An Empirical Study of the Reliability of High-Level Synthesis Tools

Blind review

***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. But are they trustworthy?**

We have subjected four widely used HLS tools – LegUp, Xilinx Vivado HLS, the Intel HLS Compiler and Bambu – 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 four 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 1178 programs that failed in at least one tool, out of which we were able to discern at least 8 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 hardware engineers an increase in productivity 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 description languages (HDL) such as Verilog and VHDL. HLS is being used in an ever greater range of domains, including such high-assurance settings as financial services [1], control systems [2], and real-time object detection [3]. As such, HLS tools are increasingly relied upon, even though "high-level synthesis research and development is inherently prone to introducing bugs or regressions in the final circuit functionality" [4, Section 3.4.6]. In this paper, we investigate whether they are trustworthy and give an empirical evaluation of their reliability.

The approach we take is *fuzzing*. This is an automated testing method in which randomly generated programs are given to compilers to test their robustness [5], [6], [7], [8], [9], [10]. 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.* [9] used it to reveal more than three hundred bugs in GCC and LLVM. In this paper, we bring fuzzing to the HLS context.

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

Figure 1. Miscompilation bug in Xilinx Vivado HLS. The generated RTL returns `0x006535FF` but the correct result is `0x046535FF`.

Example 1 (A miscompilation bug in Vivado HLS)**.** Figure 1 shows a program that produces the wrong result during RTL simulation in Xilinx Vivado HLS v2018.3, v2019.1 and v2019.2.[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. This bug was also reported to Xilinx and confirmed to be a bug.[2]The program repeatedly shifts a large integer value `x` right by the values stored in array `arr`. Vivado HLS returns `0x006535FF`, but the result returned by GCC (and subsequently confirmed manually to be the correct one) is `0x046535FF`.

The example above demonstrates the effectiveness of fuzzing. It seems unlikely that a human-written test-suite would discover this particular bug, given that it requires several components all to coincide before the bug is revealed!

Yet this example also begs the question: do 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 real world' [11]. 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, 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*. 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 problems caused by HLS tools going wrong (or indeed any sort of compiler for that matter) are particularly egregious, because it is so difficult for end-users

[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 `result`, synthesise that function, and then add a new `main` function as a testbench that calls `result`.

[2]Link to Xilinx bug report redacted for review.

to identify whether the fault lies with their design or the HLS tool.

A. Our approach and results

Our approach to fuzzing HLS tools comprises three steps. First, we use Csmith [9] 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. Second, we give these programs to four widely used HLS tools: Xilinx Vivado HLS [12], LegUp HLS [13], the Intel HLS Compiler, which is also known as i++ [14] and finally Bambu [15]. Third, if 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 [16].

Our testing campaign revealed that all four tools could be made to generate an incorrect design. In total, 6700 test cases were run through each tool out of which 1178 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 8 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 far fewer failures in versions v2019.1 and v2019.2 compared to v2018.3, but we also identified a few test-cases that only failed in versions v2019.1 and v2019.2, which suggests that some new features may have introduced bugs.

In summary, the overall aim of our paper is to raise awareness about the reliability (or lack thereof) 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. [10], 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 and only including supported constructs) 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. [17] 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.

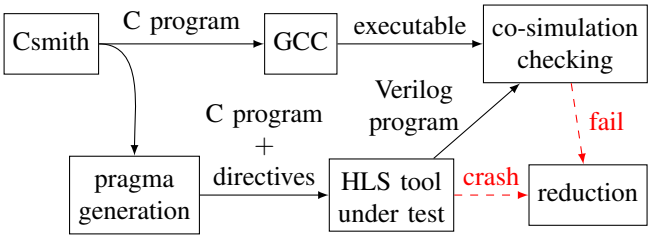

Figure 2. The overall flow of our approach to fuzzing HLS tools.

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 [18] 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. [19] 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 [20], some compiler passes, such as scheduling, are mechanically validated during compilation [21]; unfortunately, this tool is no longer available. Finally, the Catapult C HLS tool [22] 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

The overall flow of our testing approach is shown in Figure 2. This section describes how test-cases are generated (§III-A), executed (§III-B), and reduced (§III-C).

A. Generating test-cases

Csmith exposes several parameters through which the user can adjust how often various C constructs appear in the randomly generated programs. Table I describes how we configured these parameters. 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). For instance, we increase the probability of generating `if` statements so as 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. We disable various features that are not supported by HLS tools such as assignments inside expressions, pointers, and union types. We avoid floating-point numbers since they typically involve external libraries or use of hard IPs on FPGAs, which make it hard to reduce bugs to a minimal form.

Table I
SUMMARY OF CHANGES TO CSMITH'S PROBABILITIES AND PARAMETERS.

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

To prepare the programs generated by Csmith for HLS testing, we modify them in two ways. First, we inject random HLS directives, which instruct the HLS tool to perform certain optimisations, including: loop pipelining, loop unrolling, loop flattening, loop merging, expression balancing, function pipelining, function-level loop merging, function inlining, array mapping, array partitioning, and array reshaping. Some directives can be applied via a separate configuration file (.tcl), some require us to add labels to the C program (e.g. to identify loops), and some require placing pragmas at particular locations in the C program.

The second modification we make has to do with the top-level function. Each program generated by Csmith ends its execution by printing a hash of all its variables' values, in the hope that any miscompilations will be exposed through this hash value. We found that Csmith's built-in hash function led to infeasibly long synthesis times, so we replaced it with our own simple XOR-based one.

Finally, we generate a synthesisable testbench that executes the main function of the original C program, and a tool-specific script that instructs the HLS tool to create a design project and then build and simulate the design.

B. Compiling the test-cases using the HLS tools

For each HLS tool in turn, we compile the C program to RTL and then simulate the RTL. 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 that is being tested.

To ensure that our testing scales to a large number of large 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 do record them.

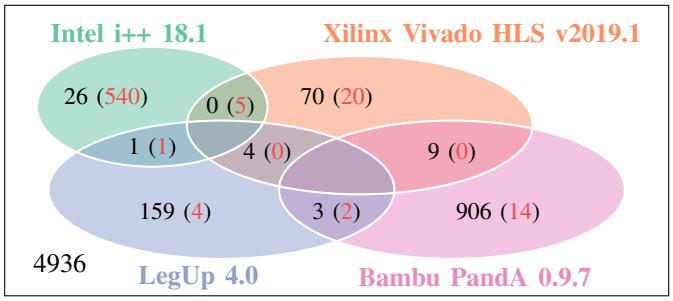

Figure 3. The number of failures per tool out of 6700 test-cases. Overlapping regions mean that the same test-cases failed in multiple tools. The numbers in parentheses report how many test-cases timed out.

C. Reducing buggy programs

Once we discover a program that crashes the HLS tool or whose C/RTL simulations do not match, we systematically reduce it to its minimal form using the C-Reduce tool [16], in the hope of identifying the root cause. This is done by successively removing or simplifying parts of the program, checking that the bug remains at each step.

We also check at each stage of the reduction process that the reduced program remains within the subset of the language that is supported by the HLS tools; without this check, C-Reduce only zeroed in on programs that were outside of this subset and hence did not represent real bugs.

IV. EVALUATION

We generate 6700 test-cases and provide them to four HLS tools: Vivado HLS, LegUp HLS, Intel i++ and Bambu. We use the same test-cases across all tools for fair comparison (except the HLS directives, which have tool-specific syntax). 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++ (version 18.1), LegUp (4.0) and Bambu (v0.9.7). LegUp 7.5 is GUI-based and therefore we could not script our tests. However, we were able to manually reproduce all the 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 918 (13.7%), 167 (2.5%), 83 (1.2%) and 26 (0.4%) test-cases fail in Bambu, 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 necessarily 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 in an effort to identify unique bugs; these are summarised in the table below.

We write '$\geq$' above to emphasise that all the bug counts are lower bounds – we did not have time to go through the rather arduous test-case reduction process for every failure.

Tool	Unique Bugs
Xilinx Vivado HLS v2019.1	≥ 2
LegUp HLS	≥ 3
Intel i++	≥ 1
Bambu HLS	≥ 2

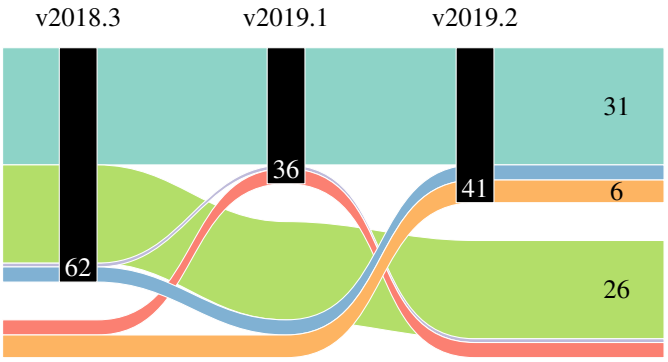

Figure 4. A Sankey diagram that tracks 3645 test-cases through three different versions of Vivado HLS. The ribbons collect the test-cases that pass and fail together. The black bars are labelled with the total number of test-case failures per version. The 3573 test-cases that pass in all three versions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

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. We see that Vivado HLS v2018.3 had the most test-case failures (62). 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 here do not necessary correspond to the number of actual bugs, but we can deduce that there must be at least six unique bugs in Vivado HLS, given that each ribbon corresponds to at least one unique bug.

C. Some specific bugs found

We now describe three more of the bugs we found: one crash bug in LegUp, and a miscompilation in Intel and Bambu respectively. As in Example 1, each bug was first reduced automatically using C-Reduce, and then reduced further manually to achieve the minimal test-case.

Example 2 (A crash bug in LegUp)**.** The program shown below leads to an internal compiler error (an unhandled assertion in this case) in LegUp 4.0 and 7.5.

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

It initialises a 3D array with zeroes, and then assigns to one element. The bug only appears when function inlining is disabled (`NO_INLINE`).

```
static volatile int a[9][1][7];
int main() {
  int tmp = 1;
  for (int b = 0; b < 2; b++) {
    a[0][0][0] = 3;
    a[0][0][0] = a[0][0][0];
  }
  for (int i = 0; i < 9; i++)
    for (int k = 0; k < 7; k++)
      tmp ^= a[i][0][k];
  return tmp;
}
```

Figure 5. Miscompilation bug in Intel i++. It should return 2 because `3 ^ 1 = 2`, however, Intel i++ returns 0 instead.

```
static int b = 0x10000;
static volatile short a = 0;

int main() {
  a++;
  b = (b >> 8) & 0x100;
  return b;
}
```

Figure 6. Miscompilation bug in Bambu. As the value of `b` is shifted to the right by 8, the output should be `0x100`. However, Bambu outputs 0.

Example 3 (A miscompilation bug in Intel i++)**.** Figure 5 shows a miscompilation bug that was found in Intel i++. Intel i++ does not seem to notice the assignment to 3 in the previous for loop, or tries to perform some optimisations that seem to analyse the array incorrectly and therefore results in a wrong value being returned.

Example 4 (A miscompilation bug in Bambu)**.** Figure 6 shows a miscompilation bug in Bambu, where the result of the value in `b` is affected by the increment operation on `a`.

V. Conclusion

We have shown how an existing fuzzing tool can be modified so that its output is suitable for HLS, and then used it in a campaign to test the reliability of three modern HLS tools. In total, we found at least 8 unique bugs across all the tools, including crashes and miscompilation bugs.

Further work could be done on supporting more HLS tools, especially ones that claim to prove that their output is correct before terminating, such as Catapult-C [22].

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 [23]. 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, whether that is 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] A. C. Canis, "LegUp: open-source high-level synthesis research framework," Ph.D. dissertation, University of Toronto, 2015.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4.

[16] 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.

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

[18] 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.

[19] 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

[20] 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.

[21] 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

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

[23] 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.