state-of-art techniques of trojan detection, we conclude that the challenges lying ahead involves two important issues: 1) detecting trojans in a faster and scalable manner, and 2) detecting extremely "hard-to-trigger" trojan logic. As a defender, one can never make a prior assumption about possible location of trojans and tune the Testing methodology in a particular way. For a defender, the confidence of a design being “Trojan-free" can only come when a test-set is generated providing a sufficient coverage on the design. Our evaluation with fuzzing and concolic testing shows that they can be combined in a synergistic way to accelerate trojan detection. This can avoid the path explosion problem of symbolic execution by controlled forking in symbolic loops using the fuzzer-generated test cases, thereby reaching deeper code segments without generating lots of states.

Combining the two mainstream techniques for security testing, FuCE can achieve high code coverage with faster Trojan detection ability. FuCE avoids getting stuck either in Fuzz testing or in symbolic execution. FuCE can avoid the path explosion problem of symbolic execution to a certain extend, by controlled forking in symbolic loops using the fuzzer-generated test cases. So it can reach deeper code areas without generating lots of states.

4 FUCE FRAMEWORK

We present the workflow of FuCE in Figure 3. The FuCE framework consists of three components: 1) Greybox fuzzing, 2) Concolic execution, and 3) Trojan detector.

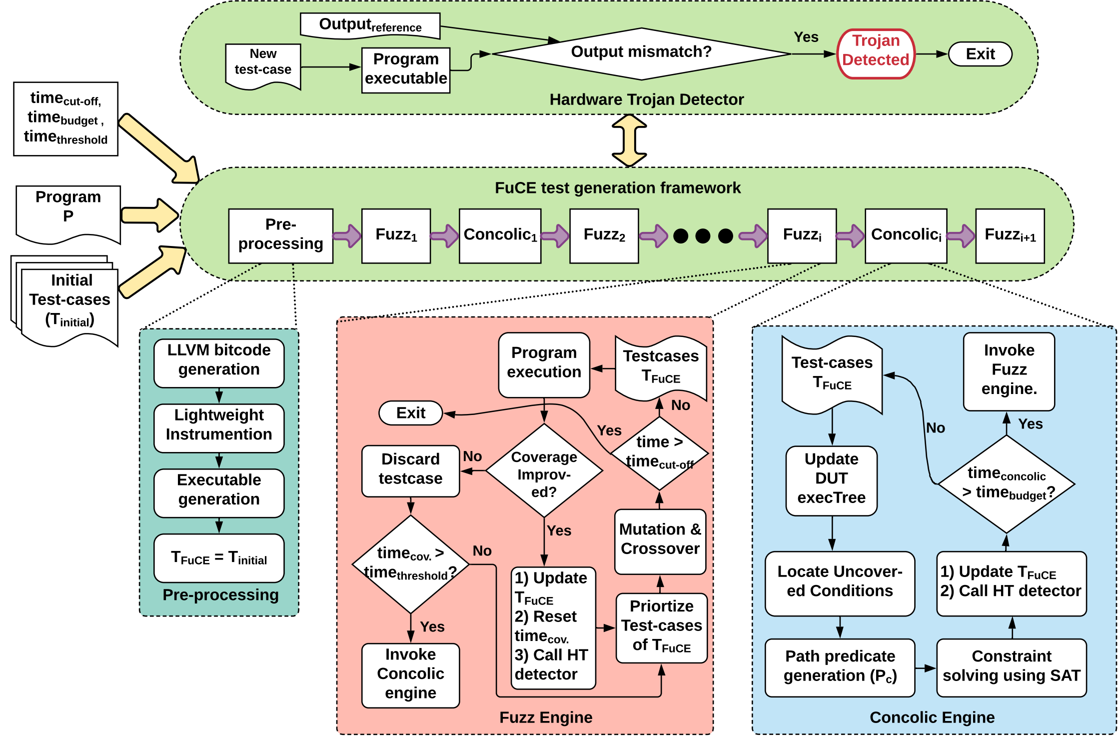

Fig. 3. FuCE test generation framework. *Fuzz engine* is fed with initial test-cases. As coverage improvement ceases in fuzz engine, *Concolic engine* starts execution with fuzz generated test cases. *Fuzz engine* and *Concolic engine* execute sequentially to penetrate deep into hard-to-satisfy conditional checks of programs. The *Trojan Detector* checks for trojan whenever *FuCE* generates a new test-case.

4.1 Greybox Fuzzing by AFL

High-level synthesized designs predominantly written in SystemC/C++ are initially passed through static analysis tool `LLVM` [23] backend to generate intermediate representation (IR). The `LLVM` generated IRs are fed to `afl-clang-fast`, which is based on `clang` [11], a front end compiler for programming languages like C, C++, SystemC, among others.

`afl-clang-fast` performs code instrumentation by automated injection of control flow statements on every conditional statement in run time, and generates executable. The core insight is: Trojan logic must be embedded under one of the conditional statements, so covering all conditional statements while verification and testing provides sufficient confidence of triggering the Trojan logic. The instrumented executable is then fed to our greybox fuzz-engine AFL along with an initial test-set ($T_{initial}$) for fuzz testing.

In algorithm 1, we outline the overall flow of Greybox fuzzing. At first, we provide the high-level DUT and a user-provided test-set $T_{initial}$ to the fuzzing framework. The *CALCULATE-ENERGY* function assigns energy to the initial seed $T_{initial}$ on the basis of external features of the test case like the execution time, bitmap coverage, depth of the test case in terms of fuzzing hierarchy. A test case that is fast, covers more branches and has more depth, is given more energy. AFL then decides the number of random fuzzing iterations for that test case. AFL uses $T_{initial}$

Algorithm 1: FUZZER(DUT, $T_{initial}$)

Data: Design Under Test DUT, User provided test-inputs $T_{initial}$, User defined time bound $time_{cut-off}$
Result: T_{fuzzed} ▹ Interesting test-inputs queue

```
T_fuzzed ← T_initial                          ▹ Initialization of the AFL's test-inputs queue
while time ≤ time_cut-off do
    for τ ∈ T_fuzzed do
                  ▹ Mutate τ to generate test-cases based on the energy parameter
        K ← CALCULATE_ENERGY(τ)
        for i ∈ {1, 2, ..., K} do
            τ′ ← MUTATE-SEED(τ)               ▹ τ′ denotes the mutated test case
            if IS-INTERESTING(DUT, τ′) then
                T_fuzzed ← T_fuzzed ∪ τ′      ▹ τ′ is interesting if it improves branch coverage
            end
        end
    end
end
return T_fuzzed
```

to perform operations like *deterministic mutations* and *havoc mutations* to generate newer test-cases with the help of the *MUTATE-SEED* function. The deterministic mutation stage scans each byte of test-case and mutates them to generate new test-case. This includes bit flipping, byte flipping, arithmetic increments and decrements and magic value substitution. The number of children test-cases generated loosely depends on the size of the original test-case. However, havoc mutation performs aggressive mutations like mutating bit/bytes values at random location with a random value, deleting or cloning sub-sequence for generating new test-cases. AFL uses branch-pair as a fitness metric to determine the quality of test-input. For each branch-pair, AFL maintains a hash-table entry the number of times it is hit. The *IS-INTERESTING* function checks whether the mutated test case is interesting or not. AFL considers a test-input to be *interesting*, if it covers a new-branch pair not hit so far, or has hit a branch-pair unique number of times compared to past observations. Interesting test-inputs are retained to form the next candidates for fuzzing. The algorithm terminates when either no more interesting test cases can be found or the user-defined $time_{cut-off}$ expires. AFL maintains all interesting test-inputs in the queue T_{fuzzed}.

4.2 Concolic Execution by S2E

In our work, we use S2E as our concolic execution engine for test-generation. S2E has two main components: 1) a concolic virtual machine based on QEMU [6] and 2) a symbolic execution engine based on KLEE to switch back and forth between concrete execution and symbolic execution. We provide it the high level design DUT and a set of test-cases $T_{initial}$. The *CONC-EXEC* execute the DUT with all test-cases $T_{initial}$ generating concrete execution traces. S2E maintains an execution tree $DUT_{execTree}$ and identifies all the true and/or false edges of conditional nodes which are not covered by $T_{initial}$. S2E then assigns symbolic values to those predicates. The *COND-PREDICATE* constructs the path constraints for the uncovered edge of a condition, forks a new thread and invokes SAT-solver (*CONSTRAINT-SOLVER*) to generate the test-case. S2E selects the path for exploration in depth-first search order, based on its coverage analyzer and heuristically selects the path that maximizes the coverage. So, the final test-cases reported by S2E ideally should cover all conditions of $DUT_{execTree}$. We outline this approach in algorithm 2.

Algorithm 2: CONCOL-EXEC(DUT,$T_{initial}$)

```
Data: Design Under Test DUT, User provided test-inputs T_initial
Result: T_concolic                 ▹ Set of test-cases generated by the concolic engine
DUT_execTree ← ϕ                                               ▹ Execution tree for DUT
for τ ∈ T_initial do
    ▹ Update DUT's execution tree with path traces obtained from concrete execution of initial test inputs
    P_trace ← CONC-EXEC(DUT,τ)
    DUT_execTree ← DUT_execTree ∪ P_trace
end
for uncovered cond c ∈ DUT_execTree do
    ▹ Perform symbolic execution steps targeting uncovered conditional statements
    p_c ← COND-PREDICATE(c)
    t_new ← CONSTRAINT-SOLVER(p_c)   ▹ t_new is newly generated test-case by the concolic engine
    T_concolic ← T_concolic ∪ t_new
end
return T_concolic
```

4.3 Fusing fuzzer with concolic execution (FuCE)

We leverage the power of concolic execution to alleviate the drawbacks of greybox fuzzing without hitting a scalability bottleneck. As shown in Figure 3, we first perform lightweight instrumentation on all conditional statements of DUT and generate an instrumented executable. We start our fuzz-engine (*FUZZER*) with a set of initial test-cases $T_{initial}$. The fuzz-engine generates interesting test-cases using genetic algorithm and explores various paths in the design. When there is no coverage improvement and a user-defined time period $time_{threshold}$ is over, the concolic engine (*CONCOL-EXEC*) is invoked for unseen path exploration. The concrete execution function of the concolic engine generates the $DUT_{execTree}$ for the DUT by feeding the fuzzer generated test cases. The *CONCOL-EXEC* identifies uncovered conditions in $DUT_{execTree}$ and forks new threads for symbolic execution on such conditions using depth-first search strategy. We use fuzzed test-cases for concolic engine to generate new test-cases satisfying complex conditional statements. In order to avoid scalability bottleneck, we limit the runtime of concolic engine to $time_{budget}$. Concolic execution generated test-cases are then fed back to the fuzzer, thereby allowing scalable exploration of deeper program segments. This process continues till trojan is detected, which is the main objective of FuCE. Whenever a new test case is generated with either the fuzz engine or the concolic engine, it is added to T_{FuCE}, the test case queue for FuCE, and the trojan detector is invoked with the latest test case added to T_{FuCE}. If trojan is detected successfully, FuCE displays the result for trojan detection and stops. We formally present our test-generation approach in algorithm 3.

There may be designs where trojans get detected before 100% branch coverage is obtained. For related goals of either generating test-cases for achieving 100% branch coverage or trojan dertection in the absence of a golden model, we can run FuCE for a pre-defined $time_{cut-off}$ using a similar flow of switching between the fuzzer and the concolic engine as in Figure 3. Instead of checking for trojan detection, we check for complete branch coverage for the design and stop FuCE on attaining 100% branch coverage or on time out.

4.4 Hardware Trojan Detection

Using FuCE test generation framework, we localize trojans in a given DUT (algorithm 4). For every test-case FuCE generated either by *FUZZER* or *CONCOL-EXEC*, we run our design, collect the output response and compare it with reference response. The *SIMULATOR* invokes the SystemC library that provides predefined structures and simulation

Algorithm 3: FuCE (DUT, $T_{initial}$, $time_{cut-off}$, $time_{threshold}$, $time_{budget}$)

```
Data: Design Under Test (DUT), Initial test-inputs (T_initial), Time limit (time_cut-off), Threshold time limit for
      coverage improvement by Fuzzer (time_threshold), Time budget allocated for concolic execution
      (time_budget)
Result: T_FuCE                                                  ▷ Final test-cases generated by FuCE
                    ▷ InvokeConcolic performs concolic execution with fuzzing generated test-inputs
Function InvokeConcolic(DUT_execTree,i):
    time_concolic ← 0                                           ▷ Monitor concolic execution runtime
    for uncovered cond c ∈ DUT_execTree do
        p_c ← COND-PREDICATE(c)
        t_new ← CONSTRAINT-SOLVER(p_c)                          ▷ t_new is a newly generated test case
        T_FuCE ← T_FuCE ∪ t_new
        Invoke Trojan detector with t_new (algorithm 4)
        if time_concolic > time_budget then
            break
        end
    end
i ← 1                                                           ▷ Phase ID of sequential execution
T_FuCE ← T_initial                                              ▷ FuCE gets initial test-cases T_initial
time_coverage ← 0               ▷ time_coverage monitors time elapsed since last test-case retained
                                                                ▷ time denotes wall time of FuCE
while time ≤ time_cut-off do
    for τ ∈ T_FuCE do
        K ← CALCULATE_ENERGY(τ)      ▷ Mutate τ to generate test-cases based on the energy parameter
        for j ∈ {1, 2, . . . , K} do
            τ′ ← MUTATE-SEED(τ)                                 ▷ τ′ denotes the mutated test case
            if IS-INTERESTING(DUT, τ′) then
                T_FuCE ← T_FuCE ∪ τ′
                Invoke Trojan detector with τ′ (algorithm 4)
                Reset time_coverage
            end
            if time_coverage > time_threshold then
                InvokeConcolic(DUT_execTree, i)       ▷ Invokes the concolic engine when the fuzzer gets stuck
                 (DUT_execTree is the program execution tree of DUT)
                if time > time_cut-off then
                    return T_FuCE                   ▷ User defined total time budget for FuCE is exhausted
                end
                else
                    i = i + 1
                    break
                end
            end
        end
    end
end
coverage_FuCE = reportCoverage(T_FuCE)                          ▷ Reports code coverage
return (T_FuCE, coverage_FuCE)
```

Algorithm 4: HT-DETECTOR (DUT, Out_{golden}, t)

```
Data: DUT is Device Under Test, Out_golden is golden model output), t is the new test-case of FuCE
out_ref ← Out_golden(t)
out_DUT ← SIMULATOR(DUT, t)
if out_DUT ≠ out_ref then
    Trojan detected
    break
end
coverage_FuCE = reportCoverage(T_FuCE)                    ▷ Reports code coverage
return (T_FuCE, coverage_FuCE)
```

kernel for simulation of SystemC designs. The assumption is: any deviation of DUT's output (eg., bit corruption at certain locations or report of internal state information) from expected response is considered to be suspicious in nature and triggered from possible trojan behaviour. FuCE terminates as soon as a trojan behaviour is detected, otherwise at a user-defined time-out $time_{cut-off}$ and reports coverage metrics using the test-cases generated.

4.5 Evaluating FuCE on our motivating example

We evaluated FuCE on our motivating example to check the efficacy of our proposed framework. Our fuzzer quickly generated a considerable number of test-cases for `stateA` and `stateB` but was unable to explore beyond line 7. The test-cases generated by fuzzer were passed on to concolic engine. Concolic engine generated values of `stateA` and `stateB` satisfying the condition at line 7. These newly generated test-cases were fed back to the fuzzer leading to faster exploration of the entire loop body. We observed that the fuzzer preserved the test-cases which covered the loop body a unique number of times. Finally, FuCE reached the Trojan location within 560s±15%, whereas neither AFL-SHT nor SCT-HTD could detect it within two hours of run.

5 EXPERIMENTAL SETUP AND DESIGN

5.1 Experimental setup

We implement FuCE using state-of-art software testing tools: AFL (v2.52b) [39] for greybox fuzzing and S2E [9] to perform concolic execution. For robust coverage measurements, we cross-validated our results using a combination of coverage measuring tools: namely *afl-cov-0.6.1* [1], *lcov-1.13* [18], and *gcov-7.5.0* [14]. Experiments are performed on 64-bit linux machine having $i5$ processor and 16 GB RAM clocked at 3.20 GHz.

5.2 Benchmark characteristics

We evaluated FuCE on the SystemC benchmark suite, *S3CBench* [35] having SystemC Trojan-infected designs. The Trojans have a wide spectrum of purpose ranging across: Denial-of-service, Information leakage, and corrupted functionality. The types of trojan based on their triggered mechanism are categorized in Table 2. For the trojan type where the payload has memory, the trojan remains active for a prolonged period of time even when the trigger condition is not active anymore. We define the severity level based on this characteristic. The *S3CBench* is synthesizable to RTL using any commercial High Level Synthesis (HLS) tool. Table 3 shows the characterization of the *S3C* benchmark both for the original circuit, and the trojan induced circuit.

The benchmarks considered have diverse characterization: 1) Image and signal processing: ADPCM, 2) Cryptography: AES, 3) Data manipulation: Bubble sort, 4) Filters: Decimation, and 5) IP protocols: UART. The trojans in S3CBench are

Table 2. Trojan types — Combinational: Comb., Sequential: Seq.

Trojan	Trigger	Payload	Severity
CWOM	Comb.	Comb.	Low
CWM	Comb.	Seq.	High
SWOM	Seq.	Comb.	Low
SWM	Seq.	Seq.	High

hard to detect with random seeds [34]. The benchmark suite provides certain test-cases having high statement coverage but does not trigger the trojan behaviour. Two benchmarks in S3C suite, namely *sobel* and *disparity* that accept image as file input, were not considered because the concolic engine was unable to generate test-cases with these input formats. However, we present our results on the largest benchmark *AES-cwom*, and on the most complex benchmark of the *S3C* suite, i.e.,*interpolation-cwom*, which indeed demonstrates the efficacy of our approach.

5.3 Design of experiments

We perform two variants of experiments to evaluate the efficacy of *FuCE*: 1) Trojan detection 2) Achievable branch coverage. We compare the results with standardized baseline techniques namely fuzz-testing based approach(AFL) and symbolic model checking(S2E). We now describe the experimental setup of each baseline:

Baseline 1 (Fuzz testing): We run AFL on S3C benchmarks using default algorithmic setting. Initial seed inputs are randomly generated.

Baseline 2 (Symbolic execution): Like baseline 1, we run S2E on S3C benchmarks having default configurations. Randomly generated seed inputs are provided to S2E as inputs.

FuCE: We run FuCE on S3C benchmarks using randomly generated input testcases. We set $time_{threshold}$ = 5s and $time_{budget}$ = 1800s for our experiments as per FuCE (algorithm 3). These are user defined configurable parameters.

We evaluate FuCE with our baseline test generation techniques on two dimension: 1) Trojan detection capability 2) Branch coverage achievable during a pre-defined time limit. The first objective assumes availability of input-output response pairs from golden model to check Trojans functionally corrupted the design. However, the second objective aims to study the efficacy of test generation framework to achieve complete branch coverage on the design. Test-cases covering all conditional statements in the design enhance defenders' confidence of capturing any anomalous behaviour in the absence of golden model. For both experimental settings, we present case-studies demonstrating the effectiveness of FuCE on S3C benchmarks. For apples-to-apples comparison with prior state-of-art approaches [19, 20], we compare the results of Trojan detection as reported in their published works.

6 EMPIRICAL ANALYSIS

6.1 Trojan detection

We first analyze trojan detection capability of FuCE on S3C benchmarks. Since FuCE invokes fuzzing and concolic engine interchangeably, we term each fuzzing phase as $fuzz_n$ and concolic execution phase as $conc_n$; where n denotes phase ID in FuCE execution. For example, a trojan detected in phase $fuzz_3$ implies that the framework has gone through test generation phases $fuzz_1$-$conc_1$-$fuzz_2$-$conc_2$-$fuzz_3$ before detecting the Trojan.

We evaluated Trojan detection capability of FuCE on S3C benchmarks since it is the state-of-art trojan infected high-level designs. We reported the results in Table 4 and 5. Table 4 shows testcase generated and runtime of each

Table 3. HLS synthesized hardware characterization of S3C benchmarks. Trojan types: CWOM, SWM and SWOM

Benchmark	Type	SystemC characterization			HLS synthesized hardware			
		Branches	Lines	Functions	LUTs	Registers	Nets	Critical path(ns)
ADPCM	orig	26	186	6	121	87	346	3.94
	SWM	28	186	6	120	118	394	3.801
	SWOM	30	187	6	163	240	588	3.019
AES	orig	50	371	13	2782	4684	8809	7.589
	CWOM	68	380	13	2886	4772	9039	7.668
Bubble_sort	orig	20	78	3	472	551	1219	8.944
	CWOM	22	78	3	494	551	1219	7.527
	SWM	22	78	3	546	584	1400	7.87
Filter_FIR	orig	14	75	2	68	36	146	5.729
	CWOM	16	75	4	89	59	213	7.46
Interpolation	orig	10	108	3	984	654	212	7.45
	CWOM	30	108	3	1071	595	212	8.331
	SWM	30	108	3	612	570	212	8.321
	SWOM	30	109	3	612	569	212	8.321
Decimation	orig	88	304	3	3018	1696	634	8.702
	SWM	94	304	3	3108	1741	634	8.702
Kasumi	orig	36	288	12	1378	956	188	8.016
	SWM	38	288	12	1385	958	272	8.016
	CWOM	38	288	12	1431	987	273	9.266
UART	orig	28	160	3	510	142	1336	3.137
	SWM-1	48	164	3	549	196	1336	2.766
	SWM-2	50	164	3	566	190	13.36	4.367

execution phase of FuCE. We compare branch coverage obtained by FuCE with AFL and S2E in Table 4. In Table 5, we report the total time taken, the number of testcases generated for Trojan detection and memory usage of FuCE and compare these with our baseline techniques (AFL and S2E) as well as with the state-of-art test-based Trojan detection approaches [19, 20]. We set a timeout of two hours for trojan detection using each technique.

1) Coverage obtained till Trojan detection: We report the branch coverage results of FuCE in Table 4. We present the number of test-cases and the time taken by FuCE in each execution phase before Trojan detection. One important point to note: FuCE could detect all the Trojans in S3C benchmarks within $conc_1$ execution phase. The designs for which AFL standalone can detect the trojan in $fuzz_1$ phase only, we do not invoke $conc_1$ phase. We present the branch coverage obtained by FuCE and baseline techniques until Trojan detection (or, reaching pre-defined timeout limit).

Comparison with baselines: We compare the branch coverage obtained by FuCE and baseline techniques in Table 4 and observe that FuCE outperforms baseline techniques in terms of Trojan detection capability and coverage achieved till Trojan detection. For *ADPCM*, FuCE could detect the trojan in $fuzz_1$ phase itself. It is observed that AFL and FuCE perform similarly in trojan detection for *ADPCM*. S2E on the other hand detects trojan with little better coverage but takes longer time. Timing comparison is shown in Table 5. For *AES*, FuCE goes through the phases $fuzz_1$ and $conc_1$ to

Table 4. Trojan Detection by *FuCE*. Trojan types: CWOM, SWM and SWOM. Detection: Yes (✓), No (✗)

Benchmarks	Testcases		Time(in s)		Branch cov. (%)		
	$fuzz_1$	$conc_1$	$fuzz_1$	$conc_1$	AFL	S2E	FuCE
ADPCM	3	-	38	-	88.1(✓)	88.9(✓)	88.1(✓)
	6	-	15	-	85.7(✓)	86.1(✓)	85.7(✓)
AES	4	1	43	4	93.8(✓)	81.5(✗)	94.9(✓)
Bubble_sort	4	8	19	192	95.5(✗)	95.5(✗)	100(✓)
	3	3	19	124	95.5(✗)	95.5(✗)	100(✓)
Filter_FIR	4	2	11	13	93.8(✓)	93.8(✓)	93.8(✓)
Interpolation	63	4	11	38	45.1(✗)	46(✗)	76.1(✓)
	3	-	4	-	57.5(✓)	56.1(✓)	57.5(✓)
	3	-	4	-	57.5(✓)	56.1(✓)	57.5(✓)
Decimation	4	-	24	-	66.7(✓)	66.7(✓)	69.1(✓)
Kasumi	23	-	5	-	87.5(✓)	84.3(✓)	87.5(✓)
	22	-	5	-	87.5(✓)	84.3(✓)	87.5(✓)
UART-1	6	3	34	234	85.7(✓)	81.2(✗)	88.5(✓)
UART-2	2	2	32	242	79.4(✓)	88.3(✓)	88.3(✓)

detect trojan with better coverage than AFL and S2E. AFL could not detect the trojan without violating the threshold time allocated for it to check for test inputs generated that hit new branch in the design. So, FuCE shifts from $fuzz_1$ phase to $conc_1$ phase thus detecting trojan in less time than AFL alone. For benchmarks *AES*, *Bubble_sort*, *Filter_FIR*, *Interpolation* and *UART*, FuCE invokes $conc_1$ indicating fuzz engine was stuck at $t_{threshold}$. Similarly, S2E could not detect all the Trojans in user-defined time limit indicating concolic execution is a slow process because of timing overhead from expensive SAT calls of concolic engine. An important advantage of using fuzzing alongside concolic execution is automatic identification of variable states which are responsible for complex checking operations. This effectively reduces the burden on concolic engine to categorise the variables between symbolic and concrete before invoking it. In the next subsection, we outline case-studies on S3C benchmarks describing how FuCE alleviates the challenges coming from standalone techniques to explore design state-space without hitting scalability bottleneck.

2) Analyzing timing improvement: In our work, we compare wall-time for trojan detection of FuCE with baseline and state-of-art techniques. The reason is that the fuzz engine takes less cpu-time for a given wall-time as it is an IO intensive process, whereas a concolic engine takes more cpu-time for a given wall-time as it forks multiple threads for test generation. For a fair comparison across a range of techniques, we choose wall-time as a metric to compare with previous works. The wall-time taken for trojan detection is presented in Table 5 (Column 3). *TO* indicates trojan is not detected within the wall-time limit of two hours.

Comparison with baselines: From Table 5, we conclude that FuCE takes less time than vanilla AFL for half of the benchmark designs considered and outperforms S2E on all the designs except on *Decimation*. FuCE avoids expensive path exploration by concolic execution using fuzzer generated seeds leading to faster and scalable Trojan detection.

Table 5. Comparing Trojan detection using *FuCE* with prior works [19, 20]. Trojan: CWOM, SWM and SWOM.

Benchmarks	Test-cases generated					Wall-time taken (s)					Memory footprint(MB)		
	AFL	AFL-SHT	S2E	SCT-HTD	FuCE	AFL	AFL-SHT	S2E	SCT-HTD	FuCE	S2E	SCT-HTD	FuCE
ADPCM	3	423	14	27	3	38	1.17	55	157	38	3029	3546	51.44
	6	414	23	7	6	15	1.67	49	31	15	3433	1341	51.53
AES	7	22	2	11	5	337	0.04	TO	23	47	9879	1386	1051
Bubble_sort	30	39	160	2	12	TO	4.82	TO	8	211	4761	1074	2698
	12	108	32	4	6	TO	337.36	TO	10	143	4759	1106	2618
Filter_FIR	11	41	5	26	6	1184	0.07	41	13	24	640	1071	480
Interpolation	63	2325402	72	-	67	TO	TO	TO	-	49	16244	-	3081
	3	47	8	-	3	4	0.16	130	-	4	8790	-	56.20
	3	47	2	-	3	4	0.16	14	-	4	3326	-	56.21
Decimation	4	-	2	-	4	22	-	12	-	24	723	-	55.57
Kasumi	23	316	76	-	23	5	1.32	245	-	5	2908	-	53.29
	22	414	49	-	22	5	1.32	83	-	5	2976	-	53.28
UART-1	7	51	15	3	9	311	0.18	TO	9	268	5929	1071	3164
UART-2	4	-	2	3	4	298	-	730	9	274	2651	1070	2618

Comparison with state-of-art approaches: We reached out to the authors of [19, 20] to obtain the test generation frameworks for independent evaluation and apples-to-apples comparison with FuCE. However, the actual implementations were unavailable. Thus, we compared the timing as reported in their papers and used similar computing platform for our experiments. We found FuCE took longer time than AFL-SHT except for two cases: *interpolation-cwom* and *bubble_sort-swm*; where FuCE detects the trojan in 49s and 243s respectively. In these two cases, AFL-SHT either took longer time or timed out. This exhibits fundamental limitation of fuzzing even though that fuzz engine is heavily customized based on benchmark characteristics. FuCE on the other hand can easily identify the state of fuzz engine getting stuck and invoke concolic engine to penetrate into deeper program state. In a subsequent section, we will elaborately describe Trojan behaviour in S3C benchmarks and FuCE's ability to detect these. FuCE detected Trojans quicker than SCT-HTD for *ADPCM*, *Interpolation*, *Decimation* and *Kasumi* because *SCT-HTD* took longer time to solve computationally intensive operations to generate the testcases.

3) Analyzing test-case quality: As listed in Table 5 (Column 2), FuCE leverages S2E with fuzz generated test-cases to accelerate the coverage over a defined period of time. The fuzzer generated input seeds guide the symbolic engine to construct the execution tree along the execution path triggered by existing test-cases and generate new test cases reaching unexplored conditional statements. For a fair comparison with FuCE, we report the number of test-cases preserved by each technique until it reaches user-defined timeout (or, till Trojan detection).

Comparison with baselines: Compared to AFL and S2E, the number of test-cases generated until Trojan detection are comparable for all benchmarks. A closer analysis reveals that the number of testcases generated by FuCE is same as AFL for the cases where AFL as standalone was sufficient in detecting the Trojans without getting stuck for $time_{threshold}$. But, for cases where AFL crossed the $time_{threshold}$ to generate a new testcase that improves coverage, FuCE invoked concolic engine for generating qualitative test-cases quickly. Finally, FuCE was able to detect Trojan using fewer

test-cases than both AFL and S2E. This indicates that FuCE uses the time-budget judiciously for creating effective test-cases that explore deeper program segments when fuzzing is stuck.

Comparison with state-of-art approaches: We compared the number of test-cases generated via state-of-art approaches with FuCE. FuCE outperforms SCT-HTD for designs like *ADPCM*, *AES* and *Filter-FIR* by generating fewer test-cases. Although SCT-HTD is better than FuCE in terms of generating effective test-cases for designs like *Bubble-sort* and *UART*, we will explore in subsequent sections that this heavily depends on the exploration strategy selected by concolic engine to explore the search space. Due to unavailability of branch coverage by SCT-HTD till Trojan detection, it is difficult to conclude whether SCT-HTD generated fewer test-cases with better coverage, or explored Trojan location quickly and terminated before achieving substantial coverage on the rest of the design.

4) Analyzing memory footprint: The last column of Table 5 reports memory usage denoting maximum memory footprint . We observe that trojans are detected with reasonable memory usage by FuCE. We compare it with concolic execution engine S2E and SCT-HTD reported in [20]. *AFL* is an input-output (IO) bound process whereas S2E is memory intensive program allocating huge memory for an application to run on a virtual machine. For AFL and AFL-SHT, we used default configuration of 50MB memory which was sufficient for all the S3C benchmarks. From the results, one can interpret that fuzzing combined with concolic execution has 50% less memory footprint(average) compared to standalone concolic execution.

6.2 Coverage Improvement

For analyzing the effectiveness of FuCE framework, we measure the branch coverage obtained by running the baseline techniques and FuCE with a time-limit of two hours. Figure 4 and Table 6 study the detailed coverage analysis over the entire time period for FuCE and baseline techniques.

From the coverage data (Figure 4), we can categorize S3C benchmarks into two types from testing perspective: simple and complex. For simple designs having small number of nested conditional statements, FuCE could achieve 100% coverage within a short span of time with fewer test-cases. These are: *Bubble-sort* and *Kasumi*. Complex benchmarks have deeper levels of nested loop and conditional statements along with ternary operations. These are: *ADPCM* and *Interpolation*. FuCE achieved 100% branch coverage on all S3C benchmarks except *UART*, *AES*, *Filter_FIR* and *Decimation*. We dig deeper into the coverage analysis for which FuCE could not achieve 100% coverage and discovered an interesting observation: presence of unreachable branch conditions in the original benchmark designs. We analytically found uncovered branch conditions from our experiments and verified that no test-case is possible for covering these code segments. We believe that these unreachable code segments will be optimized out during RTL level synthesis and therefore is not a drawback of FuCE test generation framework.

6.3 Case Studies

Here, we dive deep into FuCE's performance on four *S3CBench* designs across two orthogonal directions: 1) Trojan detection capability and 2) achievable branch coverage within a defined time limit.

1) Case Study I: Trojan Detection

Interpolation is a 4-stage interpolation filter. We consider *CWOM* trojan variant for our case study. Only FuCE could successfully detect the trojan amongst every other state-of-art technique (Table 5). We show *CWOM* trojan variant of

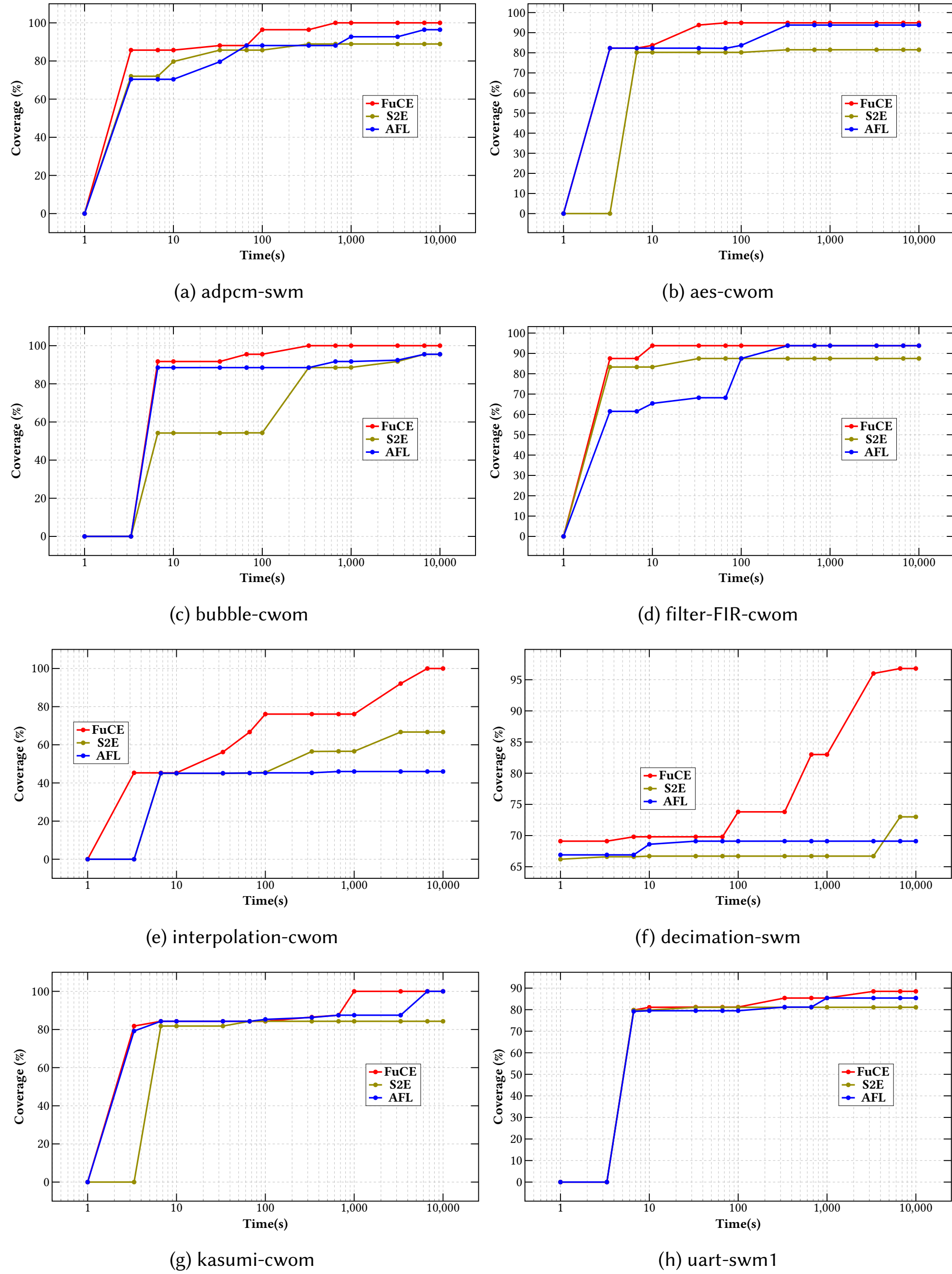

Fig. 4. Branch coverage obtained on S3C benchmarks after running *FuCE*, *S2E* and *AFL* for two hours

Table 6. Coverage improvement by *FuCE*. Trojan types: CWOM, SWM and SWOM.

Benchmarks	#Testcases generated			Time(in s)			Phases	Branch cov. (%)
	$fuzz_1$	$conc_1$	$fuzz_2$	$fuzz_1$	$conc_1$	$fuzz_2$		
ADPCM	3	39	3	43	1800	1940	$fuzz_1$-$conc_1$-$fuzz_2$	100
	6	82	4	20	1800	1640	$fuzz_1$-$conc_1$-$fuzz_2$	100
AES	4	1	-	43	4	-	$fuzz_1$-$conc_1$	94.9
Bubble_sort	4	8	-	19	192	-	$fuzz_1$-$conc_1$	100
	3	3	-	19	124	-	$fuzz_1$-$conc_1$	100
Filter_FIR	4	2	-	13	11	-	$fuzz_1$-$conc_1$	93.8
Interpolation	63	460	55	11	1800	3880	$fuzz_1$-$conc_1$-$fuzz_2$	100
	3	1427	28	9	270	2217	$fuzz_1$-$conc_1$-$fuzz_2$	100
	3	663	51	9	270	3640	$fuzz_1$-$conc_1$-$fuzz_2$	100
Decimation	4	12	5	29	126	1429	$fuzz_1$-$conc_1$-$fuzz_2$	96.8
Kasumi	23	22	-	10	1221	-	$fuzz_1$-$conc_1$	100
	22	12	-	10	1682	-	$fuzz_1$-$conc_1$	100
UART-1	6	3	-	34	234	-	$fuzz_1$-$conc_1$	88.5
UART-2	2	2	-	32	242	-	$fuzz_1$-$conc_1$	88.3

interpolation in Listing 2. The trojan triggers once the output of FIR filter's final stage ($SoP4$) matches with a specific "magic" value. Line 3 is Trigger logic and line 4 shows the payload circuit as output write operation. The trigger activates for a input resulting in the sum of product of the fourth filter($SoP4$) as $-.015985481441$. Trigger activation leads to execution of payload circuit (line 3) and writes the output $odata - write = -0.26345$. Inputs not satisfying the Trigger condition behave functionally equivalent to Trojan-free design.

Fuzzing techniques like AFL are unlikely to satisfy the conditional checks against specific values (line 3 of Listing 2) in a short time-span. AFL failed to detect the trojan in two hours time limit. S2E executing with random seeds also failed to generate inputs satisfying the trojan trigger condition. However, FuCE leverages the strength of both fuzzing and concolic execution. FuCE passes the interesting inputs as identified by the fuzzer in phase $fuzz_1$ to S2E in phase $conc_1$. S2E traces each input generated by the fuzzer discovering unexplored program states by AFL during phase $fuzz_1$. S2E generates inputs satisfying complex branch conditions to explore the undiscovered states. Thus, FuCE triggered the trojan payload using S2E and successfully detected the Trojan.

Listing 2. Interpolation - Trojan Logic

```
1 /*Trojan Triggers based on particular input combination */
2 #elif defined(CWOM) ||defined(CWOM_TRI)
3     if(SoP4 == -.015985481441)   //trojan trigger logic
4         odata_write = -0.26345;  //trojan payload circuit
5     else
6         //Trojan free execution for odata_write
7 #endif
```

Advanced Encryption Standard (AES) is a symmetric block cipher algorithm. The plain text is 128 bits long and key can be 128, 192 or 256 bits. *S3CBench* contain AES-128 bit design and the trojan type is *CWOM*. This trojan leaks the

secret key for a specific plain-text input corrupting the encryption generating incorrect cipher-text. AES-128 performs ten rounds of repetitive operation to generate cipher-text CT_{10}. The trojan implementation performs an additional 'n' rounds to generate cipher-text (CT_{10+n}). Initially, the attacker tampers the key K_n to be used in the round R_{10+n} for the round operations: *SubBytes, ShiftRows, MixColumns, AddRoundKey*. Later, key K_{10} can be recovered from the plain text P and cipher-text CT_{10+n}. Using FuCE, we compare the generated cipher-text with the expected cipher-text to detect the presence of a Trojan.

Listing 3. AES - Trojan Logic

```
1  //Trojan Triggers for a particular input present in the plaintext message
2  #ifdef CWOM
3      if (data3[0] == -1460255950) // trojan trigger logic
4          odata[(i * NB) + j] = (data1[i] $>>$ (j * 8));// trojan payload circuit
5      else  //Trojan free execution for output data
6          odata[(i * NB) + j] = 0;
7  #endif
```

We examined the performance of FuCE on *AES*. FuCE successfully detected the Trojan in *AES-cwom* in phase $fuzz_1$-$conc_1$ with a 6.5x timing speed-up compared to AFL. From Table 4, we observe that FuCE detected the trojan using the testcase generated by S2E during phase $conc_1$. We supplied four test-cases generated by AFL during $fuzz_1$ to S2E. However, S2E supplied with random test-cases was unable to detect the trojan (Table 5) and timed-out with a branch coverage of 81.5%. We indicate the Trojan infected *AES* in Listing 3 (lines 2 and 3). The Trojan trigger condition for *AES* is a rare combination of input values.

2) Case Study II: Coverage Improvement

Listing 4. ADPCM - Encode and Decode Logic

```
1  //Encode Logic
2  if (diff[15] == 1) {     //checking exactness of value
3      width.diff_data = ((diff ^ 0xffff) + 1);
4      neg_flag = true;
5  }
6  else {
7      width.diff_data = diff;
8      neg_flag = false;
9  }
10 //Decode Logic
11  if (width.enc_data > 7) {
12     width.enc_data = 7;
13     dec_tmp = width.enc_data * divider;
14     remainder1 = dec_tmp.range(1, 0);
15     if (remainder1 >= 2) {  //Nested branch conditions
16         width.pre_data += (dec_tmp >> 2) + 1;
17     }
18     else {
19         width.pre_data += (dec_tmp >> 2);
20     }
21 }
```

Adaptive Differential Pulse Code Modulation (ADPCM) converts analog information to binary data. *ADPCM* converts 16 bits Pulse Code Modulation (PCM) samples into 4-bit samples. The trojan considered is *SWM* type. The trojan gets triggered once the counter reaches a specific value corrupting the modulation process. Although FuCE detects the trojan successfully at phase $fuzz_1$ with a coverage of 88.1% but does not attain 100% coverage at this phase. From the *LCOV* report it was observed that FuCE was unable to reach certain portions of code with nested conditional branch statements as given in Listing 4. So FuCE goes to phase $conc_1$ for further exploration with the input seeds generated by $fuzz_1$. At $conc_1$, S2E traces the program following the same path taken by the fuzzer. When S2E arrives at the conditional check at line 2 (Listing 4) of the encode logic, it realizes that the path was not covered by the fuzzer. So, it produces input satisfying the condition which drives the execution to this new state transition. After the phase $conc_1$, the coverage analyzer for FuCE framework found that the coverage improved to 89.5% but the target coverage of 100% was yet to be reached. Since concolic execution is a slow process, S2E fails to generate test inputs within its time bound, that satisfy the nested conditional statements at line 11 of the decode logic(Listing 4). So FuCE goes to the next phase $fuzz_2$ to look for the undiscovered paths. AFL starts its execution with the input seeds generated by S2E at phase $conc_1$ which guides the fuzzer to quickly penetrate in the nested branch conditions and generate test cases that give 100% branch coverage for FuCE.

Thus *FuCE* could achieve 100% branch coverage following the phases $fuzz_1$-$conc_1$-$fuzz_2$ in the defined time budget of two hours. From the plot of *ADPCM-SWM* in Figure 4 (a), we can see that FuCE successfully achieves coverage of 100% in 3800 seconds. AFL on the other hand achieves coverage of 96.4% in 6115 seconds and could not improve further in the test time of two hours. S2E reaches a maximum coverage of 88.9% at 310 seconds while running for two hours. The breakdown of result for code coverage of *ADPCM-SWM* by *FuCE* could be found in Table 6.

Decimation Filter is a 5-stage filter with five FIR filters cascaded together. The type of trojan inserted here is *SWM*, which is triggered for a particular value of count in the design. The trojan is inserted here in the final stage, i.e., the fifth stage. Results on evaluation of this design of *S3CBench* have not been reported by any of the state-of-art prior works AFL-SHT or SCT-HTD. We have evaluated this benchmark successfully with FuCE, AFL and S2E. In our experiments, all the techniques could successfully detect the trojan in the circuit but FuCE outperformed others significantly with respect to achievable branch coverage. From the plot of *decimation-SWM*, in Figure 4 (f), it is observed that the seeds generated by FuCE could attain a coverage 96.8% running for 3289 seconds. Whereas for AFL the maximum coverage of 69.1% is attained with seeds generated in the time interval of (30 - 60) seconds and S2E reached the coverage of 73% in the time interval (100 - 300) seconds beyond which coverage did not increase even after running for two hours of time limit. FuCE could cover almost all the portions of code using its interlaced execution of phases $fuzz_1$-$conc_1$-$fuzz_2$ that AFL and S2E failed to cover individually.

7 CONCLUSION

In our work here, we have identified existing challenges of test-based hardware trojan detection techniques on high-level synthesized design. To this end, we proposed an end-to-end test-generation framework penetrating into deeper program segments at the HLS level. Our results show faster detection of trojans with better coverage scores than earlier methods. The complete framework for our proposed *FuCE* test-generation framework has the potential to reinforce automated detection of security vulnerabilities present in HLS designs[26]. We are investigating the ways in which trojans in RTL/gate level netlist get manifested in HLS using tools like VeriIntel2C, Verilator etc. Future works include exploration

of input grammar aware fuzzing and more focus on coverage metrics such as Modified Condition/Decision Coverage (MC/DC) [38] and path coverage.

REFERENCES

[1] afl-cov 2013. Coverage Measuring Tool. https://github.com/mrash/afl-cov.

[2] Alif Ahmed, Farimah Farahmandi, Yousef Iskander, and Prabhat Mishra. 2018. Scalable hardware Trojan activation by interleaving concrete simulation and symbolic execution. In *International Test Conference*. 1–10.

[3] A. Ahmed and P. Mishra. 2017. QUEBS: Qualifying event based search in concolic testing for validation of RTL models. In *International Conference on Computer Design*. 185–192.

[4] Mainak Banga and Michael S Hsiao. 2011. Odette: A non-scan design-for-test methodology for trojan detection in ics. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 18–23.

[5] Animesh Basak Chowdhury, Ansuman Banerjee, and Bhargab B. Bhattacharya. 2018. ATPG Binning and SAT-Based Approach to Hardware Trojan Detection for Safety-Critical Systems. In *Network and System Security*. Springer International Publishing, Cham, 391–410.

[6] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

[7] Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. 2009. MERO: A statistical approach for hardware Trojan detection. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 396–410.

[8] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems*. 1–5.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–49.

[10] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. 2019. VeriFuzz: Program aware fuzzing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 244–249.

[11] clang 2007. clang Compiler. https://clang.llvm.org/.

[12] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition*, Vol. 2. 119–129.

[13] Jonathan Cruz, Farimah Farahmandi, Alif Ahmed, and Prabhat Mishra. 2018. Hardware Trojan detection using ATPG and model checking. In *International Conference on VLSI Design*. 91–96.

[14] GCov 2012. GCov Coverage Measurement Tool. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[15] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. 2016. MERS: statistical test generation for side-channel analysis based Trojan detection. In *International Conference on Computer and Communications Security*. 130–141.

[16] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *International Conference on Computer-Aided Design*. 1–8.

[17] David Larochelle and David Evans. 2001. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*.

[18] LCOV 2012. LCOV Coverage Report. http://ltp.sourceforge.net/coverage/lcov.php.

[19] Hoang M Le, Daniel Große, Niklas Bruns, and Rolf Drechsler. 2019. Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing. In *Design, Automation & Test in Europe Conference & Exhibition*. 602–605.

[20] Bin Lin, Jinchao Chen, and Fei Xie. 2020. Selective concolic testing for hardware trojan detection in behavioral SystemC designs. In *Design, Automation & Test in Europe Conference & Exhibition*. 19–24.

[21] Bin Lin, Kai Cong, Zhenkun Yang, Zhigang Liao, Tao Zhan, Christopher Havlicek, and Fei Xie. 2018. Concolic testing of SystemC designs. In *International Symposium on Quality Electronic Design*. 1–7.

[22] Bin Lin, Zhenkun Yang, Kai Cong, and Fei Xie. 2016. Generating high coverage tests for SystemC designs using symbolic execution. In *Asia and South Pacific Design Automation Conference*. 166–171.

[23] LLVM 2003. LLVM compiler. https://llvm.org/.

[24] Y. Lyu, A. Ahmed, and P. Mishra. 2019. Automated activation of multiple targets in RTL models using concolic testing. In *Design, Automation & Test in Europe Conference & Exhibition*. 354–359.

[25] Yangdi Lyu and Prabhat Mishra. 2021. MaxSense: Side-Channel Sensitivity Maximization for Trojan Detection using Statistical Test Patterns. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26, 3 (2021), 1–21.

[26] M. Rafid Muttaki, Nitin Pundir, Mark Tehranipoor, and Farimah Farahmandi. 2021. *Security Assessment of High-Level Synthesis*. Springer International Publishing, Cham, 147–170.

[27] Zhixin Pan and Prabhat Mishra. 2021. Automated test generation for hardware trojan detection using reinforcement learning. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 408–413.

[28] Sayandeep Saha, Rajat Subhra Chakraborty, Srinivasa Shashank Nuthakki, Debdeep Mukhopadhyay, et al. 2015. Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In *International Workshop on Cryptographic Hardware and Embedded*

Systems. Springer, 577–596.

[29] Koushik Sen. 2007. Concolic testing. In *International Conference on Automated Software Engineering*. 571–572.

[30] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. 2017. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security* 1, 1 (2017), 85–102.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Network and Distributed Systems Security*. 1–16.

[32] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308* (2021).

[33] Trusthub 2015. Trusthub Benchmark. https://trust-hub.org.

[34] N. Veeranna and B. C. Schafer. 2017. Hardware Trojan detection in behavioral intellectual properties (IP's) using property checking techniques. *IEEE Transactions on Emerging Topics in Computing* 5, 4 (2017), 576–585.

[35] N. Veeranna and B. C. Schafer. 2017. S3CBench: Synthesizable security SystemC benchmarks for high-level synthesis. *Journal of Hardware and Systems Security* 1, 2 (2017), 103–113.

[36] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *International Conference on Software Engineering*. 61–64.

[37] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. 2016. Hardware Trojans: Lessons Learned after One Decade of Research. *ACM Transaction on Design Automation and Electronic Systems* 22, 1, Article 6 (2016).

[38] Yuen Tak Yu and Man Fai Lau. 2006. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software* 79, 5 (2006), 577–590.

[39] Michal Zalewski. [n.d.]. American fuzzy lop (2017). ([n. d.]). http://lcamtuf.coredump.cx/afl

[40] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. 2018. Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs. In *International Symposium on Field-Programmable Gate Arrays*. 269–278.