HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications

Qian Zhang
University of California, Los Angeles
USA
zhangqian@cs.ucla.edu

Jiyuan Wang
University of California, Los Angeles
USA
wangjiyuan@g.ucla.edu

Miryung Kim
University of California, Los Angeles
USA
miryung@cs.ucla.edu

ABSTRACT

As specialized hardware accelerators like FPGAs become a prominent part of the current computing landscape, software applications are increasingly constructed to leverage heterogeneous architectures. Such a trend is already happening in the domain of machine learning and Internet-of-Things (IoT) systems built on edge devices. Yet, debugging and testing methods for heterogeneous applications are currently lacking. These applications may look similar to regular C/C++ code but include hardware synthesis details in terms of preprocessor directives. Therefore, their behavior under heterogeneous architectures may diverge significantly from CPU due to hardware synthesis details. Further, the compilation and hardware simulation cycle takes an enormous amount of time, prohibiting frequent invocations required for fuzz testing.

We propose a novel fuzz testing technique, called HeteroFuzz, designed to specifically target heterogeneous applications and to detect platform-dependent divergence. The key essence of HeteroFuzz is that it uses a three-pronged approach to reduce the long latency of repetitively invoking a hardware simulator on a heterogeneous application. First, in addition to monitoring code coverage as a fuzzing guidance mechanism, we analyze synthesis pragmas in kernel code and monitor accelerator-relevant value spectra. Second, we design dynamic probabilistic mutations to increase the chance of hitting divergent behavior under different platforms. Third, we memorize the boundaries of seen kernel inputs and skip HLS simulator invocation if it can expose only redundant divergent behavior. We evaluate HeteroFuzz on seven real-world heterogeneous applications with FPGA kernels. HeteroFuzz is 754X faster in exposing the same set of distinct divergence symptoms than naive fuzzing. Probabilistic mutations contribute to 17.5X speed up than the one without. Selective invocation of HLS simulation contributes to 8.8X speed up than the one without.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → **Heterogeneous**.

KEYWORDS

Fuzz testing, heterogeneous applications, platform-dependent divergence

ACM Reference Format:
Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468610

1 INTRODUCTION

There is a growing interest in computer architectures to incorporate heterogeneity and specialization to improve performance [11, 13, 14, 17]. FPGA is re-programmable hardware that often exceeds the performance of general-purpose CPUs by several orders of magnitude [6, 22, 46] and offers lower cost across a wide variety of domains [5, 10, 15]. To support heterogeneous computing, hardware vendors provide CPU+FPGA multi-chip packages such as Intel Xeon [23, 47], and cloud providers support virtual machines with FPGA development frameworks such as Amazon F1 [1]. In the context of this paper, we use a term, *heterogeneous applications* to refer to software that consists of both *host* code and *kernel* code and can offload its computation-intensive *kernel* from CPU to FPGA under heterogeneous architectures.

Platform Dependent Divergence. Although FPGAs are becoming commercially available to a broad user base, they are associated with a high development cost [50]. There has been work on high-level synthesis (HLS) [16], which takes C/C++ kernel code as input and automatically synthesizes a corresponding FPGA accelerator. With HLS, programmers can implement their heterogeneous applications in C/C++; however, such C/C++ programs can produce different results on heterogeneous architectures compared to CPU due to various platform-dependent characteristics—bitwidth, available resources, memory access, recursion depth, dataflow mode, etc. For example, because an optimized FPGA kernel uses a custom bitwidth for integers and floating points, it could lead to overflows on FPGA. Thus, detecting platform-dependent divergent behavior is important because even when the application runs correctly on CPU, it could still crash or produce a wrong output silently when the kernel is executed on the FPGA accelerator [40, 41]. In fact, our investigation of Xilinx's forum—a popular online Q/A forum for FPGA HLS development—shows numerous examples of such platform-dependent divergence. Table 1 illustrates several such examples [65]. Programmers ask, *"why is there a difference between C-code, RTL-simulation, and the hardware test runs?"* [58]. In this

Table 1: Examples of Behavior Divergence Between CPU and FPGA

ID	Description	Time
894069 [57]	Segmentation fault when allocating a big array `int x[1920][1080]` on FPGA yet no error on CPU	8.5h
595225 [58]	Different outcome caused by HLS `dataflow` directive	47.7h
438446 [59]	Different outcome caused by FPGA fetching incorrect `struct vector training_set[MAXSZ]`	10.6h
754676 [60]	Different outcome caused by bitwidth `typedef ap_fixed<25,1,AP_RND> s25f24_type`	2.3h
785019 [61]	Getting all zeros when shifting an array caused by `#pragma HLS RESET`	3.1h
907213 [62]	Undecided output when overwriting a same variable within the loop yet no error on CPU	79.4h
1166264 [63]	`EMFILE` error when loading 2048 files with `#pragma HLS ARRAY_PARTITION` yet no error on CPU	11.7h
1126600 [64]	The 25-tap FIR filter bypassess some input multiplications with `#pragma HLS PIPELINE`	93.2h

post 894069 [57], due to limited stack size, running an image processing application with HLS simulation leads to a segmentation fault, but no such error happens on the CPU Linux platform. Though this programmer expressed a desire to test this *platform-dependent divergence*, no tool exists to meet such needs.

Current Practices of Testing Heterogeneous Applications. In practice, programmers often execute heterogeneous applications with a given input set on CPU and then compare against the results on heterogeneous architectures. However, *where do such inputs come from*? The most common sources include (1) test inputs handcrafted by an FPGA expert; (2) randomly generated tests by a data generator—widely used in hardware accelerator industry [25]; and (3) systematically enumerated inputs [29]. Unfortunately, these sources of inputs are unlikely to account for platform characteristics and thus are inefficient in revealing behavior divergence between FPGA and CPU. In recent years, fuzz testing has emerged as an effective test generation technique for large software systems [39]. Most fuzzing techniques, such as AFL [69], start from a seed input, generate new inputs by mutating the previous input, and add new inputs to the queue if they improve a given guidance metric such as branch coverage. They are also based on two inherent yet oversighted assumptions: (1) it takes a minuscule amount of time in the order of milliseconds to execute a target application, and (2) arbitrary mutations are likely to yield meaningful inputs.

Our experience suggests that neither of the two assumptions holds for heterogeneous applications. Compilation and hardware simulation takes several minutes (even hours), not milliseconds, and random mutations cannot account for hardware synthesis assumptions, crucial for detecting platform-dependent behavior. As shown in Table 1, we estimate an AFL-like technique that repetitively invokes an HLS simulator would require at least 11.7 hours to generate an input to detect the same error from the post 894069 [57].

Our three-pronged approach to reduce the long latency of naive fuzzing. HeteroFuzz targets testing of heterogeneous applications with the goal of generating inputs to demonstrate divergent behavior between CPU and FPGA. Our key insight is three-folds:

First, different from traditional fuzzing, whose guidance mechanism is driven by code coverage [69] or performance metrics [32] only, HeteroFuzz directly analyzes synthesis pragmas from kernel code and monitors *accelerator-relevant value spectra*: e.g., bitwidth, memory access, recursion depth, loop bound, and FIFO queue size. Any input that achieves either new code coverage or increases accelerator value spectra feedback are saved for further mutation. For example, HeteroFuzz can detect platform-dependent errors from the two posts [58, 59] in Table 1 within only six minutes by monitoring the fullness of a FIFO queue and the ranges of variable values.

Second, we design dynamic probabilistic mutations to increase the chance of exposing platform-dependent behavior. HeteroFuzz gradually increases the activation probability of the current mutation if a new accelerator value spectrum is achieved. For example, when a programmer uses `#typedef ap_uint<x> bitx` to declare a custom integer type with `x` bits, we instrument the kernel code to monitor how many bits have been actually used. If a particular mutation leads to a new bitwidth range, we label this mutation as a favored mutation and increase its activation probability.

Third, we reduce the long latency involved in the repetitive invocation of a hardware simulator. HeteroFuzz memorizes the boundary values of seen kernel inputs and selectively invokes a hardware simulator only if the current input goes beyond the previously seen range. This is based on the insight that accelerator synthesis is determined and optimized by kernel input values [30], and multiple invocations within the already seen range may only expose redundant divergence behavior of the same kind.

We evaluate HeteroFuzz's effectiveness on seven publicly available heterogeneous applications with FPGA kernels [3, 30, 71]. We compare HeteroFuzz against four alternatives: (1) HeteroFuzz without accelerator spectra monitoring, (2) HeteroFuzz without probabilistic mutations, (3) HeteroFuzz without selective invocation, and (4) naïve fuzzing, where we estimate the time based on the number of invocations to an HLS simulator. We measure speedup enabled by each of HeteroFuzz's three-pronged optimizations, while comparing the total number of errors (i.e., # of unique divergence symptoms) found within the same amount of time. In summary, this work makes the following contributions:

(1) To our knowledge, HeteroFuzz is the first fuzz testing technique to target heterogeneous applications and to detect platform-dependent differential behavior.
(2) To reduce the long latency of simulating heterogeneous applications, we designed a three-pronged approach that incorporates multi-dimensional accelerator feedback, dynamic probabilistic mutations, and selective invocations. HeteroFuzz achieves 754X speedup when finding the same number of distinct divergence symptoms.
(3) Each of the three-pronged optimizations is necessary to achieve significant speed-up without sacrificing fault detection potential: 7.78X more divergence-inducing inputs with accelerator-spectra monitoring, 17.5X speed-up by dynamic probabilistic mutations, and 8.8X speed-up by selective invocation, compared to HeteroFuzz without each optimization respectively.
(4) With the same 24-hour budget, HeteroFuzz would find 61.8X more divergence-inducing inputs compared to naive fuzzing.

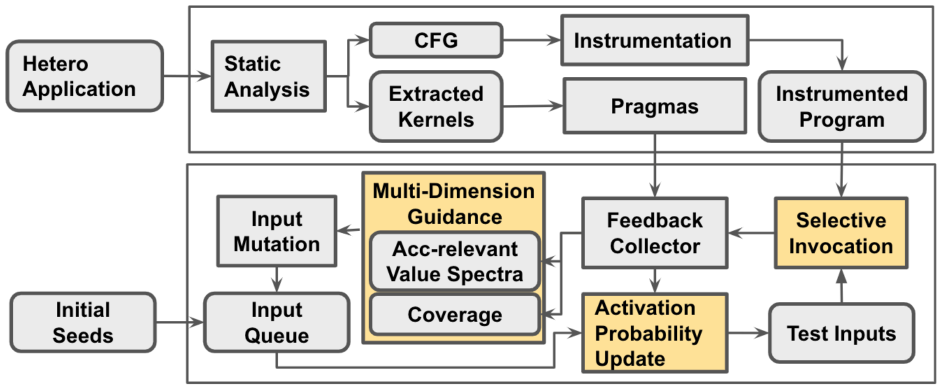

Figure 1: HeteroFuzz's three-pronged approach: multi-dimensional guidance, probabilistic mutations, and selective invocation

As the current need and prevalence of developing heterogeneous applications are rising significantly in the new era of *cloud-based hardware accelerator microservices* [4], the software engineering community must design a new automated testing technique to improve efficiency and effectiveness for this emerging category of heterogeneous applications. To our knowledge, HeteroFuzz is the first end-to-end approach that re-invents and adapts traditional fuzzing to heterogeneous applications. It speeds up differential testing under heterogeneous hardware platforms, and overcomes the limitation of long hardware simulation latency during fuzz testing, while finding significantly more divergence errors.

2 BACKGROUND

2.1 Heterogeneous Computing with FPGA HLS

Heterogeneous architectures with FPGAs have shown the potential to improve performance under strict energy constraints [16]. A heterogeneous application can be decomposed into *host* code executed on CPU and *kernel* code executed on accelerators.

High-Level Synthesis (HLS). HLS tools for FPGAs, such as Xilinx Vivado HLS [66], have raised the abstraction of hardware development by automatically generating RTL (Register-Transfer Level) descriptions from C/C++ code. During HLS, the frontend *schedules* each operation from the kernel code to certain clock cycle time slots. Next, it allocates the number and type of hardware unit *resources* used for implementing functionality. Finally, the *binding* stage maps all operations to the allocated hardware units. *This entire HLS process can take several minutes for simulation and several hours for synthesis, depending on the complexity of kernel logic.*

To achieve good quality-of-results (QoRs), HLS developers must insert synthesis directives and pragmas manually [19]. This requires having inter-disciplinary expert knowledge and knowing obscure platform-dependent details. Testing HLS code is difficult because the resulting FPGA may produce an outcome different from CPU due to the assumptions specified during accelerator synthesis. Below, we describe a few examples of how HLS directives and pragmas can produce a different execution outcome.

Custom Bitwidth. On-chip resource efficiency yields a higher level of parallelization. HLS supports arbitrary bitwidth for integers because reducing a variable's bitwidth could lead to resource reduction in FPGA directly. While the instruction set architecture (ISA) for CPU defines integer arithmetics at 32 bits by default, individual bitwidths could be programmed in FPGA. For example, if an integer variable *age* has a maximum value of 83—it only needs 7 bits instead of 32 bits. The programmer can insert a pragma `#typedef ap_uint<7> bit7` to declare an arbitrary precision integer of 7 bits. With custom bitwidth, FPGA accelerators have much more frequent overflows that would not happen on CPU, leading to failures or wrong outputs in the host code.

Memory Management. FPGA has no capability of on-chip memory management. Function calls to memory allocation are replaced by pre-allocating a static array with an estimated size and managing data elements manually. Similarly, recursions must be converted into iterations using a stack with a finite estimated size [52, 67, 68]. When an input or recursion depth exceeds the pre-estimated size, the FPGA kernel could crash or return a completely wrong output to the host code by accessing an unexpected address.

Parallelization. Reprogrammable hardware provides an inherent potential for parallelizing computation. Such parallelization can be done through pipelining of different computation stages and by duplicating processing elements or data paths to achieve an effect similar to multi-threading. To guide such parallelization, a developer must write HLS pragmas such as `#pragma HLS unroll`—creating multiples copies of the loop body which allows some or all loop iterations to occur in parallel or `#pragma HLS dataflow`—enabling task-level pipelining, which allows functions to overlap their operations. Such parallel execution can produce an outcome different from sequential execution on CPU, especially when a feedback path exists in two different functions or modules.

As described above, testing HLS code is different from C/C++ because HLS directives can cause differential behavior on accelerators.

2.2 Fuzz Testing

Without loss of generality, the procedure of fuzzing [39, 69] can be described as follows. Starting with an initial set I of seed inputs, the fuzzing procedure randomly selects one input from I, and generates new inputs by mutating several bits or bytes of the current input. In terms of which **mutations** to apply, it selects an available mutation from a given set of mutation operators. Commonly used input mutations are either bit-level or byte-level mutations in which random bits (or bytes) are flipped. It then collects the guidance feedback such as branch coverage by executing the instrumented

```
1  int main(int argc, char
         *argv[]){
2  int data[] =
         gradient(argv[1]);
3  int sum;
4  float th = argv[2];
5  for(i = 0 to data.
6       size())
7   sum+=data[i];
8  for(i = 0 to data.size()){
9    data[i] /= sum;
10   if(data[i] > th)
           keep(argv[3]);
11   else discard(argv[4]);}}
```

(a) Original Application

```
1  int main(int argc, char
         *argv[]){
2  int data[] =
         gradient(argv[1]);
3  int sum;
4  float th = argv[2];
5  int size = data.size();
6   accumulate(
7   data[size],int size);
8  for(i = 0 to data.size()){
9    data[i] /= sum;
10   if(data[i] > th)
           keep(argv[3]);
11   else discard(argv[4]);}}
```

(b) Host in Heterogeneous Application

```
1  #include "ap_int.h"
2   int accumulate(
3   int data[size],int size);
4   typedef ap_uint<8> bit8;
5   typedef ap_uint<9> bit9;
6   #define M 400;
7   #pragma HLS INTERFACE
8    m_axi port=data
9    offset=slave bundle=gmem
10  #pragma HLS INTERFACE
11   s_axilite port==data
12   bundle=control
13  #pragma HLS INTERFACE
14   m_axi port=size
15   offset=slave bundle=gmem
16 bit8 data_fpga[M];
17 bit9 i;
18 bit8 sum;
19 for (i = 0 to size) {
20  #pragma HLS unroll
21   data_fpga[i] = (bit8)data[i];}
22 SUM_LOOP: for (i = 0 to size)
23  #pragma HLS unroll factor=2
24   data_fpga[i] = (bit8)data[i];
25   sum + = data_fpga[i];}
26 sum = (int) sum;
27 return sum;}
```

(c) Kernel in Heterogeneous Application

Figure 2: Example Program Showing Divergence: Bitwidth

program with new inputs [32, 39, 69]. All inputs that enhance a **guidance** metric—e.g., exercising a new branch or leading to a unique crash—are then saved to the working set, S. Then, with S, the fuzzing procedure moves onto the next step of selecting an input from S and applying mutations to the input.

This fuzzing procedure is based on two inherent yet over-sighted assumptions: (1) it takes a minuscule amount of time to execute the target program, and (2) arbitrary mutations are likely to yield meaningful inputs. *Such assumptions do not hold for heterogeneous applications because hardware simulation takes several minutes and random mutations cannot account for hardware synthesis intricacies.*

3 MOTIVATING SCENARIO

This section presents motivating examples for why testing platform-dependent behavior is necessary. Suppose that Bob writes an image denoising application shown in Figure 2a. This application filters the gradients based on their percentages with respect to the sum of all gradients. Line 2 calculates the gradient vector of an input image by taking the absolute difference between two adjacent pixels. Line 4 defines the filtering threshold. Lines 5-7 aggregate the gradients across the entire vector. Lines 10-11 filter out the gradients less than the filtering threshold. Bob runs this C application on CPU and finds that the loop at lines 5-7 is a hot code path responsible for a significant execution time.

Therefore, Bob decides to convert this original application in C to a *heterogeneous application*. He refactors the loop at lines 5-7 in Figure 2a into an HLS kernel function `accumulate` in Figure 2c. The original application is then converted to *host code* in Figure 2b that still runs on CPU and communicates directly with a hardware accelerator generated from the *kernel code* in Figure 2c.

```
1  #include "ap_int.h"
2  void kernel{
3    #pragma HLS dataflow
4    fifo a; fifo b;
5    funX(a,b);
6    funY(a,b);}
7  void funX{
8    typedef ap_uint<1> bit1;
9    bit1 exist = b.read_nb(temp_b);
10    if (exist and temp_b == 3){
11    a.write(1);}
12   else{a.write(0);}}
13 void funY{
14   temp_a = a.read();
15   b.write(temp_a+3);}
```

Figure 3: Example Kernel Program Showing Divergence: Dataflow Mode

As discussed in Section 2, when writing kernel code, HLS developers must insert directives and pragmas to expose layout regularity and parallelism for FGPA accelerator synthesis. After analyzing the values of sample data sent to the kernel, Bob includes the `ap_int` header file at line 1 in Figure 2c to use custom bitwidths of 8 and 9 respectively in lines 4 and 5 instead of using 32-bit integers. He also pre-defines the max array size as 400 in line 6. Lines 7-15 define the data transfer interface between CPU and FPGA. Lines 19-21 offload `data` from CPU to `data_fpga` on FPGA. To exploit hardware-level parallelization, Bob inserts `#pragma HLS unroll` in line 23 to make two copies of the `SUM_LOOP` body. After compiling this heterogeneous application using Vivado HLS, CPU-side host code in Figure 2b will invoke the kernel `accumulate` at lines 6-7, send data to the synthesized FPGA, and wait for the returned result.

After converting this original application to a heterogeneous application, Bob may want to test it by using handcrafted inputs or randomly generating data. Bob must check whether this heterogeneous application produces divergent behavior on the same input. For example, when the kernel inputs `[1,1,1,253]` are sent to FPGA, Bob finds a divide-by-zero error in the host code at line 9 of Figure 2b, which does not happen when running on CPU only. When the kernel inputs `[2,1,1,253]` are sent to FPGA, the FPGA accelerator will return the sum as 1, instead of the correct sum 257 computed by CPU, leading to a completely wrong filtering output in lines 10-11 of Figure 2b. Finding such divergence-inducing inputs is extremely challenging—suppose Bob uses an AFL-like technique in an attempt to find such divergence-inducing inputs. AFL cannot distinguish the above two inputs that produce distinct divergence symptoms because no such errors happen using 32-bit integers on CPU and these two inputs are identical in terms of their branch coverage in host code.

Figure 3 describes another kernel example. In line 9, `read_nb()` is a `Boolean` type HLS read function which returns false if data is unavailable. When executing on CPU only, `funX` is executed before `funY` sequentially, so `fifo b` is always empty and line 11 can never be executed. As a consequence, `fifo a` is full of 0 and `fifo b` becomes empty. However, when executing on FPGA, `#pragma HLS dataflow` in line 2 enables parallel execution of `funX` and `funY`. Therefore, `fifo b` is not empty and the `if` condition at line 10 can evaluate to true, if the current data in b is 3.

HeteroFuzz is motivated by such challenge of testing heterogeneous applications described above. HeteroFuzz directly analyzes HLS pragmas in kernel code, monitors accelerator spectra and

Table 2: Mapping from HLS Pragmas to Accelerator Value Spectra

Category	HLS Directive /Pragma	Description of Pragma	FPGA-Relevant Value Spectra
Data Type	`typedef ap_uint<x> bitx`	Define an unsigned Integer with a custom bitwidth	The actual variable values with this type
	`typedef ap_int<x> bitx`	Define a signed Integer with a custom bitwidth	
Memory	`#pragma HLS array_partition`	Partition an array into smaller arrays or individual elements	The set of accessed offsets
	`#pragma HLS array_reshape`	Combine array partitioning with vertical array mapping	
	`#define size M`	Define the estimated array size	
Recursion	N/A	N/A	The actual used size of a stack and the number of iterations
Parallelization	`#pragma HLS dataflow`	Enable task-level pipelining	The FIFO queue size if feedback path exists
	`#pragma HLS pipeline`	Allow concurrent execution of operations	
Loop	`#pragma HLS unroll`	Create multiples copies of the loop body	The actual number of iterations
	`#pragma HLS loop_tripcount`	Specify the total number of iterations	
	`#pragma HLS loop_flatten`	Allow nested loops to be flattened into a single loop hierarchy	
	`#pragma HLS loop_merge`	Merge consecutive loops into a single loop to reduce overall latency	
Interface/ Configuration	`#pragma HLS INTERFACE` etc.	Perform input and output operations using a specific I/O protocol in the design interface	N/A

branch coverage in host code, adjusts the activation probabilities of mutation operations, and reduces unnecessary simulations to find divergence errors.

4 APPROACH

HeteroFuzz contains three novel components that work in concert to detect platform-dependent differential behavior for heterogeneous applications. Figure 1 describe its overall architecture: multi-dimensional guidance feedback based on accelerator spectra (Section 4.1), (B) dynamic probabilistic mutations (Section 4.2), and (C) selective invocation to reduce the latency of repetitively invoking a hardware simulator (Section 4.3). Its three-pronged approach is based on two key insights: platform-dependent behavior can be exposed by monitoring accelerator-relevant spectra and accelerator synthesis is optimized based on kernel input values; therefore multiple invocations within the same value range may expose redundant divergence symptoms. Although HeteroFuzz is designed for FPGA accelerators, its key idea can be easily extended to other heterogeneous platforms by monitoring platform-specific spectra.

4.1 Accelerator Spectra Monitoring

Inputs with the same branch coverage can still have distinct impacts on hardware characteristics; HeteroFuzz augments branch coverage feedback with hardware-dependent characteristics to detect divergence symptoms when running host and accelerator together. Prior studies [40, 41] find that numerous severe security problems originate from such host-accelerator interaction.

Coverage Feedback in Host Code. HeteroFuzz instruments the host program based on its extracted control flow graph (CFG) using LLVM [36]. Each node in a CFG represents a basic code block and each edge (A, B) represents a transition between two blocks A and B in the CFG. Similar to AFL, HeteroFuzz initializes an array with binary bits called `trace_bits`, with zeros, in each fuzzing iteration. *Iteration* here means one execution of a target program with a generated test input. Each bit represents a branch in the program. If an edge (A, B) is exercised, HeteroFuzz updates the corresponding entry from 0 to 1 in `trace_bits`.

Accelerator Value Spectra. Tracing branch coverage in a synthesized hardware kernel is infeasible, because HLS takes kernel code as input but implements branching logic using a pipeline of multiplexers at the hardware level. Each branch in the synthesized hardware is executed in parallel as long as prior signals are ready; however, the output being produced by the hardware logic for an untaken branch is never used. Thus, branch coverage cannot be derived directly from the synthesized hardware [29]. Branch coverage collected from the host code only is ineffective as a guidance feedback due to its inability to detect errors in heterogeneous applications, because inputs with the same coverage in the host code may exhibit different behavior, as discussed in Section 3.

HeteroFuzz analyzes the inserted HLS pragmas and traces their associated FPGA-relevant spectra as *accelerator feedback*. It currently supports five kinds of value spectra, shown in Table 2. These mappings are user-extensible by modifying a configuration file.

Figure 2c is an accumulation FPGA kernel with ten HLS directives. It uses `typedef ap_uint<8> bit8` and `typedef ap_uint<9> bit9` to customize 32-bit integer data into 8 and 9 bit integers respectively. HeteroFuzz monitors the actual values of variables declared with these types, such as `data_fpga`, `sum`, and `i` in lines 22-24 of Figure 2c. HeteroFuzz ignores the pragmas in lines 7-21 as HLS data transfer interface has no impact on kernel logic. The pre-allocated array `data_fpga` in line 22 is declared with a maximum size of 400. HeteroFuzz monitors the accessed offsets in case an unexpected address is accessed. The `unroll` pragma in line 29 makes two copies of the loop body. HeteroFuzz records the actual iteration counts, which can be used together with array access offsets to detect potential divergence when the size of offloaded data is not multiple of 2. Then HeteroFuzz instruments the kernel to record these three types of values spectra for Figure 2c: (1) the value range of `fpga_data`, output variable `sum`, and intermediate variable `i`; (2) the set of accessed offsets in array `data_fpga`; and (3) the actual number of loop iterations.

Similar to how AFL [69] keeps track of a single test's branch coverage and cumulative branch coverage for all tests, HeteroFuzz keeps track of value spectra for each test execution and cumulative value spectra. In each test execution, it initializes an array `acc_feedback`, where each entry has four fields: `<value spectra type`, `name`, `min value`, `max value>`. It then updates `total_feedback` to record cumulative value spectra for all seen inputs that can safely execute on the synthesized FPGA. For example, when the kernel inputs are `[1,1,1,253]`, the tracked hardware feedback is shown in column FPGA Accelerator Spectra in Table 3. While coverage-guided

Table 3: Example Execution of Generated Inputs

	Act.	Program	Kernel		FPGA Accelerator Spectra				New		Mutation	Memorization	
ID	Mut.	Inputs	Inputs	Invoke	Type	Name	Min	Max	Branch	Overflow	Probability	Range	Size
seed	N/A	[1,2,1,2,11]	[1,1,1,9]	yes	input_value	fpga_data	1	9	yes	no	[0.17,0.17,0.17,	(1,9)	4
					variable_value	i	0	3			0.17,0.17,0.17]		
					variable_value	sum	2	12					
					mem_offset	fpga_data	[0,1,2,3]						
					loop	SUM_LOOP	N/A	2					
1	M6	[1,2,1,2,2]	[1,1,1,0]	**no**	N/A	N/A	N/A	N/A	no	N/A	[0.17,0.17,0.17,	(1,9)	4
											0.17,0.17,0.17]		
2	M3	[1,2,1,2,255]	[1,1,1,253]	yes	input_value	fpga_data	**1**	**253**	**no**	**yes**	[0.16,0.16,**0.22**,	**(1,9)**	4
					variable_value	i	0	3			0.16,0.16,0.16]		
					variable_value	sum	**0**	**3**					
					mem_offset	fpga_data	[0,1,2,3]						
					loop	SUM_LOOP	N/A	2					
3	M3	[0,2,1,2,255]	[2,1,1,253]	yes	input_value	fpga_data	1	253	no	yes	[0.16,0.16,0.22,	(1,9)	4
					variable_value	i	0	3			0.16,0.16,0.16]		
					variable_value	sum	**1**	**4**					
					mem_offset	fpga_data	[0,1,2,3]						
					loop	SUM_LOOP	N/A	2					
4	M1	[1,2,1,2,11,36]	[1,1,1,9,25]	**yes**	input_value	fpga_data	1	25	no	no	[0.21,0.15,0.21,	(1,25)	5
					variable_value	i	0	5			0.15,0.15,0.15]		
					variable_value	sum	2	37					
					mem_offset	fpga_data	[0,1,2,3,4,5]						
					loop	SUM_LOOP	N/A	3					

fuzzing would discard the inputs for not achieving new branch coverage, HeteroFuzz saves them for increasing the value spectra of `fpga_data` and `sum`. Based on the collected branch coverage feedback and monitored FPGA accelerator spectra, an input will be kept in the generated tests, if it increases either kind of feedback.

4.2 Probabilistic Mutations

Designing mutations to detect platform dependent behavior in heterogeneous applications is challenging because: (1) mutations modify the inputs of a host program, not just those host inputs related to kernel inputs, and (2) different mutations contribute to divergence-inducing inputs in different degrees.

We propose dynamic probabilistic mutations to increase the chance of detecting hardware-dependent behavior. These mutations represent input modifications to explore the input space during fuzz testing, as opposed to code modifications used in mutation testing. Input mutations are directly applied to inputs in host code. We classify those inputs into *kernel-sensitive inputs* and *kernel-irrelevant inputs*. Kernel-sensitive inputs refer to a subset of host program inputs that will be offloaded to an FPGA kernel, and all other inputs are kernel-irrelevant inputs. To identify kernel-sensitive inputs, HeteroFuzz uses static backward slicing [53, 55] of the argument names, `data` and `size` of the kernel function, `accumulate` all the way to the original input arguments in the host code. For example, in Figure 2b, starting from the invocation of `accumulate` at lines 6 and 7, we use backward slicing on its arguments, `data` and `size`, in turn marking line 5 and line 2 where `data` and `size` are defined respectively. Starting from line 2, it then marks `argv[1]`, which is an input to the host code's `main` function in line 1. A gradient threshold `argv[2]` is used in in line 4, `argv[3]` is used at `keep` in line 10, and `argv[4]` is used at `discard` in line 11, making them irrelevant to `accumulate` in Figure 2c. Therefore, we label the input argument `argv[1]` as a kernel-sensitive input. HeteroFuzz initializes the selection probability of individual host inputs to be mutated as follows:

$$P_i = \begin{cases} 1 & \text{if i is a kernel-sensitive input;} \\ a & \text{otherwise} \end{cases} \tag{1}$$

That is, the kernel-sensitive inputs are always selected and mutated. a is the probability of selecting a non-kernel input, as it is still necessary to mutate all inputs in the host code to exercise diverse behavior and increase branch coverage in the host code. In our experiments, we use $a = 0.10$ as default.

Considering that data offloaded to a hardware kernel is often an array or a matrix, HeteroFuzz uses six basic mutations extended from AFL to generate new inputs, shown as follows. Scala values are treated as one element array. The reason why we focus on arrays and matrices is that hardware accelerators are designed to batch process multiple elements in parallel. All divergence errors that can be found by naive fuzzing should be all found by HeteroFuzz as well because HeteroFuzz 's input mutations are a superset of naive input mutations.

- **Data Size Mutation (M1)** inserts/deletes one or several random elements if the input data is an array: e.g., from `[1,2,3]` to `[1,2,3,4]` or `[1,3]`
- **Data Dimension Mutation (M2)** adds/removes one or several columns if the input data has multiple dimensions: e.g., from `[[1,2,3],[3,2,1]]` to `[[1,2,3]]`.
- **Data Element Mutation (M3)** mutates the value of one element based on its type: e.g., from `[1,2,3]` to `[1,2,4]`.
- **Type Mutation (M4)** modifies the type of a selected entry, while keeping the same value: e.g., from `Integer` to `Float`.
- **Bit Mutation (M5)** flips one or several bits.
- **Byte Mutation (M6)** flips one or several bytes.

While conventional fuzzing does not update the activation probabilities of mutations and generally keeps them uniform, HeteroFuzz assigns different activation probabilities to individual mutations and updates their probabilities based on accelerator spectra feedback. If a new child input generated by mutation m increases the monitored accelerator spectra, m will be labeled as a *favored* mutation. Favored

Table 4: Subject Programs

ID	Subject Program	Kernel	Description	# of Symptoms HF	# of Symptoms WithoutSpectra	% of Div-inducing Inputs HF	% of Div-inducing Inputs WithoutSpectra
P1	Median Filter	Bubble Sort	Blur an image by replacing the pixel value to a median	6	1	68.2%	6.8%
P2	Median Filter	Merge Sort	Blur an image by replacing the pixel value to a median	6	1	37.3%	4.5%
P3	Image Denoising	Accumulation	Denoise an image based on analyzing image gradients	5	0	71.0%	0%
P4	KNN	L2norm	Finds the top-k most relevant elements	5	2	72.9%	14.5%
P5	Signal Transmission	RGB2YUV	Transform RGB signals to YUV signals	7	2	40.0%	4.5%
P6	3D Rendering	Rendering	Render an image based on 3D information	9	2	56.0%	10.0%
P7	Face Detection	Detection	Detect human faces in an image	9	2	52.2%	5.0%

mutations will then have higher activation probabilities. Given the activation probabilities of l mutations $P = \{P_0, P_1, ...P_{l-1}\}$, they will be updated dynamically:

$$P_m = \begin{cases} P_m + \alpha & \text{if } m \text{ is chosen and it} \\ & \text{increases spectra} \\ P_m - \frac{\alpha}{l-1} & \text{otherwise} \end{cases} \quad (2)$$

l is the number of mutations. Since we have six mutations, here l is 6. Every P_m is initialized as $1/l$ and α is the update factor that is pre-defined as 0.05. In column Mutation Probability of Table 3, the activation probability for each mutation is initialized to $1/l = 0.17$. In the second execution (ID 2), inputs created by mutation M3 increase the spectra of `input_value` and `variable_value`. HeteroFuzz consequently increases M3's activation probability from $P_m = 0.17$ to $P_m + \alpha = 0.22$, and adjusts the probabilities of other mutations to $P_m - \frac{\alpha}{l-1} = 0.16$.

4.3 Selective Invocation

To reduce the long latency of repetitive hardware simulation, HeteroFuzz maintains the range of seen kernel input values that run correctly on FPGA Memorization/Range in Table 3, and their data size, Memorization/Size in Table 3. We use a global variable `enable_sim` to indicate whether hardware simulation should be invoked or not. This variable is set to be true if the range or the data size of seen kernel inputs grow beyond the current records. Suppose that HeteroFuzz had seen a set of concrete values [2,3,5] for an integer array `x`, we maintain the range of `x` as $(2, 5)$. It is *safe* to record only the range $(2, 5)$ instead of considering combinations of concrete values as a set $\{2, 3, 5\}$ for the purpose of finding a new, distinct divergence error symptom. If an error were to happen for an unseen combination such as $\{2, 4, 5\}$. This error symptom will be identical to the error symptoms that one would get for executing a value less than 2 or greater than 5. This is due to the unique property of FPGA synthesis, where an integer or fixed-point variable maps to a contiguous range of values.

In Table 3, for ID `seed`, column Memorization/Range is updated to (1,9) for variable `fgpa_data`, as the smallest item is 1 and the largest item is 9 and its execution does not lead to any error on FPGA execution. For ID 1, since the kernel input [1,1,1,0] is still within the range of (1,9), HeteroFuzz skips hardware simulation. For ID 2, since the kernel input [1,1,1,253] now includes a value 253 that goes beyond the range of (1,9), HeteroFuzz invokes hardware simulation and finds that an overflow signal is captured on FPGA, indicating unsafe accelerator execution. Though the max value of this input is 253, HeteroFuzz keeps the range of *safe* execution as (1,9).

As Memorization/Range is updated for FPGA-safe executions, it is guaranteed to be narrower than or equal to the range of data that the accelerator designer uses to optimize the FPGA data type. Therefore, FPGA execution with inputs that fit the memorized range can either be fully correct or only report a redundant divergence symptom arising from trying out a new value exceeding the safe range. Thus, we can safely obviate such executions to speed-up the input generation process. What we mean by "safely obviate" is that, for the purpose of finding a new divergence symptom, it does not matter because you will get the same kind of divergence error.

5 EVALUATION

We evaluate following research questions:

RQ1 How effective is HeteroFuzz's accelerator spectra monitoring in generating divergence-inducing inputs?
RQ2 How much speed-up is enabled by HeteroFuzz's dynamic probabilistic mutations?
RQ3 How much speed-up is enabled by HeteroFuzz's selective invocation?
RQ4 How effective and efficient would HeteroFuzz be, in comparison to naïve fuzzing using an AFL-like technique?
RQ5 Can input range checking obviate the need of fuzz testing and still find platform-dependent errors?

Benchmarks. Our benchmarks include seven real-world and publicly available heterogeneous applications written in C/C++ with FPGA kernels, listed in Table 4. P1-P3 are from OpenCV [42] examples, P4-P5 are from [30], and P6-P7 are from Rosseta [71]. Our subject selection criteria is based on whether the programs cover different synthesis optimizations, and whether a diverse set of HLS pragmas is used for detecting platform-dependent divergence behavior. Seven programs in Table 4 cover all twenty four kinds of pragmas (*e.g.,* custom bitwidth, parallelization, memory management, etc.), and thus activate different kinds of synthesis optimizations.

These subject programs may look small in size, but they are actually larger than kernel benchmarks the FPGA community uses [24, 49, 71] and real-world heterogeneous applications described in Xilinx posts [57–59]. Building a hardware accelerator is similar to designing a new instruction type in CPU instruction set architecture (ISA). Most kernel code is in order of tens of lines, as it maps directly to hardware circuits. In fact, in a usual FPGA development workflow, developers instrument software on CPU, find out its hot code path corresponding to tens of lines of code, and extract it as a separate kernel for FPGA synthesis. Therefore, our work cannot be judged under the same scalability standard used for pure software research (e.g., handling GitHub projects with millions of lines of code). Simply put, the current landscape of heterogeneous platforms cannot handle FPGA synthesis of such large kernel size.

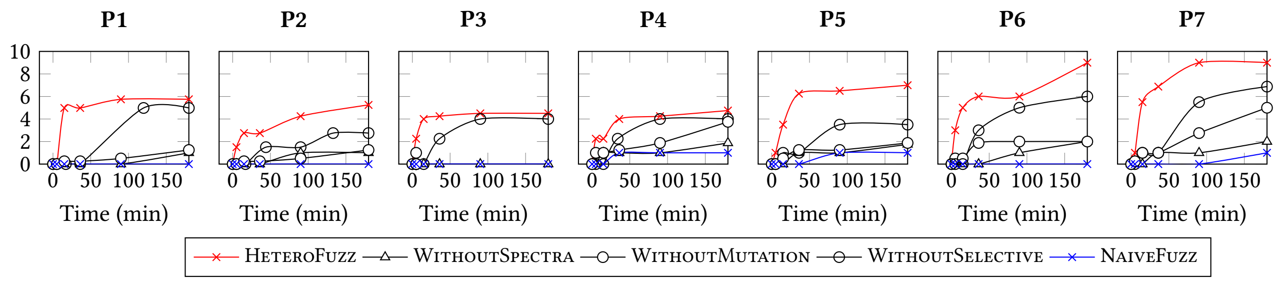

Figure 4: Number of Unique Divergence Symptoms

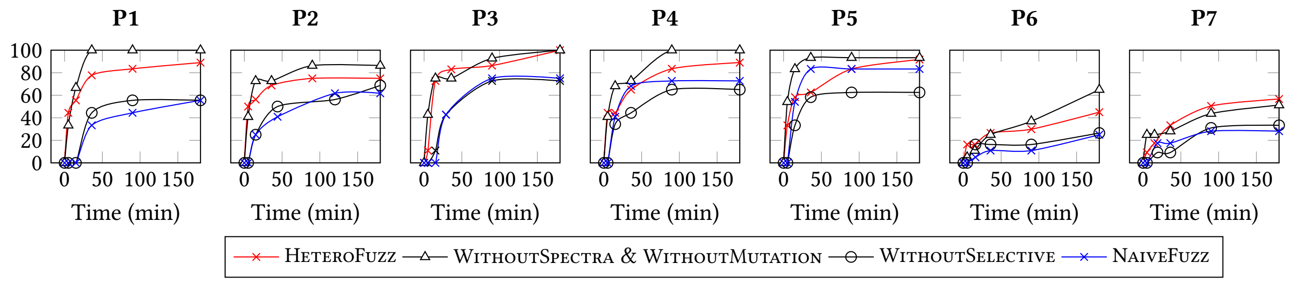

Figure 5: % Cumulative Branch Coverage in Host Code

To answer the research questions above, we create the following four baseline versions by downgrading HeteroFuzz.

- WithoutSpectra: This option disables accelerator spectra monitoring from HeteroFuzz to measure how effectively HeteroFuzz can find more divergence errors by monitoring accelerator-relevant feedback.
- WithoutMutation: This option disables dynamic probabilistic mutations from HeteroFuzz to measure how fast HeteroFuzz can find the same divergence errors by increasing the probabilities of divergence-inducing mutations.
- WithoutSelective: This option disables selective invocation from HeteroFuzz to measure how fast HeteroFuzz can find the same divergence errors by obviating the need to invoke an HLS simulator that finds redundant errors.
- NaiveFuzz: This option enables only branch coverage as guidance and invokes an HLS simulator for every input. We estimate its running time by multiplying the number of invocations required for WithoutSpectra with the average HLS simulation time.

Experimental Environment. All experiments are done by leveraging Vivado Design Suite 2018.03 to simulate kernel execution on Xilinx Virtex UltraScale+ XCVU9P FPGA.

5.1 RQ1: Benefit of Accelerator Spectra

To evaluate HeteroFuzz's guidance strategy that monitors accelerator spectra in addition to branch coverage, we generate inputs for P1-P7 by running HeteroFuzz and WithoutSpectra for three hours. With the generated inputs, we execute the program on CPU versus a heterogeneous platform that runs *host* on CPU and *kernel* on the FPGA/HLS similuator. We then measure the percentage of divergence-inducing inputs and the number of unique divergence symptoms. This experiment is done over ten independent runs and the rightmost two columns in Table 4 report the results (HeteroFuzz in column HF). On average, 56.8% of inputs generated by HeteroFuzz is divergence-inducing, while 6.48% of inputs generated by WithoutSpectra is divergence-inducing. This is because WithoutSpectra takes the FPGA accelerator as a black box and enlarges the covered branches in host code only. On the contrary, HeteroFuzz monitors branches and accelerator spectra in tandem, leading to 7.78X more divergence-inducing inputs.

We then group divergence-inducing inputs into a set of unique symptoms, because different inputs may exhibit the same kind of a divergence error. In Figure 4, Y axis is the average cumulative number of detected symptoms. Within the same time budget, WithoutSpectra detects 10 unique symptoms in total, while HeteroFuzz detects 47, almost 3.7x more divergence symptoms. *It is also important to note that existing inputs shipped with the original benchmark do not find any divergence errors; in other words, HeteroFuzz has the potential to detect real-world platform-dependent errors proactively.*

We also assess speed-up enabled by HeteroFuzz by measuring the time taken to find the same set of divergence symptoms found by WithoutSpectra. Across seven applications, WithoutSpectra takes total 21 hours to find 10 unique symptoms, while HeteroFuzz takes only 0.52 hours, demonstrating 40X speed-up.

Table 5 lists five sample symptoms found in P3. We describe why these divergent behavior appear between CPU and FPGA and how HeteroFuzz finds them in detail below.

First, as with most hardware designs, the kernel in P3 uses optimized bitwidths for data offloaded from CPU to FPGA and its intermediate variables. When a large number 2147483600 is sent to the kernel, it only keeps eight most significant bits or least significant bits in the binary representation of 2147483600 and cuts off the others, leading to a wrong result. When the inputs `[1,1,1,253]` and `[2,1,1,253]` are executed on FPGA kernel, runtime overflow happens with variable `sum` in line 25 of Figure 2c, leading to a divide-by-zero error in host code and a wrong returned result respectively. Although integer overflow can happen in CPU as well, it shows up much more frequently in accelerators, and it is not easy to predict the consequent impact. HeteroFuzz monitors the value ranges of

Table 5: Example Divergence Symptoms for P3

ID	Symptom	Description	Input Checking
S1	Unexpected Endless Loop	Memory overflow happens when FPGA attempts to write `data_fpga` at an unexpected address in line 21 of Figure 2c.	×
S2	Kernel Offloading Error	Host is offloading a large number 2147483600 to FPGA in line 21 of Figure 2c, leading to a wrong returned result.	✓
S3	Kernel Runtime Overflow	The value of intermediate variables in line 25 of Figure 2c exceeds its bitwidth capacity, leading to a wrong result.	×
S4	Divide by Zero in Host	FPGA returned result leads to divide-by-zero error in line 9 of Figure 2.	×
S5	Incorrect Loop Unrolling	CPU and FPGA produce different results when the input array size is not multiple of 2.	✓

inputs and intermediate variables to increase the chance of showing differential behavior caused by accelerator integer overflows.

Second, when a program attempts to access an invalid or illegal address in CPU, the memory management unit can give exception signals such as `Segmentation Fault`. However, in FPGA, all memory accesses are mapped to a legal physical address on BRAM, which can result in severe security problems and unexpected accelerator behavior. HeteroFuzz generates input `[1,...,0]` wherein the 401st element is 0. Line 21 in Figure 2c then writes this over-ranged `data_fpga[400]` to the address of a loop iterator `i`, leading to an endless loop execution of lines 19-21.

Third, to further complicate the difficulty of finding divergence errors, because P3 makes two copies of the loop body during FPGA synthesis to enable parallelization, a wrong result happens only if the size of an input array is not multiples of the unroll factor 2.

HeteroFuzz and WithoutSpectra achieve similar coverage, as shown in Figure 5. The Y-axis indicates the percentage of covered branches in host code. For all applications except P7, HeteroFuzz's coverage grows slightly *slower* than WithoutSpectra. This is because HeteroFuzz increases the activation probabilities of mutations that lead to a new accelerator feedback, pushing the fuzz engine to explore more platform-dependent divergence rather than enlarging branch coverage in host code.

Summary 1

HeteroFuzz finds 7.78X divergence-inducing inputs (3.7X unique divergence symptoms) by monitoring accelerator spectra in addition to the branch coverage of host code.

5.2 RQ2: Benefit of Probabilistic Mutation

To evaluate the benefit of dynamic probabilistic mutations, we create a downgraded version WithoutMutation that disables probabilistic mutations from HeteroFuzz. We assess how fast HeteroFuzz and WithoutMutation detect divergence symptoms within the three-hour limit. We repeat the experiments ten times and report average results in Figure 4.

In total, HeteroFuzz detects 47 unique symptoms, while WithoutMutation reports 18, given the same time limit. Compared to

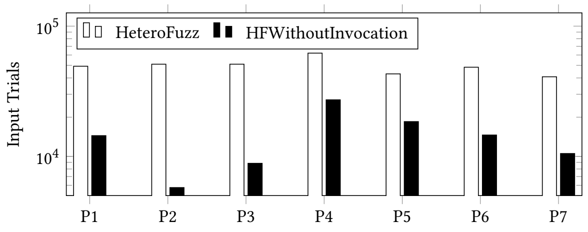

Figure 6: Number of Input Trials.

WithoutMutation, HeteroFuzz finds the same 18 divergence symptoms reported by WithoutMutation 17.5 times faster, taking only 1.2 hours as opposed to 21 hours. For example, in P3, while HeteroFuzz finds five divergence errors, WithoutMutation finds only one divergence error by generating input `[1,1,1,311]` that leads to an error, because the max value to be offloaded to FPGA is 255. The other four errors are not found by WithoutMutation, because it wastes most fuzzing time on generating type-invalid data for the FPGA kernel, such as `[1,1,1#]`, when an integer array is expected. HeteroFuzz leverages probabilistic mutations to increase the chance of hitting divergence behavior by isolating kernel-sensitive inputs and prioritizing divergence-inducing mutations such as modifying the value of a specific element in the array. The achieved branch coverage of WithoutMutation is the same with the coverage of WithoutSpectra, as shown in Figure 5, because no accelerator spectra monitoring implies that dynamic mutations cannot be enabled. (However, when turning off dynamic mutations, HeteroFuzz can still use accelerator spectra as a guidance feedback.)

Summary 2

HeteroFuzz achieves 17.5X speed-up in detecting the same set of errors by dynamically adjusting the activation probability of divergence-inducing mutations.

5.3 RQ3: Benefit of Selective Invocation

To assess speed-up enabled by selective invocation of a hardware simulator, we compare HeteroFuzz with a downgraded version WithoutSelective. We measure the number of inputs the tools can explore within the same 24-hour budget.

In Figure 6, the Y-axis is the average number of input trials over ten independent runs. In P2, WithoutSelective can invoke a simulator with 5760 inputs only, while HeteroFuzz can attempt over 51k inputs, achieving 8.8X speedup. HeteroFuzz achieves such speedup by skipping the repetitive simulation calls when inputs hit the memorized value range of seen kernel inputs. This selective invocation saves time but does not sacrifice fault detection capability, because kernel inputs contained within already seen ranges can lead to either correct FPGA executions or already discovered divergence error symptoms. Collecting kernel input values does not incur additional hardware level instrumentation, as such information can be extracted from HLS simulation results.

Summary 3

By reducing unnecessary hardware simulation calls, HeteroFuzz speeds up differential testing by 8.8X without sacrificing fault detection capability.

5.4 RQ4: Comparison against Naive Fuzzing

Fuzz testing is often built on an implicit assumption that *the program under test can be executed millions of times within a matter of hours*. However, such assumption does not hold for heterogeneous applications due to the long latency of hardware simulation. We estimate the time required for NaiveFuzz by multiplying the number of iterations required to find the same kind of divergence symptom using WithoutSpectra (—where one iteration means running a program on a new input) with an average hardware simulation time for each program. Figure 4 shows comparison between the running time of HeteroFuzz and the estimated time of NaiveFuzz. Please note that though we report an estimated time not an actual running time, because HW simulation time does not vary much for each input (the standard deviation $\sigma = 1.62$), this estimated should be highly similar to the actual time of using an AFL-like technique that directly invokes an HLS simulator with a new input. Within three hours, NaiveFuzz finds only one divergence symptom for P4, P5 and P7, but could not find any in other programs. For P2, NaiveFuzz requires at least 2262 hours to detect all symptoms detected by HeteroFuzz within three hours, leading to 754X speed-up. In addition to the results in Figure 4, we ran HeteroFuzz for 24 hours and compare against what NaiveFuzz would find within the same time budget. HeteroFuzz detects 61.8X more error-inducing inputs with the same budget of 24 hours, compared to NaiveFuzz.

Summary 4

Using an AFL-like technique to repetitively invoke a hardware simulator would be too slow and insufficient to reveal platform-dependent errors in heterogeneous applications.

5.5 RQ5: Comparison against Input Checking

One may question whether input validity checking on the side of the host code is feasible and adequate for preventing platform-dependent errors in kernel code to be executed on a hardware accelerator. Unlike pure software systems where a caller function can prevent errors by checking the pre-condition of its callee prior to invocation, such input validity checking is not always feasible in heterogeneous applications [30, 40]. The reason is that *it is nearly impossible to identify the precise pre-condition in advance due to the difficulties of modeling individual FPGA devices* [12, 28, 70], because the pre-condition is dependent on the resource availability on a specific platform. For example, when a merge-sort kernel requires a 5MB array for dynamic block memory, Xilinx-Zynq-7030 with 9.3MB BRAM will work fine, but Xilinx-Zynq-7020 with 4.9MB BRAM will produce an incorrect sorting result silently.

To further substantiate this argument, we conducted a case study on several divergent symptoms found by HeteroFuzz in the example application shown in Figure 2. Table 5 summarizes a divergence symptom, a detailed error description, and whether input range checking could have prevented such error. After analyzing the HLS pragmas in line 4, line 6, and line 23, we manually extract the pre-condition of kernel code in terms of a range check and insert it an input guard: *an integer array whose size is multiple of two but no larger than 400, and each element in this array should be less than 256.* After inserting this guard into the original host code, divergence symptoms S2 and S5 are prevented because the accelerator is trying to process data with an invalid value or size. However, such input checking is still inadequate and does not prevent the platform-dependent error S1 and kernel runtime errors S3 and S4. For example, S4 is caused by inputs that satisfy the guard condition, `[1,1,1,253]`. In other words, assertions inserted by a defensive developer in the host code may not fully prevent runtime errors coming from hardware accelerators due to varying resource availability of individual FPGA devices. To our knowledge, HeteroFuzz is the only testing tool that can detect such platform-dependent errors missed by input checking in host code.

Summary 5

Our case study shows evidence that even if a developer manually constructs and inserts assertions in host code, kernel errors from accelerators cannot be fully prevented.

6 THREATS TO VALIDITY

We discuss the threats to validity as follows.

Device Dependence. We simulate all the kernel executions on a single Xilinx Virtex UltraScale+ XCVU9P FPGA, which is currently the widely used FPGA. This setup may restrict the generalizability of our results to other devices, because the detected divergence symptoms could vary for different platforms, e.g. Intel's Altera. Though the absolute numbers of execution time and symptoms are dependent on a detailed configuration, we believe that HeteroFuzz would retain the overall benefits of speedup and divergence-finding capability when it is applied to different platforms. Since HeteroFuzz uses FPGA simulation, it does not find mechanical failures caused by temperatures, aging of devices, and radiation on FPGA. Such hardware in-field testing is often done by device physicists.

Time Limit. We empirically set three hours as the time limit for fuzzing. Longer execution time may expose more divergence errors or more execution paths as suggested in [27]; however, this time limit is reasonable, as we did not see any increase in new types of divergence errors with a higher time limit for subjects P1-P7.

Input Mutations. In terms of *mutations*, HeteroFuzz refers to input modifications to explore the input space during fuzz testing, as opposed to injecting code faults in mutation testing. HeteroFuzz not only is faster than naive fuzzing but is *safe*—i.e., HeteroFuzz can find all errors that can be found by naive fuzzing, as HeteroFuzz's input mutations are a super-set of low-level input mutations. Designing new kinds of input mutations could affect the efficiency of fuzz testing. Currently, there are no equivalent high-level input mutations in HeteroFuzz. Low-level bit or byte mutations retained by HeteroFuzz could subsume other high-level input mutations, because combinations of low-level mutations could map to high-level mutations.

7 RELATED WORK

Fuzz Testing. Fuzzing has gained popularity in both academia and industry due to its black/grey box approach with a low barrier to entry [43]. The key idea of fuzz testing originates from random test generation where inputs are incrementally produced with the hope to exercise previously undiscovered behavior [18, 20, 44]. For example, AFL mutates a seed input to discover previously unseen coverage profiles [69]. To carefully explore a vast space of inputs and unbounded program paths, Lemieux et al. create custom mutations so that the generated inputs gravitate toward exercising rare branches [33]. Other approaches incorporate symbolic execution with fuzzing to guide selection and mutation of the inputs to invoke unique program paths [7, 51]. Padhye et al. incorporate the semantic validity of input mutations in Zest [45] to reduce the search space of inputs by mapping low-level, bit-level mutations to valid structural changes in the high-level input representation. All these fuzzing techniques are built on the assumption that the program under test can be executed millions of times within a matter of hours. However, in the domain of heterogeneous applications, a single invocation of a hardware simulator may take several minutes, which is the exact problem that HeteroFuzz addresses.

Instead of using coverage as guidance, several techniques have investigated how to use custom guidance mechanisms. PerfFuzz [32] uses the execution counts of exercised instructions together with branch coverage as fuzzing guidance to explore pathological performance behavior. UAFL [54] incorporates typestate properties and information flow analysis to detect the use-after-free vulnerabilities. MemLock [56] employs both coverage and memory consumption metrics to guide the fuzzing process. AFLgo [2] extends AFL to direct fuzzing towards user-specified target sites. However, none explicitly monitors hardware-level accelerator spectra and metrics to reveal platform-dependent divergence like HeteroFuzz.

Another angle to optimize fuzz testing is to update which mutation operations to apply. SymFuzz [8] uses symbolic execution to determine the number of bits to be mutated in a seed input. Angora [9] updates mutation operations to be aware of taint-level observations. SDF [35] uses seed properties to guide mutation in web-browser fuzz testing. In grammar-based fuzzing, Saffron [31] repairs the given grammar based on whether the program accepts unexpected inputs outside of the provided grammar, and then it adaptively refines the probabilities of every production rule. MOPT [38] finds an optimal probability distribution for mutation operators to discover vulnerabilities more efficiently. To our knowledge, none designs probabilistic mutations by associating monitored accelerator spectra with the probability of activating specific mutation operators.

Testing in HLS and Hardware Accelerators. HLS tools automatically generate RTL descriptions from C/C++ programs. Yann et al. [25] test HLS by randomly generating programs and verifying the equivalence between the synthesized design and the original code. Christopher et al. [34] investigate many-core compiler fuzzing in the context of heterogeneous computing with OpenCL kernels. They report more than 50 OpenCL compiler bugs. Silver [37] proposes a single end-to-end correctness theorem about running a verified compiler on a verified FPGA platform. It generates machine code for Silver based on a high-level executable specification, and the synthesized FPGA hardware will have the observable behavior of the original high-level specification. Different from HLS compiler testing that finds bugs in compilers, HeteroFuzz detects platform-dependent behavior in heterogeneous applications.

HeteroFuzz focuses on testing software applications with host code and kernel code combined together. In other words, HeteroFuzz's problem concerns C-like code testing, where a sub region of code could be offloaded to FPGA accelerators. On the other hand, the hardware design community targets *circuit* verification in the form of bitstream and/or hardware description languages (HDL) such as Verilog, VHDL, etc., using formal verification [26, 48] and runtime verification [29]. For example, RFUZZ [29] is a circuit runtime verification tool for FIRRTL IR (UC Berkeley's own version of RTL language). RFUZZ invents a new notion of *MUX toggle cooverage* for circuit testing at gate level and employs a rapid memory resetting on FPGA for RTL circuit verification. As another example, Qin and Mishra [48] present a scalable test generation technique [48] for hardware kernels in Verilog by interleaving concrete and symbolic execution to bridge the gap between model checking and testing. As opposed to these techniques that find crashes on kernels only, HeteroFuzz targets *end-to-end application code testing* and reveals differential behavior of the entire heterogeneous application (i.e., host and accelerator together) under different platforms. In other words, it is not feasible to directly compare HeteroFuzz against these circuit testing techniques [29, 48], because they do not have capability to test host code together with kernel code, and their input languages are Verilog variants, not C variants.

Revealing Precision Errors. FPGen [21] uses symbolic execution to generate inputs to trigger large numerical floating-point errors. It defines inaccurate precision loss checks and injects these checks at strategic program locations to construct specialized branches to induce floating-point errors. Different from FPGen that focuses on floating-point overflows and errors, HeteroFuzz has such a broad scope in generating inputs that lead to variable overflow on kernels, kernel exceptions, incorrect returned result, etc.

Mediating Host-Accelerator Interactions. Interaction and communication between accelerators and the host can pose severe security problems.Crossing Guard [40] is a coherence interface between the host and accelerators. It prevents potential bugs caused by host-accelerator communication. Border Control [41] is a sandboxing mechanism that guarantees that the memory access permissions are respected by accelerators, regardless of design errors or malicious intent. While the above work focuses on preventing bugs caused by host-accelerator interactions, HeteroFuzz on the other hand is a test generation tool to detect bugs when running host and hardware accelerator together.

8 CONCLUSION

As hardware specialization, energy efficiency, and flexible re-programmability are becoming increasingly important, a new type of *cloud-based hardware accelerator microservices* based on FPGA has emerged. Major service providers such as Amazon F1 and Microsoft Azure have begun to support heterogeneous application development to enable acceleration with customizable hardware.

HeteroFuzz makes three key contributions in automated testing of heterogeneous applications by incorporating multi-dimensional guidance, dynamic probabilistic mutations, and selective invocation. In total, the speed-up achieved by HeteroFuzz's three-pronged approach in finding the same set of errors is up to 754X, compared to using an AFL-like technique naively. HeteroFuzz is the first end-end technique that significantly improves testing effectiveness and efficiency for this new breed of heterogeneous applications.

ACKNOWLEDGMENTS

The participants of this research are in part supported by NSF grants CHS-1956322 CCF-1764077, CCF-1723773, ONR grant N00014-18-1-2037, and Intel CAPA grant.

REFERENCES

[1] Amazon.com. 2021. Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, David Evans, Tal Maklin, and Dongyan Xu (Eds.). Association for Computing Machinery (ACM), United States of America, 2329–2344. https://doi.org/10.1145/3133956.3134020 ACM Conference on Computer and Communications Security 2017<br/>, CCS 2017 ; Conference date: 30-10-2017 Through 03-11-2017.

[3] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer vision with the OpenCV library.* O'Reilly Media, Inc.

[4] Mary Branscombe. 2017. FPGAs and the New Era of Cloud-based Hardware Microservices. https://thenewstack.io/developers-fpgas-cloud/.

[5] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[6] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[7] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 725–741. https://doi.org/10.1109/SP.2015.50

[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy*. 725–741. https://doi.org/10.1109/SP.2015.50

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[10] Zhe Chen, Hugh T. Blair, and Jason Cong. 2019. LANMC: LSTM-Assisted Non-Rigid Motion Correction on FPGA for Calcium Image Stabilization. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 104–109. https://doi.org/10.1145/3289602.3293919

[11] Andrew A Chien, Allan Snavely, and Mark Gahagan. 2011. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science* 4 (2011), 1987–1996.

[12] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 4 (Feb. 2019), 20 pages. https://doi.org/10.1145/3294054

[13] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 2010. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 225–236. https://doi.org/10.1109/MICRO.2010.36

[14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2593069.2596667

[15] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. 2018. SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for DNA Sequencing. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 206–206. https://doi.org/10.1109/FCCM.2018.00040

[16] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[17] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. 2011. Customizable Domain-Specific Computing. *IEEE Design Test of Computers* 28, 2 (2011), 6–15. https://doi.org/10.1109/MDT.2010.141

[18] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (2004), 1025–1050.

[19] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Langlois. 2019. Module-per-Object: a Human-Driven Methodology for C++-based High-Level Synthesis Design. *arXiv preprint arXiv:1903.06693* (2019).

[20] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[21] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1261–1272. https://doi.org/10.1145/3377811.3380359

[22] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 127–135. https://doi.org/10.1109/FCCM.2019.00027

[23] Prabhat Gupta. 2021. Xeon+FPGA Platform for the Data Center. https://www.archive.ece.cmu.edu/~calcm/carl/lib/\exe/fetch.php?media=carl15-gupta.pdf.

[24] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems*. 1192–1195. https://doi.org/10.1109/ISCAS.2008.4541637

[25] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 277–287. https://doi.org/10.1145/3373087.3375310

[26] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. 2009. Replacing Testing with Formal Verification in Intel® CoreTM I7 Processor Execution Engine Validation. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 414–429.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[28] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127. https://doi.org/10.1109/ISCA.2016.20

[29] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 8 pages. https://doi.org/10.1145/3240765.3240842

[30] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. 2020. HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 493–505. https://doi.org/10.1145/3377811.3380340

[31] Xuan-Bach D. Le, Corina S. Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 14. https://doi.org/10.1145/3364452.3364455

[32] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[33] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485. https://doi.org/10.1145/3238147.3238176

[34] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[35] Ying-Dar Lin, Feng-Ze Liao, Shih-Kun Huang, and Yuan-Cheng Lai. 2015. Browser fuzzing by scheduled mutation and generation of document object models. In *2015 International Carnahan Conference on Security Technology (ICCST)*. 1–6. https://doi.org/10.1109/CCST.2015.7389677

[36] LLVM. 2021. https://llvm.org/.

[37] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

[38] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[39] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* PP (10 2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[40] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 163–176. https://doi.org/10.1145/3093337.3037715

[41] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 470–481. https://doi.org/10.1145/2830772.2830819

[42] OpenCV. 2021. https://opencv.org/.

[43] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering* (Hyderabad, India) *(FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 117–132. https://doi.org/10.1145/2593882.2593885

[44] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[45] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[46] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2016. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *Commun. ACM* 59, 11 (Oct. 2016), 114–122. https://doi.org/10.1145/2996868

[47] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[48] Xiaoke Qin and Prabhat Mishra. 2014. Scalable Test Generation by Interleaving Concrete and Symbolic Execution. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems (VLSID '14)*. IEEE Computer Society, USA, 104–109. https://doi.org/10.1109/VLSID.2014.25

[49] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[50] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*. 1102–1105. https://doi.org/10.1109/ASICON.2011.6157401

[51] 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. https://doi.org/10.14722/ndss.2016.23368

[52] David B. Thomas. 2016. Synthesisable recursion for C++ HLS tools. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 91–98. https://doi.org/10.1109/ASAP.2016.7760777

[53] Frank Tip. 1994. A survey of program slicing techniques. (1994).

[54] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

[55] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[56] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MEMLOCK: Memory Usage Guided Fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 765–777. https://doi.org/10.1145/3377811.3380396

[57] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Dynamic-memory-allocation-in-Vivado-HLS-and-segmentation-faults/td-p/894069.

[58] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Problems-with-simple-program-and-dataflow-directive/m-p/595225.

[59] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/HLS-design-problem-The-result-of-CSim-and-C-RTL-cosimulation-is/m-p/438446.

[60] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Fixed-point-arithmetic/m-p/754676.

[61] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/HLS-Synthesis-Difference-in-C-Simulation-and-C-RTL-Cosimulation/m-p/785019.

[62] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Issue-about-overwrite-in-a-loop-HLS-coding-style/m-p/907213.

[63] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/HLS-error-with-co-sim/m-p/1166264.

[64] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Simple-top-function-skipping-inner-loop-logic-entirely/m-p/1126600.

[65] Xilinx. 2021. https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/bd-p/hls.

[66] Xilinx. 2021. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[67] Zeping Xue and David B. Thomas. 2015. SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.1109/FPL.2015.7293959

[68] Zeping Xue and David B. Thomas. 2016. SynADT: Dynamic Data Structures in High Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 64–71. https://doi.org/10.1109/FCCM.2016.26

[69] Michał Zalewski. 2021. American Fuzz Loop. http://lcamtuf.coredump.cx/afl/.

[70] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 430–437. https://doi.org/10.1109/ICCAD.2017.8203809

[71] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. (2018), 269–278. https://doi.org/10.1145/3174243.3174255