Fuzzing High-Level Synthesis Tools

Zewei Du
Imperial College London, UK
Email: zewei.du19@imperial.ac.uk

Yann Herklotz
Imperial College London, UK
Email: yann.herklotz15@imperial.ac.uk

Nadesh Ramanathan
Imperial College London, UK
Email: n.ramanathan14@imperial.ac.uk

John Wickerson
Imperial College London, UK
Email: j.wickerson@imperial.ac.uk

***Abstract*—High-level synthesis (HLS) is becoming an increasingly important part of the computing landscape, even in safety-critical domains where correctness is key. As such, HLS tools are increasingly relied upon. In this paper, we investigate whether they are trustworthy.**

We have subjected three widely used HLS tools – LegUp, Xilinx Vivado HLS, and the Intel HLS Compiler – to a rigorous fuzzing campaign using thousands of random, valid C programs that we generated using a modified version of the Csmith tool. For each C program, we compiled it to a hardware design using the HLS tool under test and checked whether that hardware design generates the same output as an executable generated by the GCC compiler. When discrepancies arose between GCC and the HLS tool under test, we reduced the C program to a minimal example in order to zero in on the potential bug. Our testing campaign has revealed that all three HLS tools can be made either to crash or to generate wrong code when given valid C programs, and thereby underlines the need for these increasingly trusted tools to be more rigorously engineered. Out of 6700 test cases, we found 272 programs that failed in at least one tool, out of which we were able to identify at least 6 unique bugs.

I. Introduction

High-level synthesis (HLS), which refers to the automatic translation of software into hardware, is becoming an increasingly important part of the computing landscape. It promises to increase the productivity of hardware engineers by raising the abstraction level of their designs, and it promises software engineers the ability to produce application-specific hardware accelerators without having to understand hardware desciption languages (HDL) such as Verilog and VHDL. It is even being used in high-assurance settings, such as financial services [1], control systems [2], and real-time object detection [3]. As such, HLS tools are increasingly relied upon. In this paper, we investigate whether they are trustworthy.

The approach we take in this paper is *fuzzing*. This is an automated testing method in which randomly generated programs are given to compilers to test their robustness [4], [5], [6], [7], [8], [9]. The generated programs are typically large and rather complex, and they often combine language features in ways that are legal but counter-intuitive; hence they can be effective at exercising corner cases missed by human-designed test suites. Fuzzing has been used extensively to test conventional compilers; for example, Yang *et al.* [8] used it to reveal more than three hundred bugs in GCC and Clang . In this paper, we bring fuzzing to the HLS context.

```
1 unsigned int b = 0x1194D7FF;
2 int a[6] = {1, 1, 1, 1, 1, 1};
3
4 int main() {
5   for (int c = 0; c < 2; c++)
6     b = b >> a[c];
7   return b;
8 }
```

Figure 1. Miscompilation bug found in Xilinx Vivado HLS v2018.3, v2019.1 and v2019.2. The program returns `0x006535FF` but the correct result is `0x046535FF`.

An example of a compiler bug found by fuzzing

Figure 1 shows a program that produces the wrong result during RTL simulation in Xilinx Vivado HLS.[1] The bug was initially revealed by a randomly generated program of around 113 lines, which we were able to reduce to the minimal example shown in the figure. The program repeatedly shifts a large integer value `b` right by the values stored in array `a`. Vivado HLS returns `0x006535FF`, but the result returned by GCC (and subsequently confirmed manually to be the correct one) is `0x046535FF`.

The circumstances in which we found this bug illustrate some of the challenges in testing HLS tools. For instance, without the for-loop, the bug goes away. Moreover, the bug only appears if the shift values are accessed from an array. And – particularly curiously – even though the for-loop only has two iterations, the array `a` must have at least six elements; if it has fewer than six, the bug disappears. Even the seemingly random value of `b` could not be changed without masking the bug. It seems unlikely that a manually generated test program would bring together all of the components necessary for exposing this bug. In contrast, producing counter-intuitive, complex but valid C programs is the cornerstone of fuzzing tools. For this reason, we find it natural to adopt fuzzing for our HLS testing campaign.

[1]This program, like all the others in this paper, includes a `main` function, which means that it compiles straightforwardly with GCC. To compile it with an HLS tool, we rename `main` to `main_`, synthesise that function, and then add a new `main` function as a testbench that calls `main_`.

Our contribution

This paper reports on our campaign to test HLS tools by fuzzing.

- We use Csmith [8] to generate thousands of valid C programs from within the subset of the C language that is supported by all the HLS tools we test. We also augment each program with a random selection of HLS-specific directives.
- We give these programs to three widely used HLS tools: Xilinx Vivado HLS [10], LegUp HLS [11] and the Intel HLS Compiler, which is also known as i++ [12]. When we find a program that causes an HLS tool to crash, or to generate hardware that produces a different result from GCC, we reduce it to a minimal example with the help of the C-Reduce tool [13].
- Our testing campaign revealed that all three tools could be made to crash while compiling or to generate wrong RTL. In total, 6700 test cases were run through each tool out of which 272 test cases failed in at least one of the tools. Test case reduction was then performed on some of these failing test cases to obtain at least 6 unique failing test cases.
- To investigate whether HLS tools are getting more or less reliable over time, we also tested three different versions of Vivado HLS (v2018.3, v2019.1, and v2019.2). We found that in general there about half as many failures in versions v2019.1 and v2019.2 compared to v2018.3. However, there were also test-cases that only failed in versions v2019.1 and v2019.2, meaning bugs were probably introduced due to the addition of new features.

The overall aim of our paper is to raise awareness about the (un)reliability of current HLS tools, and to serve as a call-to-arms for investment in better-engineered tools. We hope that future work on developing more reliable HLS tools will find our empirical study a valuable source of motivation.

II. Related Work

The only other work of which we are aware on fuzzing HLS tools is that by Lidbury et al. [9], who tested several OpenCL compilers, including an HLS compiler from Altera (now Intel). They were only able to subject that compiler to superficial testing because so many of the test-cases they generated led to it crashing. In comparison to our work: where Lidbury et al. generated target-independent OpenCL programs that could be used to test HLS tools and conventional compilers alike, we specifically generate programs that are tailored for HLS (e.g. with HLS-specific pragmas) with the aim of testing the HLS tools more deeply. Another difference is that where we test using sequential C programs, they test using highly concurrent OpenCL programs, and thus have to go to great lengths to ensure that any discrepancies observed between compilers cannot be attributed to the inherent nondeterminism of concurrency.

Other stages of the FPGA toolchain have been subjected to fuzzing. Herklotz et al. [14] tested several FPGA synthesis tools using randomly generated Verilog programs. Where they concentrated on the RTL-to-netlist stage of hardware design, we focus our attention on the earlier C-to-RTL stage.

Several authors have taken steps toward more rigorously engineered HLS tools that may be more resilient to testing campaigns such as ours.

- The Handel-C compiler by Perna and Woodcock [15] has been mechanically proven correct, at least in part, using the HOL theorem prover. However, the tool does not support C as input directly, so is not amenable to fuzzing.
- Ramanathan et al. [16] proved their implementation of C atomic operations in LegUp correct up to a bound using model checking. However, our testing campaign is not applicable to their implementation because we do not generate concurrent C programs.
- In the SPARK HLS tool [17], some compiler passes, such as scheduling, are mechanically validated during compilation [18]. Unfortunately, this tool is not yet readily available to test properly.
- Finally, the Catapult C HLS tool [19] is designed only to produce an output netlist if it can mechanically prove it equivalent to the input program. It should therefore never produce wrong RTL. In future work, we intend to test Catapult C alongside Vivado HLS, LegUp, and Intel i++.

III. Method

This section describes how we conducted our testing campaign, the overall flow of which is shown in Figure 2. In §III-A, we describe how we configure Csmith to generate HLS-friendly random programs for our testing campaign. In §III-B, we discuss how we augment those random programs with directives and the necessary configuration files for HLS compilation. In §III-C, we discuss how we set up compilation and co-simulation checking for the three HLS tools under test. Finally, in §III-D, we discuss how we reduce problematic programs in order to obtain minimal examples of bugs.

A. Generating programs via Csmith

For our testing campaign, we require a random program generator that produces C programs that are both semantically valid and feature-diverse; Csmith [8] meets both these criteria. Csmith is designed to ensure that all the programs it generates are syntactically valid (i.e. there are no syntax errors), semantically valid (for instance: all variable are defined before use), and free from undefined behaviour (undefined behaviour indicates a programmer error, which means that the compiler is free to produce any output it likes, which renders the program useless as a test-case). Csmith programs are also deterministic, which means that their output is fixed at compile-time; this property is valuable for compiler testing because it means that if two different compilers produce programs that produce different results, we can deduce that one of the compilers must be wrong.

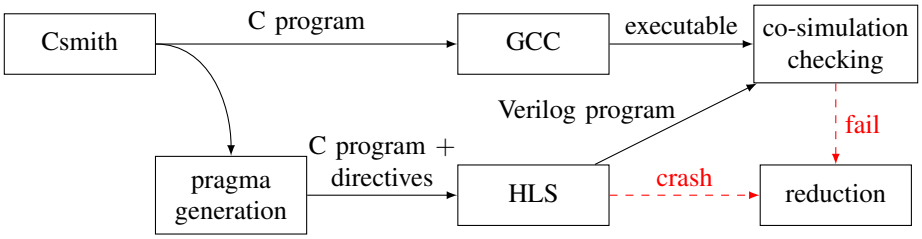

Figure 2. The overall flow of our approach to fuzzing HLS tools.

Property/Parameter	Change
`statement_ifelse_prob`	Increased
`statement_for_prob`	Reduced
`statement_arrayop_prob`	Reduced
`statement_break/goto/continue_prob`	Reduced
`float_as_ltype_prob`	Disabled
`pointer_as_ltype_prob`	Disabled
`union_as_ltype_prob`	Disabled
`more_struct_union_type_prob`	Disabled
`safe_ops_signed_prob`	Disabled
`binary_bit_and/or_prob`	Disabled
`--no-packed-struct`	Enabled
`--no-embedded-assigns`	Enabled
`--no-argc`	Enabled
`--max-funcs`	5
`--max-block-depth`	2
`--max-array-dim`	3
`--max-expr-complexity`	2

Table I
SUMMARY OF IMPORTANT CHANGES TO CSMITH'S FEATURE PROBABILITIES AND PARAMETERS TO GENERATE HLS-FRIENDLY PROGRAMS FOR OUR TESTING CAMPAIGN.

Additionally, Csmith allows users control over how it generates programs. For instance, the probabilities of choosing various C constructs can be tuned. This is vital for our work since we want to generate programs that are HLS-friendly.

Table I lists the main changes that we put in place to ensure that HLS tools are able to synthesise all of our generated programs. Our overarching aim is to make the programs tricky for the tools to handle correctly (in order to maximise our chances of exposing bugs), while keeping the synthesis and simulation times low (in order to maximise the rate at which tests can be run). To this end, we increase the probability of generating `if` statements in order to increase the number of control paths, but we reduce the probability of generating `for` loops and array operations since they generally increase run times but not hardware complexity. Relatedly, we reduce the probability of generating `break`, `goto`, `continue` and `return` statements, because with fewer `for` loops being generated, these statements tend to lead to uninteresting programs that simply exit prematurely.

More importantly, we disable the generation of several language features to enable HLS testing. First, we ensure that all mathematical expressions are safe and unsigned, to ensure no undefined behaviour. We also disallow assignments being embedded within expressions, since HLS generally does not support them. We eliminate any floating-point numbers since they typically involve external libraries or use of hard IPs on FPGAs, which in turn make it hard to reduce bugs to their minimal form. We also disable the generation of pointers for HLS testing, since pointer support in HLS tools is either absent or immature [10]. We disable the generation of unions as these were not supported by some of the tools such as LegUp 4.0.

To decide whether a problematic feature should be disabled or reported as a bug, the tool in question is taken into account. Unfortunately there is no standard subset of C that is supported by HLS tools; every tool chooses a slightly different subset. It is therefore important to choose the right subset in order to maximise the number of real bugs found in each tool, while avoiding generating code that the tool does not support. Therefore, we disable a feature if it fails gracefully (i.e. with an error message stating the issue) in one of the tools. If the HLS tool fails in a different way though, such as generating a wrong design or crashing during synthesis, the feature is kept in our test suite.

We enforce that the main function of each generated program must not have any input arguments to allow for HLS synthesis. We disable structure packing within Csmith since the "`#pragma pack(1)`" directive involved causes conflicts in HLS tools because it is interpreted as an unsupported pragma. We also disable bitwise AND and OR operations because when applied to constant operands, some versions of Vivado HLS errored out with 'Wrong pragma usage.'

Finally, we tweak several integer parameters that influence program generation. We limit the maximum number of functions (five) and array dimensions (three) in our random C programs, in order to reduce the design complexity and size. We also limit the depth of statements and expressions, to reduce the synthesis and simulation times.

B. Augmenting programs for HLS testing

We augment the programs generated by Csmith to prepare them for HLS testing. We do this in two ways: program instrumentation and directive injection. This involves either modifying the C program or accompanying the C program with a configuration file, typically a `tcl` file. Finally, we must also generate a tool-specific build script per program, which instructs the HLS tool to create a design project and perform the necessary steps to build and simulate the design.

a) Instrumenting the original C program: We generate a synthesisable testbench that executes the main function of the original C program. This top-level testbench contains a

custom XOR-based hash function that takes hashes of the program state at several points during execution, combines all these hashes together, and then returns this value. By making the program's output sensitive to the program state in this way, we maximise the likelihood of detecting bugs when they occur. Csmith-generated programs do already include their own hashing function, but we replaced this with a simple XOR-based hashing function because we found that the Csmith one led to infeasibly long synthesis times.

b) Injecting HLS directives: Directives are used to instruct HLS tools to optimise the resultant hardware to meet specific performance, power and area targets. Typically, a HLS tool identifies these directives and subjects the C code to customised optimisation passes. In order to test the robustness of these parts of the HLS tools, we randomly generated directives for each C program generated by Csmith. Some directives can be applied via a separate configuration file, others require us to add labels in the C program (e.g. to identify loops), and a few directives require placing pragmas at particular locations in a C program. We generate three classes of directives: those for loops, those for functions, and those for variables. For loops, we randomly generate directives including loop pipelining (with rewinding and flushing), loop unrolling, loop flattening, loop merging, loop tripcount, loop inlining, and expression balancing. For functions, we randomly generate directives including function pipelining, function-level loop merging, function inlining, and expression balancing. For variables, we randomly generate directives including array mapping, array partitioning and array reshaping.

C. Testing various HLS tools

Having generated HLS-friendly programs and automatically augmented them with directives and meaningful testbenches, we are now ready to provide them to HLS tools for testing. For each HLS tool in turn, we compile the C program to RTL and then simulate the RTL. Independently, we also compile the C program using GCC and execute it. Although each HLS tool has its own built-in C compiler that could be used to obtain the reference output, we prefer to obtain the reference output ourselves in order to minimise our reliance on the tool being tested.

To ensure that our testing is scalable for a large number of large, random programs, we also enforce several time-outs: we set a 5-minute time-out for C execution and a 2-hour time-out for C-to-RTL synthesis and RTL simulation. We do not count time-outs as bugs, but we record them.

D. Reducing buggy programs

Once we discover a program that crashes the HLS tool or whose C/RTL simulations do not match, we further scrutinise the program to identify the root cause(s) of the undesirable behaviour. As the programs generated by Csmith can be fairly large, we must systematically reduce these programs to identify the source of a bug.

Reduction is performed by iteratively removing some part of the original program and then providing the reduced program to the HLS tool for re-synthesis and co-simulation. The goal is to find the smallest program that still triggers the bug. We apply two consecutive methods of reduction in this work. The first step is to reduce the labels and pragmas that were added afterwards to make sure that these do not affect the behaviour of the program. These are reduced until there are no more declarations left or the bug does not get triggered anymore. We then use the C-Reduce tool [13] to automatically reduce the remaining C program. C-Reduce is an existing reducer for C and C++ and runs the reduction steps in parallel to converge as quickly as possible. It is effective because it reduces the input while preserving semantic validity and avoiding undefined behaviour. It has various reduction strategies, such as delta debugging passes and function inlining, that help it converge rapidly to a test-case that is small enough to understand and step through.

However, the downside of using C-Reduce with HLS tools is that we are not in control of which lines and features are prioritised for removal. As a consequence, we can easily end up with C-Reduce producing programs that are not synthesisable, despite being valid C. Even though C-Reduce does not normally introduce undefined behaviour, it can introduce behaviour that is unsupported in the HLS tools. An example is the reduction of a function call, where the reducer realises that a mismatch is still observed when the function call's arguments are removed, and the function pointer is assigned a constant instead. This is however often unsupported in HLS tools, since a function pointer does not have a concrete interpretation in hardware, because in the absence of instructions, functions are not associated with a particular memory location. Once unhandled behaviour is introduced at any point in the reduction, the test-cases will often zero in on that unhandled behaviour, even though it does not actually represent an interesting bug. To prevent this, we use a script to guide C-Reduce away from introducing these unhandled behaviours as much as possible. This script involves adding `-fsanitize=undefined` to the GCC options in order to abort when undefined behaviour is detected at runtime, and erroring out whenever any warning is encountered while running the HLS tool (except common warnings that are known to be harmless).

Tool	Unique Bugs
Xilinx Vivado HLS (all versions)	≥ 2
LegUp HLS	≥ 3
Intel i++	≥ 1

Table II
UNIQUE BUGS FOUND IN EACH TOOL. THE "$\geq$" SIGN SIGNIFIES A LOWER BOUND ON THE NUMBER OF UNIQUE BUGS FOUND AFTER TEST-CASE REDUCTION.

IV. Evaluation

We generate 6700 test-cases and provide them to three HLS tools: Vivado HLS, LegUp HLS and Intel i++. We use the same test-cases across all tools for fair comparison. We were able to test three different versions of Vivado HLS (v2018.3, v2019.1 and v2019.2). We tested one version of Intel i++

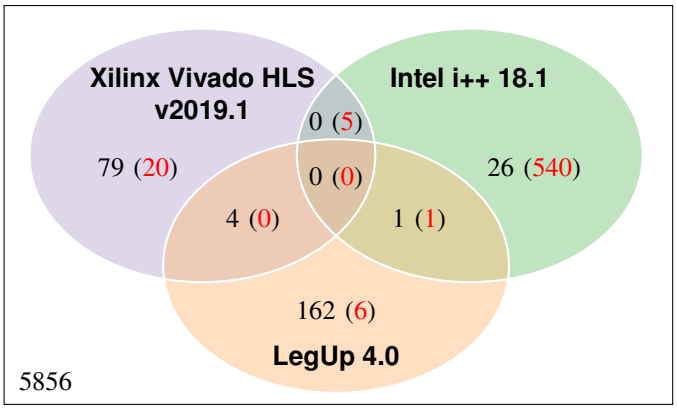

Figure 3. A Venn diagram showing the number of failures in each tool out of 6700 test-cases that were run. Overlapping regions mean that the test-cases failed in multiple tools. The numbers in parentheses represent the number of test-cases that timed out.

(version 18.1), and one version of LegUp (4.0). LegUp 7.5 is GUI-based and therefore we could not script our tests. However, we were able to manually reproduce bugs found in LegUp 4.0 in LegUp 7.5.

A. Results across different HLS tools

Figure 3 shows a Venn diagram of our results. We see that 167 (2.5%), 83 (1.2%) and 26 (0.4%) test-cases fail in LegUp, Vivado HLS and Intel i++ respectively. Despite i++ having the lowest failure rate, it has the highest time-out rate (540 test-cases), because of its remarkably long compilation time. Note that the absolute numbers here do not necessary correspond to the number of bugs in the tools, because a single bug in a language feature that appears frequently in our test suite could cause many programs to crash or fail. Hence, we reduce many of the failing test-cases to identify unique bugs, as summarised in Table II. We write '$\geq$' in the table to indicate that all the bug counts are lower bounds – we did not have time to go through the test-case reduction process for every failure.

B. Results across versions of an HLS tool

Besides comparing the reliability of different HLS tools, we also investigated the reliability of Vivado HLS over time. Figure 4 shows the results of giving 3645 test-cases to Vivado HLS v2018.3, v2019.1 and v2019.2. Test-cases that pass and fail in the same tools are grouped together into a ribbon. For instance, the topmost ribbon represents the 31 test-cases that fail in all three versions of Vivado HLS. Other ribbons can be seen weaving in and out; these indicate that bugs were fixed or reintroduced in the various versions. The diagram demonstrates that Vivado HLS v2018.3 contains the most failing test-cases compared to the other versions, having 62 test-cases fail in total. Interestingly, as an indicator of reliability of HLS tools, the blue ribbon shows that there are test-cases that fail in v2018.3, pass in v2019.1 but then fail again in v2019.2.

As in our Venn diagram, the absolute numbers in Figure 4 do not necessary correspond to the number of bugs. However, we can deduce from this diagram that there must be at least six

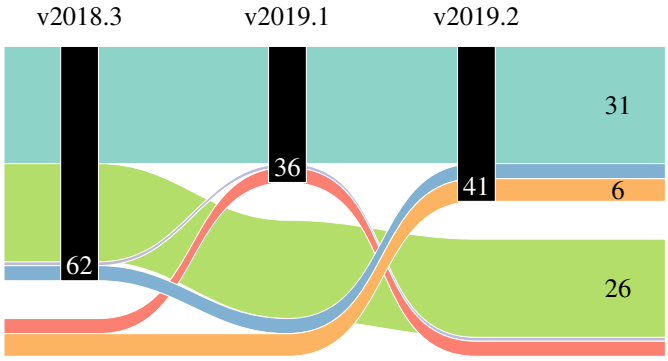

Figure 4. A Sankey diagram that tracks 3645 test-cases through three different versions of Vivado HLS. The ribbons collect the test-cases that pass and fail together. The black bars are labelled with the total number of test-case failures per version. The 3573 test-cases that pass in all three versions are not depicted.

```
1 int a[2][2][1] = {{{0},{1}},{{0},{0}}};
2
3 int main() {
4   a[0][1][0] = 1;
5 }
```

Figure 5. This program causes an assertion failure in LegUp HLS when `NO_INLINE` is set.

unique bugs in Vivado HLS, given that a ribbon must contain at least one unique bug. In addition to that, it can then be seen that Vivado HLS v2018.3 must have at least 4 individual bugs, of which two were fixed and two others stayed in Vivado HLS v2019.1. However, with the release of v2019.1, new bugs were introduced as well.

C. Some specific bugs found

This section describes some of the bugs that were found in the various tools that were tested. We describe two bugs in LegUp and one in Vivado HLS; in each case, the bug was first reduced automatically using C-Reduce, and then reduced further manually to achieve the minimal test-case. Although we did find test-case failures in Intel i++, the long compilation times for that tool meant that we did not have time to reduce any of the failures down to an example that is minimal enough to present here.

1) LegUp assertion error: The code shown in Figure 5 leads to an assertion error in LegUp 4.0 and 7.5 even though it should compile without any errors. An assertion error counts as a crash of the tool, as it means that an unexpected state was reached by this input. This shows that there is a bug in one of the compilation passes in LegUp, however, due to the assertion the bug is caught in the tool before it produces an incorrect design.

The buggy test-case has to do with initialisation and assignment to a three-dimensional array, for which the above piece of code is the minimal example. However, in addition to that it requires the `NO_INLINE` flag to be set, which disables function inlining. The code initialises the array with zeroes except for `a[0][1][0]`, which is set to one. Then the main

```
volatile int a = 0;
int b = 1;

int main() {
  int d = 1;
  if (d + a)
    b || 1;
  else
    b = 0;
  return b;
}
```

Figure 6. An output mismatch: LegUp HLS returns 0 but the correct result is 1.

function assigns one to that same location. This code on its own should not actually produce a result and should just terminate by returning 0, which is also what the design that LegUp generates does when the `NO_INLINE` flag is turned off.

2) LegUp miscompilation: The test-case in Figure 6 produces an incorrect Verilog in LegUp 4.0 and 7.5, which means that the results of RTL simulation is different to the C execution.

In the code above, `b` has value 1 when run in GCC, but has value 0 when run with LegUp. If the `volatile` keyword is removed from `a`, then the Verilog produces the correct result. As `a` and `d` are constants, the `if` statement should always produce go into the `true` branch, meaning `b` should never be set to 0. The `true` branch of the `if` statement only executes an expression which is not assigned to any variable, meaning the initial state of all variables should not change. However, LegUp HLS generates a design which enters the `else` branch instead and assigns `b` to be 0. The cause of this bug seems to be the use of `volatile` keyword, which interferes with the analysis that attempts to simplify the `if` statement.

3) Vivado HLS miscompilation: Figure 7 shows code that does not output the right result when compiled with all Vivado HLS versions. It returns `0x0` with Vivado HLS, instead of `0xF`. This test-case is much larger compared to the other test-cases that were reduced. We could not reduce this program any further, as everything in the code was necessary to trigger the bug.

The array `a` is initialised to all zeroes, as well as the other global variables `g` and `c`, so as to not introduce any undefined behaviour. However, `g` is also given the `volatile` keyword, which ensures that the variable is not optimised away. The function `d` then accumulates the values `b` that it is passed into a hash stored in `c`. Each `b` is eight bits wide, so function `e` calls the function seven times for some of the bits in the 64-bit value of `f` that it is passed. Finally, in the main function, the array is initialised partially with a `for` loop, after which the `e` function is called twice, once on the volatile function and once on a constant. Interestingly, the second function call with the constant is also necessary to trigger the bug.

4) Intel i++ miscompilation:

```
volatile unsigned int g = 0;
int a[256] = {0};
int c = 0;

void d(char b) {
  c = (c & 4095) ^ a[(c ^ b) & 15];
}

void e(long f) {
  d(f); d(f >> 8); d(f >> 16); d(f >> 24);
  d(f >> 32); d(f >> 40); d(f >> 48);
}

int main() {
  for (int i = 0; i < 56; i++)
    a[i] = i;
  e(g);
  e(-2L);
  return c;
}
```

Figure 7. An output mismatch: Vivado HLS returns `0x0` but the correct result is `0xF`.

V. Conclusion

We have shown how existing fuzzing tools can be modified so that their outputs are compatible with HLS tools. We have used this testing framework to run 6,700 test-cases through three different HLS tools, and 3,645 test-cases through three different version of Vivado HLS to show how bugs are fixed and introduced. In total, we found at least 6 unique bugs in all the tools. These bugs include crashes as well as instances of generated designs not behaving in the same way as the original code.

One can always question how much bugs found by fuzzers really *matter*, given that they are usually found by combining language features in ways that are vanishingly unlikely to happen 'in the wild' [20]. This question is especially pertinent for our particular context of HLS tools, which are well-known to have restrictions on the language features that they handle. Nevertheless, we would argue that although the *test-cases* we generated do not resemble the programs that humans write, the *bugs* that we exposed using those test-cases are real, and could also be exposed by realistic programs. Moreover, it is worth noting that HLS tools not exclusively provided with human-written programs to compile: they are often fed programs that have been automatically generated by another compiler. Ultimately, we believe that any errors in an HLS tool are worth identifying because they have the potential to cause problems, either now or in the future. And when HLS tools *do* go wrong (or indeed any sort of compiler for that matter), it is particularly infuriating for end-users because it is so difficult to identify whether the fault lies with the tool or with the program it has been given to compile.

Further work could be done on supporting more HLS tools, especially ones that claim to prove that their output is correct

before terminating. This could give an indication on how effective these proofs are, and how often they are actually able to complete their equivalence proofs during compilation in a feasible timescale.

Conventional compilers have become quite resilient to fuzzing over the last decade, so recent work on fuzzing compilers has had to employ increasingly imaginative techniques to keep finding new bugs [21]. In comparison, we have found that HLS tools – at least, as they currently stand – can be made to exhibit bugs even using the relatively basic fuzzing techniques that we employed in this project.

As HLS is becoming increasingly relied upon, it is important to make sure that HLS tools are also reliable. We hope that this work further motivates the need for rigorous engineering of HLS tools, either by validating that each output the tool produces is correct or by proving the HLS tool itself correct once and for all.

REFERENCES

[1] EE Journal, "Silexica expands into fintech industry bringing next-generation compute acceleration," Press release, June 2020. [Online]. Available: https://bit.ly/hls-fintech

[2] LegUp Computing, "Migrating motor controller C++ software from a microcontroller to a PolarFire FPGA with LegUp high-level synthesis," White Paper, June 2020. [Online]. Available: https://bit.ly/hls-controller

[3] PR Newswire, "Mentor's Catapult HLS enables Chips&Media to deliver deep learning hardware accelerator IP in half the time," Press release, January 2019. [Online]. Available: https://bit.ly/hls-objdetect

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208.

[5] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[6] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[7] C. Zhang, T. Su, Y. Yan, F. Zhang, G. Pu, and Z. Su, "Finding and understanding bugs in software model checkers," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 763–773.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294.

[9] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 65–76.

[10] Xilinx, "Vivado high-level synthesis," 2020. [Online]. Available: https://bit.ly/39ereMx

[11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, 9 2013.

[12] Intel, "SDK for OpenCL applications," 2020. [Online]. Available: https://intel.ly/30sYHz0

[13] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[14] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in FPGA synthesis tools," in *FPGA*. ACM, 2020, pp. 277–287.

[15] J. Perna and J. Woodcock, "Mechanised wire-wise verification of Handel-C synthesis," *Science of Computer Programming*, vol. 77, no. 4, pp. 424 – 443, 2012.

[16] N. Ramanathan, S. T. Fleming, J. Wickerson, and G. A. Constantinides, "Hardware synthesis of weakly consistent C concurrency," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, J. W. Greene and J. H. Anderson, Eds. ACM, 2017, pp. 169–178. [Online]. Available: http://dl.acm.org/citation.cfm?id=3021733

[17] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.*, Jan 2003, pp. 461–466.

[18] R. Chouksey and C. Karfa, "Verification of scheduling of conditional behaviors in high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2020. [Online]. Available: https://doi.org/10.1109/TVLSI.2020.2978242

[19] Mentor, "Catapult high-level synthesis," 2020. [Online]. Available: https://bit.ly/32xhADw

[20] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: how much does it matter?" *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 155:1–155:29, 2019.

[21] K. Even-Mendoza, C. Cadar, and A. Donaldson, "Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour," in *IEEE/ACM International Conference on Automated Software Engineering, New Ideas and Emerging Results Track (ASE-NIER 2020)*, 09 2020.