Using CSmith to Fuzz HLS Compilers

Jason Panelli
panelli@ucla.edu
University of California - Los Angeles

William O'Brien
wobrien@ucla.edu
University of California - Los Angeles

Samuel Lee
samuellee@cs.ucla.edu
University of California - Los Angeles

Emily Kao
emilykao@ucla.edu
University of California - Los Angeles

Abstract

High-level synthesis (HLS) compilers are tools that create register transfer level code for FPGAs using C++/C code. HLS compilers are growing in popularity as they enable hardware programmers to quickly write production-quality code for FPGAs, which leads to a more efficient development process, more control over optimization of designs, and the ability to build designs at a higher level of abstraction than RTL/Verilog. One effective method of testing that is increasingly being used for compilers is the generation of random programs to potentially trigger incorrect errors from the compiler. While there are several fuzzers that have been written to test C compilers with randomly generated C programs, they are unfortunately incompatible with HLS compilers. HLS compilers are able to take in C programs as input, but do not support the full capabilities of the C language. In theory, these random generation tools can be modified to generate C programs with the same restrictions of HLS tools. Additionally, HLS tools typically support additional features such as pragmas, directives, and more.

In particular, we research and extend Csmith - a random generator of C programs open sourced in 2009. We study the steps and experiment with the work required to modify Csmith to adapt it for HLS tools. In order to efficiently create a working solution for HLS fuzzing, we extensively research unsupported C99 features and HLS specific features. We also discuss the methodology and issues encountered in testing several different HLS compilers during our experiment in extending C generators to support HLS.

1 Introduction

In the current computational landscape, high-level synthesis (HLS) tools are becoming increasingly important in the development of FPGAs. HLS tools such as the Intel HLS Compiler, Xilinx Vivado HLS, LegUp, and Bambu take in high-level languages like C or C++ to create register transfer level (RTL) models. This is very useful in FPGA development because developing models in high-level languages is orders of magnitude faster than developing models over RTL. However, as with any software, it is possible that these HLS tools contain bugs, therefore it is very important that we thoroughly test these tools to ensure that the RTL models they generate are correct.

One way to test HLS tools is through fuzzing, specifically, randomly generating valid HLS-compatible C programs then compiling them with HLS compilers to check for bugs. Bugs can either be detected through crashes or by comparing the output of multiple compilers to reveal mis-compilations. An existing tool that randomly generates C programs is Csmith [1]. Csmith uses a grammar-based fuzzing technique, where it utilizes a grammar of a subset of C to randomly generate C programs. A test driver will then feed the randomly generated programs to multiple compilers and compare the output. Csmith exploits the idea that if all compilers follow the same specifications, all deterministic inputs should produce the same output. Therefore, if one compiler generates a different output than the others, a bug has been exposed in that one compiler [12]. The creators of Csmith were able to discover many bugs, mostly related to incorrect compiler optimizations, in popular C compilers such as GCC and LLVM.

Noting the success of Csmith in catching bugs in C compilers, we investigate if we can adapt Csmith to generate HLS-compatible code and use Csmith to catch bugs in HLS tools. This approach has been used before in [3] and was successful in identifying several bugs in popular HLS compilers. However, the current pathway is to use Csmith to randomly generate C programs and then inject various HLS directives later. Our goal is to modify Csmith to be able to inject HLS directives as it randomly generates C programs, thereby building a version of Csmith that can natively fuzz HLS compilers. Having a single program that generates HLS-compatible code including HLS-specific directives will simplify HLS compiler fuzzing.

In order to narrow the scope of our research, we decided to focus on adapting Csmith to be compatible with the Intel HLS Compiler. The Intel HLS Compiler (on the commandline we use i++ and dpcpp to run the HLS Compiler) was chosen because it is one of the most popular and well-documented HLS tools, the license was free and Intel provides a free environment called Intel DevCloud with all of Intel's software already installed. In addition to adapting Csmith to be HLS compatible, we also explored injecting HLS directives, which are pragmas that instruct the compiler to do certain optimizations. We believe that injecting HLS directives are important in HLS tool testing because as seen in Csmith, many compiler bugs are actually related to optimization, so it is likely that the same issue would exist in HLS tools. Though we focus on a single HLS tool, for future extensions of this research, we would expand this tool to be able to test multiple HLS tools, which would allow us to compare their output and potentially reveal bugs. In summary, our research will focus on modifying Csmith to generate HLS compatible programs and injecting various HLS directives throughout the randomly generated programs.

2 Related Work

In this section we detail work related to HLS tool testing.

2.1 Random Program Generation for Compiler Testing

Testing compilers and programming language interpreters with grammar-based program generation tools has proven to be a highly effective strategy. In fact, compiler testing has been one of the most important applications of grammar-based fuzzing [14]. Csmith, which we extend in this paper, is one of the most prominent tools, and as mentioned, it has identified numerous (over 400) bugs in popular C compilers [13]. It was developed at the University of Utah by Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Csmith was built to enable differential testing of compilers, so it only generates C programs that are well-formed and deterministic so that there is only one correct compilation of a Csmith-generated program [12]. Another notable idea from Csmith is to have each generated program compute and print out a checksum of all its global variables, to make it easier to identify mis-compilations [12]. Csmith was a natural choice for our starting point because it is recent and well-known and its past success on standard C compilers indicates it may also be very effective on HLS C compilers.

Csmith was not the first or last random C program generator. Previous tools include DDT (1998), which was the first to apply differential testing, but did not avoid undefined behavior like Csmith does, which limits its usefulness for automated fuzzing [12]. Lindig (2005) [6] and Sheridan (2007) [10] also created random C program generators, but theirs could not generate as expressive programs as Csmith, which limited their ability to trigger compiler bugs. A more recent tool is YARPGen [7], developed by the University of Utah and Intel and published in 2020. It has identified over 200 bugs in popular C/C++ compilers as of the publication of the paper. It is meant to improve upon Csmith in a few ways, such as by avoiding the use of wrapper functions for dynamic safety checks of generated arithmetic, which the authors considered too heavy-handed. The main difference from Csmith is that it implements generation policies to systematically mutate the code to try to expose as many bugs as possible. However, it does not support all the C features that Csmith supports, partially due to YARPGen simply being a newer tool. We considered extending YARPGen instead of Csmith for HLS compiler testing, but decided on Csmith because it has a longer track record of identifying compiler bugs and because of the C features YARPGen does not implement.

Another recent and successful approach to C compiler testing is the EMI Project [5], from the University of California, Davis in 2014, which uses grammar-based test case mutation. It profiles test programs as they execute, then prunes parts of the program that were not executed. The idea is that, unless there was a compiler error, running all the modified variants of the program should produce the same result, because only unexecuted parts of the program were removed. This approach is better at detecting miscompilations than Csmith, which is better at detecting compiler crashes. Therefore, extending this approach to HLS compiler testing could be a useful complementary work, but it may be more difficult to profile FPGA simulations.

2.2 Adapting Program Generation to HLS Compilers

Our work is not the first to adapt random program generation to HLS C compilers. The only previous work in this area that we are aware of was done by Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson at Imperial College London in 2021 [1]. Their approach was to generate many C programs using an unmodified Csmith, with the incompatible C constructs disabled, then pass the programs through a script to create a component function and add some HLS-specific directives. Specifically, they added three loop pragmas: "loop_coalesce," "unroll," and "max_concurrency." To identify failures, they simulated the compiled RTL and ran a version compiled with GCC and detected crashes and mismatches in the output checksum. Their work identified 17 to 900 failures per compiler, including 27 in the Intel HLS Compiler. This gives us confidence that our approach will also expose new compiler errors.

Our work replicated their approach of disabling C constructs by providing a probabilities file to Csmith. We used the probabilities file they generated as the starting point for our experimentation on disabling various C constructs. The key difference in our approach is that we extended Csmith to create the component function and add HLS directives during program generation. We believe a unified tool will enable simpler HLS fuzzing than a combination of a program generator and a processing script. Our approach may also be more extensible to different HLS compilers, because we believe it is simpler to modify Csmith to add HLS-specific C features than to create a separate program to parse the Csmith-generated programs and insert those features. Compared to the previous project, we also added more HLS compiler directives, including annotations on static variables and more loop pragmas, which will allow us to test more HLS-specific compiler features and likely trigger more bugs. One advantage of their approach, however, is that the same Csmith-generated program can be processed by different compiler-specific scripts so that test cases can be more easily reused across compilers with different input requirements.

2.3 Other HLS Compiler Testing Techniques

There is very limited information publicly available about how commercial HLS compilers are tested. However, there have been a few attempts to make the HLS compiler testing process more rigorous. Some compilers have been partially proven correct using theorem provers, such as Handel-C [8], or model checking, like LegUp [9]. Some compilers go even further and only generate RTL code they can prove is correct. For example, the Catapult C compiler proves that its outputs are equivalent to the input design [11]. However, it is slow and its validation tool has not been proven correct, preventing a complete correctness guarantee [2]. A very recent HLS compiler called Vericert [2] from the same group at Imperial College London has been fully mechanically verified using a theorem prover. Its main downside is that the designs it generates are many times slower than those generated by existing tools, but the group claims this is mainly due to its current lack of optimizations.

3 Methodology

Our methodology for adapting CSmith consisted of two major components. First, we must disable the features of C99 that Intel HLS Compiler does not support. Next, we add features that are exclusive to Intel HLS Compiler to allow us to more rigorously test it.

3.1 Blocking Unsupported C99 Features

As a first step, we aim to identify the features of C99 that are prohibited in Intel's HLS compilers, namely some memory addressing operations and some high-level language concepts. From Intel's documentation [4], we specifically identify the following features that are not supported: dynamic memory allocation, virtual functions, function pointers, C/C++ library functions except explicitly stated support for some math functions, non-static class functions, and template functions without specific specialization. Fortunately, CSmith generates C99 code by applying probabilities to many C99 grammar rules that can be changed to disable features or make them more prominent. This feature is typically useful for targeting specific features with higher probability and fuzzing in more probabilistic manners, but we can exploit it to turn off features of C99 that are not supported by HLS. Table 1 shows the features that we disabled and why we disabled them. In addition to the explicitly unsupported C99 features, we also disable structs and unions because the Intel HLS compilers do not accept the default assignment operator for these types, which CSmith uses heavily.

As a result, disabling the necessary features is as simple as removing them via the flags and probability file. Resolving which flags and probabilities are necessary was an iterative process that required us to turn flags and probabilities on and off and analyze the errors that are produced until we reach a subset of features that nullify any errors. There are still iterations to be performed on this set, however, as the current probabilities and flags disable features that Intel HLS compilers can support but require additional effort. This notably includes floats, structs, and unions.

3.2 Adding HLS Specific Features

While reducing the C99 grammar allows us to compile CSmith produced code, it is still not a sufficient representation of the code that the HLS compiler needs to accept. The compiler accepts some additional pragmas and special notations in order to map high level C code to RTL in more sensible manners. While there are a number of features that could be added, we first focus on adding just a few to the CSmith grammar.

3.2.1 Component Functions.

Development on FPGAs normally includes abstractions that break up the code into blocks. These blocks are called components and can be thought of as smaller subcircuits that a developer can treat similarly to a function in a higher level language. Essentially, components have wired input and output that can be connected to other parts of the circuit to abstract different operations.

It makes sense to support this in an HLS compiler because it can lead to RTL code that maps more closely to what an FPGA developer might actually produce and leads to circuits that are more logical to humans. It can also help with writing a testbench in the RTL that can test individual components instead of testing the program as a whole. In order to support this, the Intel HLS compiler uses the keyword 'component' that is used directly before a function and its definition. Functions marked with the component keyword map directly to FPGA components.

Our first method for adding "component" annotations to C programs was to randomly append the annotation before function definitions. To add this to the CSmith grammar, we made changes to the CSmith Function class, Probability class, and CGOptions class. The CGOptions and Probability classes allow us to add command line options and probability file arguments that can control how often component functions are generated. The main grammar change occurs in the Function class, where we add the 'component' keyword before a function definition if the probabilities and flags indicate that we should.

The above method has a major limitation because it only works when compiling the code for software emulation; it does not work when compiling for an FPGA due to restrictions on how components can interact. In particular, when compiling for an FPGA, at least one function must be marked as a component. Because of these restrictions, our initial method of randomly annotating functions as components

CSmith Probability Definition	Feature Corresponding to this Probability	Reason for Disabling
more_struct_union_type_prob	probability of having more types in the program	We disable structs and unions entirely
bitfields_signed_prob	probability of signed bitfield	We disable structs and unions entirely
bitfield_in_normal_struct_prob	probability of bitfield in a struct	We disable structs and unions entirely
scalar_field_in_full_bitfields_struct_prob	probability of the whole struct using a bitfield	We disable structs and unions entirely
safe_ops_signed_prob		
select_deref_pointer_prob	probability for using a dereferenced pointer	Pointers are not supported
stricter_const_prob		
looser_const_prob		
field_const_prob		
std_unary_func_prob		
pointer_as_ltype_prob	probability to create a pointer variable	Pointers are not supported
struct_as_ltype_prob	probability to create a struct variable	We disable structs and unions entirely
union_as_ltype_prob	probability to create a union variable	We disable structs and unions entirely
float_as_ltype_prob	probability to create a float variable	Floats require special HLS libraries for support
unary_minus_prob		
unary_bit_not_prob		
binary_bit_and_prob		
binary_bit_or_prob		
void_prob		void types produce casting errors
float_prob	probability of using floats	Floats require special HLS libraries for support
–no-argc	no arguments in main function	main function arguments will just be ignored anyway
–max-array-dim 3	limits dimensions of arrays	FPGAs have limited memory
–max-funcs 5	limits number of functions in program	FPGAs are limited in how large the programs can be
–max-expr-complexity 2	limits nested complexity of expressions	Reduces wiring of FPGAs and code complexity
–no-float	eliminates floats through a flag	Floats require special HLS libraries for support
–no-embedded-assigns		
–max-block-depth 2		
–no-unions	Eliminates unions through a flag	We disable structs and unions entirely
–no-packed-struct	Eliminates packed structs through a flag	We disable structs and unions entirely
–no-const-pointers	Eliminates const pointers through a flag	Pointers are not supported
–no-pointers	Eliminates pointers through a flag	Pointers are not supported
–strict-const-arrays		

Table 1. Flags and probabilities needed to disable different C99 features

was not sophisticated enough to generate programs to test the HLS compiler's ability to generate Verilog programs.

Therefore, to test compilation for FPGA, we extended Csmith to deterministically add one component function that encapsulates all the randomly-generated logic. We added a new "HLS mode" to Csmith that adds this component function to its test programs. Previously, Csmith would deterministically generate a main function which always calls a function called "func_1," which is the entry point to all the random logic in the test program. After func_1 and all the functions it invokes have completed, the main function computes a checksum on the global variables and outputs the result for the user. We modified the part of Csmith that generates the test program's main function so that the main function is treated as a testbench when using HLS mode. Instead of calling func_1, when in HLS mode, the main function calls the component function (called componentFunc), which in turn calls func_1. That way, all the randomly-generated logic is encapsulated in the component, while the main function is responsible for printing the result for the user. This approach eliminated all the errors associated with the "component" annotation when compiling for an FPGA. These changes were made in Csmith's OutputMgr class, and they are invoked with a new command-line option "–hls-mode."

3.3 For Loop Pragmas

Intel's HLS Compiler provides various pragmas that instruct the compiler on how to pipeline the loops in our components. In order to modify Csmith to be able to thoroughly test an HLS compiler, pragmas are very important to include because all the different ways the Intel HLS compiler pipeline loops should be tested. From Intel's list of loop pragmas, we implement the following: disable_loop_pipelining, ii, ivdep, loop_coalesce, loop_fuse, max_concurrency, max_interleaving, nofusion, and unroll [4]. A quick overview of what each of the pragmas does is listed in Table 2.

Adding loop pragmas to Csmith's random program generator is very similar to the random method of adding component annotations. We make similar changes in the CGOptions class, Probabilities class, and RandomProgramGenerator file. However instead of modifying the Function class, we modify the construction of the header for the StatementFor class. Before the creation of every for loop, we check if the flag for loop pragmas are turned on and then randomly

Pragma	Description
disable_loop_pipelining	Prevents compiler from pipelining a loop
ii	Forces a loop to have a loop initiation interval (II) of a specified value
ivdep	Ignores memory dependencies between iterations of this loop
loop_coalesce	Tries to fuse all loops nested within this loop into a single loop
max_concurrency	Limits the number of iterations of a loop that can simultaneously execute at any time
max_interleaving	Controls whether iterations of a pipelined inner loop in a loop nest from one invocation of the inner loop can be interleaved in the component data pipeline with iterations from other invocations of the inner loop
nofusion	Prevents the annotated loop from being fused with adjacent loops
unroll	Unrolls the loop completely or by a number of times

Table 2. Different for loop pragmas for Intel HLS Compiler

choose one of the loop pragmas to place before the loop. In our modifications to Csmith, we focus on the construction of for-loops because for-loops are the only loops in C that are supported by Csmith [12].

3.3.1 Static Variables. The Intel HLS Compiler allows function-scope static variables and recommends them for storing state in a component. It also allows the user to specify whether the memory system holding that static variable should be initialized each time the component is reset or only when it is powered on. To do so, the user annotates the static variable with either "hls_init_on_reset" (the default behavior) or "hls_init_on_powerup." This is another feature that a testing framework should target. However, Csmith does not generate function-scope static variables, so we added this ability and annotated some of the static variables with one of the two flags.

The process of adding static variables was similar to that of adding component annotations and loop pragmas. We introduce two new Csmith command-line flags: –static vars, which directs Csmith to label some function-scope variables as static variables, and –static-var-prob [probability], which lets the user specify the probability with which function-scope variables are static variables. As before, this involves changes in CGOptions, Probabilities, and RandomProgramGenerator. The Variable class is also modified so that when variable definitions are constructed, the static keyword may be added according to the provided probability. When it is added, it may be preceded by the annotation "hls_init_on_startup" (50% of the time), "hls_init_on_reset" (25% of the time), or nothing (25% of the time). The probability is divided this way because the latter two options are equivalent.

4 Results

In this section, we review the results of disabling different C99 features and the code produced from our added features.

4.1 Results of Disabling Different Features

Many features are disabled and enabled by a combination of flags and probabilities. We choose to investigate the groupings of flags that lead to production of different features in this section, namely generation of pointers, structs, unions, and floats. We initially thought to report on the number of errors per feature disabled, but this did not prove valuable as the size of a CSmith generated program on its own highly varies and the number of errors can simply depend on program size.

In order to enable pointer generation, we must remove the flags –no-const-pointers and –no-pointers. We must also cease setting the following probabilities to zero: select_deref_pointer_prob and pointer_as_ltype_prob. When we do this, the generated CSmith code produces a number of errors relating to the CSmith generated pointers, as shown below.

```
error: cannot initialize a variable of type
'int32_t **' (aka 'int **') with an rvalue of type
'void *'
int32_t **l_240 = (void*)0;
error: no matching function for call to 'func_32'
    (*l_145) ^= func_32(
    ((func_34((l_40[4][0][0] && p_12), g_18,
    p_11, l_41, &g_24) || p_13) , (void*)0));
```

At an initial glance, the errors do not seem related specifically to HLS compilation, and this was problematic for us at first. Initially, we believed that the errors were related to some of the behaviors being unsupported in C++ and having support in C99, but the Intel HLS documentation addressed that pointers are not supported by the compiler. Eliminating thes probabilities led to the complete disappearance of these errors.

To re-enable structs, we eliminate flag –no-packed-structs and set the struct_as_ltype probability back to default. There are additional probabilities related to structs, but these two are enough to turn them back on and see what errors are encountered. Below is an example of the errors that are encountered:

```
error: no viable overloaded '='
    g_63 = g_62;
```

```
error: no viable overloaded '='
    g_38 = g_37[0][4][2];
```

We were unable to resolve the source of these errors because Intel HLS claims to support structs. Unions result in similar behavior and we turn them back on by removing the –no-unions flag. To continue development, we simply disabled structs and unions.

Floats can be re-enabled by resetting float_as_ltype_prob and float_prob to their default values, as well as removing the –no-float flag. Doing so results in errors pertaining to missing functions, as shown:

```
error: no matching function for call to
'transparent_crc_bytes'
    transparent_crc_bytes (&g_5, sizeof(g_5),
        "g_5", print_hash_value)
```

The likely possibility here is that our CSmith installation is missing a function relevant to floats. As we looked into this, however, we discovered that float operations in HLS will require additional support anyway, so for simplicity, we disabled floats.

4.2 Code Generated by CSmith

In this section, we show some examples of code generated by the additions we made to Csmith. All additions here consistently compiled successfully under the -march=x86-64 flag in dpcpp or i++.

First is an example of generation of a component function:

```
component static int16_t  func_6(uint32_t  p_7,
    int8_t  p_8, uint32_t  p_9);

component static int16_t  func_6(uint32_t  p_7,
    int8_t  p_8, uint32_t  p_9)
{ /* block id: 5 */
    uint64_t l_18[1][3];
    int32_t l_19 = 0x501874DAL;
    int32_t l_132 = 0L;
    /* function truncated for length */
    return g_218[0][2];
}
```

Next is an example of generation of for loop pragmas:

```
#pragma nofusion
for(l_33 = 0; (l_33 >= (-13)); l_33 =
    safe_sub_func_uint64_t_u_u(l_33, 9))
{ /* block id: 13 */
    uint32_t l_40 = 5UL;
    g_19 = ((safe_mul_func_uint16_t_u_u(
       (safe_div_func_uint32_t_u_u((l_40 > p_22),
        4294967295UL)), l_41)) >= g_19);
    g_19 = ((safe_add_func_uint32_t_u_u(p_22,
        0x9DB2213AL)) && g_19);
}
```

Next is an example of generation of static variables with HLS arguments:

```
static int64_t l_10 = 0L;
static int32_t l_20 = 0x4653FFD0L;
hls_init_on_powerup static int32_t l_114 = 1L;
uint64_t l_122 = 5UL;
hls_init_on_reset static int32_t l_131 = 1L;
int32_t l_132 = 9L;
```

4.3 Issues in Different Compiler Targets

During our development process, outlined in the methodology section, we compiled using the i++ flag "-march=x86-64" as a means to run our FPGA code in an emulator flow without the need to install and run FPGA simulators. This was very useful to check our compilation, particularly while disabling C features and seeing errors disappear. However, when we changed the flag to "-march=CycloneV" to perform the synthesis and enable co-simulation, it revealed new errors. Namely, there are restrictions on the 'component' keyword when synthesizing the C code to Verilog. To compile in this co-simulation mode, there must be at least one component function, static variables must not be shared among components or shared by a component and a testbench, and components are not allowed to call other components. This came as a surprise to us as our use of the component keyword had not raised errors when using the "-march=x86-64" flag and we had not realized that it had a number of extra restrictions. This led us to change our approach for generating components, as described in the section on component functions. Other HLS-specific notations in C files have similar requirements, some of which are dependent on the compiler target architecture. These requirements must be well understood in order to accurately fuzz HLS compilers, and they complicate the fuzzing grammar.

4.4 Evaluation of Results

The results from our study of disabling various C constructs in Csmith shows that there can easily be successful fuzzing of many HLS compiler features simply by those disabling features in Csmith. However, there is substantial work to be done in order to fully test an HLS compiler with Csmith, as demonstrated by the complexity of adding HLS-specific specific features to the Csmith grammar. The additions to Csmith required to fuzz all the features of the HLS compiler will be cumbersome and sometimes target dependent. These changes are also specific to each HLS compiler, so the work extending Csmith for one HLS compiler will not be applicable to others. Therefore, we believe the most viable option is to fuzz adamantly on the subset of C99 that Csmith can easily produce, then reason about and prioritize the important features that must be added in order to adequately test the HLS compiler. Building a fully-featured HLS compiler fuzzer will be a difficult engineering task, but our success in extending Csmith with a few HLS-specific features shows that Csmith can serve as a reasonable starting point.

5 Discussion of Approach, Threats to Validity, and Additional Experiments

Our approach can be organized into three different steps - setting up a suite of HLS compilers on machines for testing purposes, modifying the probabilities of Csmith to block unsupported features of C99, and implementing features of HLS that are not supported by Csmith. Through the process of our testing and work, we found several challenges related to finding and setting up relevant compilers, navigating and modifying Csmith, and understanding the adjustable probabilities in Csmith. In order to gauge which HLS compilers were reasonably runnable on our machines, we first researched a number of HLS compilers. This included the attempted installation of most of the available HLS compilers.

Our first successful installation came with Instant SOC, however after more research and testing we found that Instant SOC only supports C++ code, which is incompatible with Csmith - the random C program generator. We also researched and installed ROCCC (the Riverside Optimizing Compiler for Configurable Computing), which compiles C code to VHDL. However, we found that it supports a limited subset of C and requires special formatting, which would make it more difficult to adapt Csmith to. It is also less prominent, which means it has less available support and fuzzing it may have a smaller impact overall than with a more popular compiler. After finding no success with these compilers, we refocused our efforts on the installation of Intel's HLS compiler on our local machines. This was difficult due to the compiler's dependencies and limited support for running the compiler by itself without the rest of the Intel Quartus Prime Pro Edition software suite, which requires a paid license. Intel's Devcloud, however, provides an already-prepared software stack that allows for easy set up and use of the Intel oneAPI DPC++/C++ Compiler (also known as dpcpp), which has an option to compile C/C++ code for FPGAs and generate Verilog. Since C is nearly a subset of C++, and the HLS compiler input is a subset of C, the next step was to take a deep dive into supported and unsupported C features. In general, the compiler is able to support static functions as well as classes, structs, and more. It does not support dynamic memory allocation, virtual functions, function pointers, C++ or C library functions, non-static class functions, and template functions without an explicit specialization.

Logging HLS features as well as unsupported C features led to an approach to modify the probabilities of Csmith to better support HLS compilers. Additionally, the HLS compiler's features such as loop pragmas, components, and static variables were features that we decided to add (triggered by flags). This is useful in testing the HLS compiler with programs related to HLS-specific features. This task proved to be difficult as Csmith's lack of documentation and difficult-to-navigate project led to hours of study and modification of code. Through much effort, we were able to extend Csmith to add support for HLS-specific features.

Through the approach taken, the HLS compiler is able to compile some of the randomly generated programs with the modified probabilities and HLS-specific features. However, some threats to the validity of the experiments include the limited HLS-specific features that we implemented as well as overly-limited C code due to the modification of the probabilities. While such tests and results are essential in proving the possibility of using Csmith to test HLS compilers, additional experiments could further refine the probabilities to match HLS's subset of C perfectly as well as add more HLS-specific features to potentially create more thorough test suites.

6 Conclusions and future work

We introduced a grammar-based HLS fuzzing extension to Csmith to modify existing random C program generation for HLS test input generation. Csmith's complex structures and functions for C compilers are applicable to HLS. As shown by our experimentation with random generation probabilities as well as our addition to Csmith to generate HLS-specific code, HLS-specific fuzzing is possible as an extension of random C generation. We can leverage the strength of Csmith's program generation to be applicable to HLS tools that are growing to be more prominent in industry-level FPGA development.

Through our grammar-based changes and modification of probability in Csmith, we are able to successfully extend Csmith to generate HLS compatible programs. Since this is possible, the next step in the extension of Csmith is generation of programs that can attack more HLS-specific components that may trigger HLS compiler bugs. HLS-specific features can be added or improved. For example, the loop pragmas can be implemented more intelligently to add pragmas dependent on the type of loop generated instead of the arbitrary bound that is currently used. Additionally, there are some probabilities, such as float_as_ltype_prob, that require special HLS libraries for generation. Csmith can be extended to have support for relevant special HLS libraries, which will be helpful for generating HLS-specific code that requires the libraries.

In order to detect HLS bugs that are generated by such changes, we can utilize differential testing without an oracle by running a deterministic program over multiple compilers in order to find HLS compiler bugs. We can also utilize an SMT solver that conducts an equivalence check for error detection. These methods are applied in "Finding and Understanding Bugs in C Compilers" [12] and "Finding and Understanding Bugs in FPGA Synthesis Tools" [3], which are relevant to the next steps for finding HLS compiler bugs. Another experiment that could be extended upon is the testing of an even wider scope of HLS compilers. Obstacles including outdated setup information as well as pay-to-use software limited our project to free and updated HLS compilers such as Intel's i++ on DevCloud. Future extensions for a larger variety of HLS compilers would be useful, and it would add to the extensive process of HLS testing. Since Csmith proves to be a usable program generator for HLS testing, it can be extended for more specific bugs and a generator for HLS that has complete coverage of HLS rules.

Acknowledgments

Special thank you to Yann Herklotz, who helped guide us, four strangers who sent him an email one day, in features to remove from CSmith and who's previous work inspired our own.

References

[1] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of High-Level Synthesis Tools. (2021), 219–223. https://doi.org/10.1109/FCCM51124.2021.00034

[2] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (oct 2021), 30 pages. https://doi.org/10.1145/3485494

[3] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. (2020), 277–287. https://doi.org/10.1145/3373087.3375310

[4] Intel. 2021. *Intel High Level Synthesis Compiler Pro Edition Reference Manual.* Retrieved March 6, 2022 from https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html

[5] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. *SIGPLAN Not.* 49, 6 (jun 2014), 216–226. https://doi.org/10.1145/2666356.2594334

[6] Christian Lindig. 2005. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA) *(AADEBUG'05)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1085130.1085132

[7] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

[8] Juan Perna and Jim Woodcock. 2012. Mechanised Wire-Wise Verification of Handel-C Synthesis. *Sci. Comput. Program.* 77, 4 (apr 2012), 424–443. https://doi.org/10.1016/j.scico.2010.02.007

[9] Nadesh Ramanathan, Shane T. Fleming, John Wickerson, and George A. Constantinides. 2017. Hardware Synthesis of Weakly Consistent C Concurrency. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/3020078.3021733

[10] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience* 37, 14 (2007), 1475–1488. https://doi.org/10.1002/spe.812

[11] Siemens. 2021. *Catapult High-Level Synthesis and Verification.* Retrieved March 6, 2022 from https://static.sw.cdn.siemens.com/siemens-disw-assets/public/2viQ3qHCWJQxzqSkwBauMQ/en-US/Siemens-SW-Catapult-HLS-HLV-Platform-FS-82981-C1_0921.pdf&sa=D&source=docs&ust=1646637738974372&usg=AOvVaw2pmYqbmHNZ030UeX2huICc

[12] Eric Eide Xuejun Yang, Yang Chen and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (jun 2011), 283–294. https://doi.org/10.1145/1993316.1993532

[13] Eric Eide Xuejun Yang, Yang Chen and John Regehr. 2021. *CSmith.* Retrieved March 6, 2022 from https://embed.cs.utah.edu/csmith/

[14] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book.* CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2021-10-26 15:30:20+02:00.

A Installation and Execution

A.1 Setting up Intel DevCloud

1. Obtain access to Intel DevCloud by creating an Intel DevCloud account.
2. Log into Intel DevCloud through this JupyterLab link
3. Click Launcher and then click Terminal.
4. Add required files to use dpcpp like i++.

```
git clone
https://github.com/jasonpanelli/hls-include
```

A.2 Installing and Running HLSCsmith

1. Clone HLSCsmith

```
git clone
https://github.com/jasonpanelli/csmith
```

2. Go into Csmith directory

```
cd csmith
```

3. Build and Install HLSCsmith. The prefix can be anything you choose to be, we chose the name "build".

```
cmake -DCMAKE_INSTALL_PREFIX=build .
make && make install
```

4. Add HLSCsmith to PATH

```
export PATH=$PATH:$HOME/csmith/build/bin
```

5. Run HLSCsmith.

   a. Pure Csmith

   ```
   csmith > random.c
   ```

   b. Flags to run HLSCsmith with the probability file. Probability file can be found in the repository

   ```
   csmith --probability-configuration prob.txt
   --no-argc --max-array-dim 3 --max-funcs 5
   --max-expr-complexity 2 --no-float
   --no-embedded-assigns --max-block-depth 2
   --no-unions --no-packed-struct
   --no-const-pointers --no-pointers
   --strict-const-arrays > random.c
   ```

   c. Flag to run HLSCsmith with component function. To change the probability of component function, open

prob.txt and modify the component_function_prob=<number between 0-100>.
`csmith --component-function > random.c`

d. Flag to run HLSCsmith with loop pragmas. To change the probability of loop pragams, open prob.txt and modify the loop_pragma_prob=<number between 0-100>.
`csmith --loop-pragmas > random.c`

6. Compile generated file

```
dpcpp -Wno-narrowing -fintelfpga -march=x86-64
[filename] -o [output executable name].exe
-I[path to hls-include]
```

```
dpcpp -Wno-narrowing -fintelfpga -march=x86-64
random.c -o random.exe -Ihls-include
```

7. Run compiled file. Should generate a checksum.

```
./random.exe
```