It's About Time: Detecting Timing Side-Channel Vulnerabilities in High-Level Synthesis Designs

Denis Zuppiger*
ETH Zurich
Zurich, Switzerland
zdenis@student.ethz.ch

Katharina Ceesay-Seitz*†
ETH Zurich
Zurich, Switzerland
kceesay@ethz.ch

Jiahui Xu
ETH Zurich
Zurich, Switzerland
jxu@ethz.ch

Lana Josipović
ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

Kaveh Razavi
ETH Zurich
Zurich, Switzerland
kaveh@ethz.ch

Abstract

High-level synthesis (HLS) is becoming increasingly popular because it allows the specification of hardware accelerators in a higher level of abstraction than Register Transfer Level (RTL), for example, in C/C++. Common scenarios include acceleration of machine learning inference and even cryptographic functions. But is HLS secure for such scenarios?

Constant-time (CT) programming is a common approach for mitigating timing side channels when executing sensitive software on CPUs. It is unclear whether generic CT rules such as not passing secret-dependent data to memory addresses or branching decisions provide adequate protection when an HLS flow turns CT software into hardware. Furthermore, CPU vendors and instruction set architectures (ISAs) often specify which instructions execute in data-independent time and are safe to use for CT programming, which is currently missing for HLS. To address this gap, we introduce CTurtle, the first C program generator that generates random, yet valid, C programs that follow customizable CT rules, making them CT with respect to certain inputs marked as secret. We further introduce TurboTurtle, an HLS fuzzer that can discover cases where HLS flows turn CT software into hardware that is vulnerable to timing side channels. TurboTurtle detects timing vulnerabilities in the HLS generated Register Transfer Level (RTL) designs, using a miter test bench that can detect when the generated accelerator may take a variable amount of time caused by different secret inputs. Applying TurboTurtle on four different HLS flows, we discover 12 new vulnerabilities where HLS flows produce non-CT hardware from CT software, for which we got two CVEs. Eight of these vulnerabilities establish unsafe operators for a particular HLS flow, and three persist in two HLS flows even after applying generic and tool-specific CT rules.

*Equal contributing first authors.
†Corresponding author.

CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832683

CCS Concepts

• **Security and privacy → Hardware security implementation**.

Keywords

Timing side channels, high-level synthesis, hardware security

ACM Reference Format:
Denis Zuppiger, Katharina Ceesay-Seitz, Jiahui Xu, Lana Josipović, and Kaveh Razavi. 2026. It's About Time: Detecting Timing Side-Channel Vulnerabilities in High-Level Synthesis Designs. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3830454.3832683

1 Introduction

High-level synthesis (HLS) is gaining popularity as it allows the specification of hardware accelerators at a higher level of abstraction than the Register Transfer Level (RTL), such as in C/C++, reducing development time and thus time to market [1, 13, 28, 47, 63, 75, 81]. HLS-generated designs often accelerate applications that process sensitive data, such as machine-learning workloads or cryptographic algorithms [10]. Hardware, however, has been shown to be vulnerable to a plethora of timing side channels that undermine the security of such applications [7, 18, 56, 67]. While there is some evidence that HLS may introduce timing side channels in the generated hardware [68, 69], we currently lack a method to systematically find such cases in various HLS flows. To address this gap, we introduce TurboTurtle, the first fuzzer that is capable of testing HLS flows against timing side channels. Our extensive evaluation shows that existing HLS flows may turn software that has been hardened against timing side channels with constant-time programming into hardware that is vulnerable to timing side channels in realistic scenarios.

Timing side channels in hardware. Programs executed on general-purpose CPUs may leak secrets via timing side channels when their execution time depends on secret data. These timing side channels stem from three sources: First, when the secret data is used in making branching decisions in the program, resulting in different execution times from each side of the branch. Second, when secret-dependent memory accesses leave traces in the caches and influence execution times [53]. Third, particular instructions may have data-dependent execution latencies [7, 18, 56, 74], which

led CPU vendors and *instruction set architectures* (ISAs) to specify instructions with *data-independent timing* (DIT) [43, 60, 76].

Constant-time programming. *Constant-time* (CT) programming is the main software countermeasure against timing side channels [5, 19]. Software that avoids branching on secret-dependent data, does not perform secret-dependent memory accesses, and does not pass secret-dependent data to instructions with data-dependent execution latency is believed to be free of timing side channels when executed on general-purpose CPUs. Recent studies have shown that this assumption does not always hold. CT programs may be compiled into non-CT instruction sequences unexpectedly [29, 31, 79]. Hence, it is common practice for developers to write the security-sensitive CT portion of their software in assembly. Yet, this practice is not possible for HLS flows, as the input programs are not compiled into assembly instructions.

Timing side channels from HLS. As discussed, HLS flows do not have their own assembly instructions given their high-level nature. This means that the HLS user must write high-level programs that avoid the introduction of timing side channels. A natural thought is to follow CT rules specified for software that is intended to run on CPUs. However, the effectiveness of these rules for HLS has not yet been sufficiently explored. Similar to CT software that can be compiled to non-CT instruction sequences, an HLS flow might turn a seemingly CT code into a hardware design that is vulnerable to timing side channels. Similar to how the latency of certain instructions may be input-data-dependent, certain HLS constructs may also introduce timing side channels in the synthesized hardware. Unlike general-purpose CPUs, however, there are no DIT specifications for HLS flows. This means that a user of an HLS flow cannot know which high-level programming constructs are safe to use.

TurboTurtle. The first challenge is to develop a method that can establish whether a side channel was introduced by the HLS flow. To this end, we design and implement TurboTurtle, an HLS fuzzer for finding language constructs that, despite following CT rules, may lead to synthesizing hardware designs that are vulnerable to timing side channels. TurboTurtle takes inputs that are CT with respect to certain input variables, and determines whether HLS flows preserve the CT properties in the generated RTL. To identify timing variabilities in the RTL, we employ a miter circuit [20, 23, 92] that compares the execution time of two instances of the synthesized design, feeding them with matching random data for public inputs, and differing secret inputs. If the two instances exhibit different execution times, the timing depends on the secret input, thereby introducing a timing side-channel vulnerability into the design. To find vulnerabilities effectively, TurboTurtle needs sufficiently diverse programs which we address with CTurtle.

CTurtle. To the best of our knowledge, there exists no C program generator that can produce random programs, which follow CT rules. We introduce CTurtle, the first random C program generator whose programs are CT-rule-compliant with respect to certain inputs. CTurtle accepts the specification of custom CT rules, allowing us to iteratively discover HLS tool-specific rules during our fuzzing campaign. Rules can be composed of C code constructs, and HLS configurations, like pragmas, directives and compiler arguments. CTurtle then maintains a secret and a public data flow and ensures that the generated C programs comply with the custom CT rules on the secret data flow. Upon the discovery of a timing side-channel vulnerability, TurboTurtle can automatically reduce the generated program to a minimal human-understandable form, while maintaining rule compliance.

Vulnerabilities. We evaluate TurboTurtle by fuzzing two open-source HLS flows, Bambu [28] and Dynamatic [1], and two commercial HLS flows. TurboTurtle applies a random set of HLS configurations, compiler flags, and directives. In total, we generate and verify 8735 RTL designs. We find 12 vulnerabilities in Bambu and Dynamatic that turn constant-time C code into non-constant-time hardware. One of these vulnerabilities, C short-circuit evaluation, is inserted by the frontend compilers, equally affecting CT programming for CPUs. Some vulnerabilities, e.g., triggered by unsafe operators in Bambu, are caused by the HLS middleend inserting data-dependent decisions into the input program's control flow. Another vulnerability is introduced in the backend, where Dynamatic generates RTL with different data-dependent execution paths from input programs that originally do not have such path. We show that these insecure code constructs can be found in real-world machine learning and cryptographic code, compromising hardware that is generated by the HLS compilers under our study.

Contributions. We make the following contributions:

- We develop CTurtle, the first C program generator that generates random constant-time programs. The CT rules are configurable and can be combined with HLS pragmas, directives, and configurations.
- We develop TurboTurtle, a fuzzer that allows us to test many input programs and HLS compiler configurations for timing side-channel vulnerabilities.
- We test two open-source and two industrial HLS flows for timing side channels. We discover 12 new timing side channels introduced by the open-source HLS tools into RTL generated from CT programs. Eight establish that certain operators should not be used in some HLS flows combined with particular HLS configurations, and one vulnerability can be triggered by various different code constructs.
- We show real-world examples of machine learning and cryptographic code that are affected by such vulnerabilities.

2 Background

We give background on HLS, timing side channels, and compiler verification and testing.

2.1 High-level synthesis

HLS compiles a behavioral software program written in a *high-level programming language* (HLL), such as C or C++, into a functionally equivalent, potentially sequential, *register transfer level* (RTL) design written in a *hardware description language* (HDL), such as Verilog or VHDL [49]. In hardware/software co-design, HLS is used to synthesize a user-selected program function into RTL for accelerated execution on a *field-programmable gate array* (FPGA); the rest of the program runs on a CPU [6]. Function arguments and return statements are synthesized into hardware inputs and outputs, respectively, and external memory accesses into various hardware bus interfaces [1, 6, 26, 28].

HLS flow. A typical HLS flow is shown in Figure 1. The frontend is often inherited from CPU compilers to ensure static correctness,

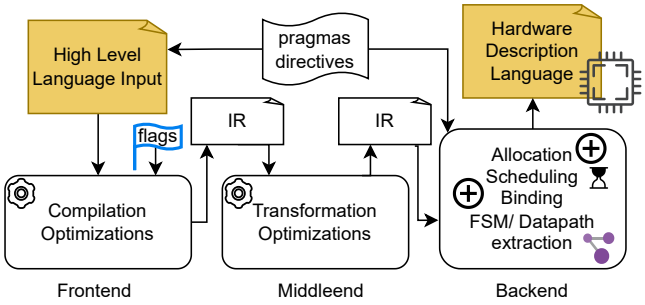

Figure 1: Overview of the high-level synthesis flow.

perform software-level optimizations [80], and emit an *intermediate representation* (IR). Sometimes, a middleend performs further HLS-specific software transformations and optimizations [28].

HDL generation. The backend generates the HDL output. First, it allocates the hardware resources needed to perform all data operations present in the input HLL [28]. Next, it schedules operations according to their control and data dependencies. This step introduces clock-cycle delays into the so-far untimed program. *Static HLS* compilers assign each operation to a *fixed* clock cycle during scheduling. The recently-proposed *dynamic HLS* promises better performance [49] by making scheduling decisions during runtime [47]. The backend then proceeds to the binding step, where it aims at optimizing the number of hardware resources. Based on the *control data flow graph* (CDFG), the backend then generates the finite state machines (FSMs) that steer the data flow through the hardware resources. Finally, it outputs the HDL files [75, 80].

Pragmas and directives. To achieve power, performance, and area goals, HLS pragmas, directives, and compiler flags can be applied either globally or to specific parts of the HLL code [6, 26, 28]. They have a significant impact on the generated HDL output. For example, they direct the HLS compiler to unroll or pipeline loops, inline functions, or add load-store queues for loads from memory. They can also be used to describe constraints (e.g., available functional units) that the HLS compiler must follow.

2.2 Timing side channels

A timing side channel in software or hardware occurs when the execution latency of a program or computation depends on secret data. In that case, attackers can infer the value of secret data by measuring variations in execution times [98].

Constant-time programming. Constant-time programming is a software-based countermeasure against timing side channels, often used in cryptographic software [53]. It specifies a set of rules for writing programs for general-purpose CPUs [5]:

Rule #1: no secret-dependent control flow. Software timing side channels can occur when secret-dependent divergent control flows are compiled into paths with different numbers of instructions or into instructions with different execution times [21, 23, 40, 92].

Rule #2: no secret-dependent memory accesses. In CPUs, hardware optimizations such as caches can cause address-dependent execution latencies, depending on whether data at an address is present in the cache or not [53]. Secrets can leak if memory addresses are derived from secret data.

Rule #3: no instructions with data dependent timing. Timing side channels can arise from instructions with data-dependent execution latencies. Certain instructions, like divisions or floating point operations [7, 54, 74], may execute with data-dependent execution latency in certain hardware implementations [5, 54]. As such side channels are CPU-specific [21, 23–25, 39, 92], CPU vendors inform software programmers about which instructions are secure [43, 60, 76].

2.3 Compiler verification and testing

Many software compiler [31, 61, 79, 97] and hardware synthesis tool verification techniques [38, 86] exist, but HLS compiler verification is less explored [36, 37]. Compiler verification needs two input kinds: software programs (compiler input) and data (program input).

Fuzzing compilers. Compiler transformations may be formally verified, providing guarantees for all possible inputs of both kinds. However, this method requires high tool-specific effort and source code access [37, 57, 69]. Instead, black-box fuzzers do not rely on the internal workings of a device under test (DUT) and can thus be easily reused for multiple DUTs [59, 64, 84]. A challenge when fuzzing is how to generate sufficiently diverse inputs to cover a large part of the input state space [18, 64, 84, 90]. Compilers may be tested by compiling many existing libraries. As an example, previous work uses cryptographic libraries for finding CT violations introduced by software compilers [29, 31, 79]. A wider compiler input space can be covered with random program generation [9, 36, 86]. Recently, CPU fuzzers have generated random CT assembly programs, targeting x86 [64] or RISC-V CPUs [18, 56]. Yet, we currently lack a method for generating CT HLL programs.

Fuzzing compiler outputs. Software compilers (or hardware synthesizers) are often tested for bugs via differential testing. Random data inputs are applied to two compiler outputs obtained from the same input program, but from different compilers, and the program's results (or hardware's outputs) are compared against each other [31, 86, 97]. This technique is unsuitable for finding timing side channels produced by compilers, as different compilers are not expected to generate binaries or hardware with the same execution latency. Synthesized hardware is often formally verified by comparing different abstractions of the same design, covering all possible design inputs [38]. Hardware timing side channels can be found via self-composition [18, 23–25, 92] or taint tracking [8, 41, 85, 89] using formal verification [21, 23–25, 39, 92] or random simulation [18, 56].

To systematically study multiple HLS tools, we build a generic black-box fuzzer that randomizes HLL input programs and data.

3 Threat Model

We consider a threat model where an HLS-synthesized hardware accelerator operates on sensitive input data. We further consider the source code of the HLL program to be public, and that the attackers are able to invoke the accelerator (or otherwise determine when the operation has started). If they can observe the end of the operation, they are able to measure its execution time. The accelerator's operation could be, e.g., triggered via a custom CPU instruction from a program running on a CPU that interfaces with the accelerator. Sub-clock-cycle timing variations and power side channels are out of scope.

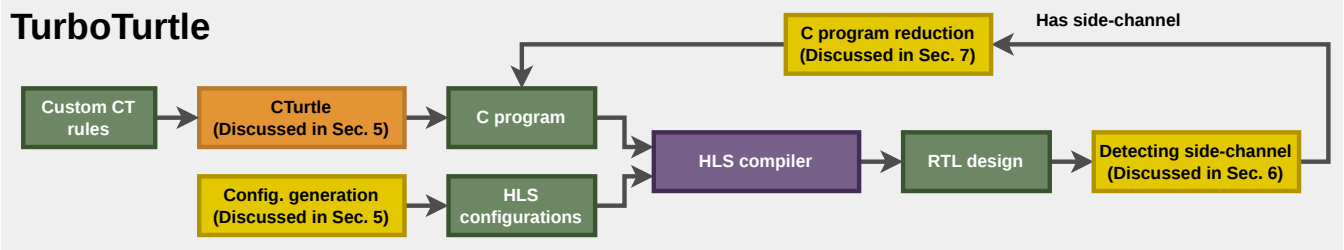

Figure 2: The architecture of TurboTurtle: CTurtle generates a random C program with custom CT rules (see Section 5). We select random HLS pragmas and inject them into the C program. An HLS compiler takes the generated inputs and produces HDL output. If TurboTurtle detects a vulnerability (see Section 6), it starts reducing the C program (see Section 7), HLS pragmas and HLS directives and restarts the HLS to verification workflow.

4 Motivation and Challenges

We motivate the need for automated testing of HLS-specific CT rules and discuss the challenges involved.

4.1 Motivation

Missing HLS constant-time rules. We first discuss the CT rules in the context of HLS.

Secret-dependent control flow: Compiling a program for being executed on a CPU and synthesizing it with HLS are fundamentally different. In the former case, the program is compiled into instructions to execute on the target CPU; in the latter, the program is converted into custom hardware. Consider a CPU with two parallel multiplication units and a program with an unbalanced secret-dependent branch, where one branch path performs one multiplication, whereas the other performs five independent ones. On this CPU, the second branch path would have to be executed as several sequential instructions and thus the branch paths have different execution latencies. If user-defined constraints permit, an HLS compiler could synthesize this program into a combinational circuit where both branch paths execute in one clock cycle, leading to no side channel; yet, it may equally generate an unbalanced FSM, where one side of the branch is synthesized into more state transitions than the other. There is evidence that secret-dependent control flow in HLS input programs leads to timing side channels in the generated hardware [41, 45, 68, 69, 80]. CT *Rule #1* is often suggested to avoid such side channels, but as we will show in Section 8, a program that is perfectly compliant with CT *Rule #1* can be translated into a vulnerable circuit by the HLS compiler.

Secret-dependent memory accesses: In a program intended to run on a CPU, CT *Rule #2* forbids secret-dependent pointers. HLS may convert arrays into internal registers with constant access time, or into memories. External memory hierarchies may implement caches, or HLS compilers can generate cache structures themselves [15]. Thus, if an HLL program performs secret-dependent accesses to arrays, we hypothesize that a secret could be leaked in some cases, and that avoiding such accesses might avoid timing side channels.

Studying the effect of HLS on CT programs. Several studies [29, 31, 79, 100] suggest that compiler optimizations can introduce timing side channels into CT software. Cryptographic libraries often implement critical parts of the code in constant-time assembly to avoid unwanted consequences through compilation [11]. We suspect that HLS compilers may equally introduce timing side channels into hardware generated from CT HLL programs. As HLS input programs are never translated into CPU instructions, HLS programmers cannot hard-code the security-sensitive code regions with assembly, and no guidelines exist yet on how to write CT HLL code for HLS. To systematically study whether HLS compilers invalidate CT programming rules during *any* stage of the HLS flow, we need a method that detects timing side-channel vulnerabilities in the generated RTL and can establish if the vulnerability was introduced by the HLS flow. This leads to our first research question:

Q1. How can we systematically and automatically find timing side channel vulnerabilities introduced by HLS flows?

We address this question by relying a large number of CT HLL input programs and verifying the HLS output. To cover a large input space, our input programs should have data flows of different sizes and complexities, combine different data types and operations, and include HLS compiler-specific configurations, such as HLS pragmas and source code labels for HLS compiler directives. One option to obtain CT C programs would be to use existing C program generators [9] and verify their output or existing C programs for CT with existing CT verification solutions as listed in Table 3 of Section A.1. However, this option is unsuitable, as many of these solutions either operate on CPU-specific assembly, or on an IR and thus their results are not translatable to the C source code. Hence, we opted for building CTurtle, which generates CT programs by construction with an extendable set of CT rules.

Variable-latency operations: It is further unclear how a *Rule #3* could be specified for HLS input programs, as it is specified on CPU instruction level. HLS compilers might compile data operations into hardware units with data-dependent execution latencies, but we currently have no evidence that this happens in practice yet. They might also produce side channels from other, not yet known, code constructs. To test the effect of yet-to-be-established CT rules we require a method that allows us to discover new HLS CT rules and determine whether HLL programs follow these rules. This leads us to our second research question:

Q2. How can we establish CT rules for HLS flows?

We address these two research questions with TurboTurtle and its evaluation on popular HLS compilers.

4.2 TurboTurtle: An HLS compiler fuzzer for detecting timing side channels

To address these research questions, we introduce TurboTurtle—a HLS compiler fuzzer for detecting timing side-channel vulnerabilities. A high-level architecture of TurboTurtle is depicted in Figure 2. TurboTurtle addresses the following challenges:

C1. Obtaining random HLL programs and HLS configurations that follow customizable constant-time rules.

To the best of our knowledge, no viable solution currently exists for addressing this challenge. Random program generators or tools targeting CT violations cannot be configured to follow custom or HLS-specific CT rules [9, 27, 30, 44, 97]. Thus, we build CTurtle, the first CT C program generator, introduced in Section 5. CTurtle generates random C programs, following known [5] or customizable CT rules. This allows us to assess HLS flows regarding timing side channels and potential CT rules. Starting from function arguments, CTurtle constructs CT programs by maintaining a secret and a public data flow within the generated program, and applying customized CT rules to them during program construction. Random pragmas, directives, and compiler flags can be added by TurboTurtle.

C2. Efficiently and automatically detecting timing side channels in HLS-generated hardware designs.

To detect if the generated RTL design has any secret-dependent timing behaviors, TurboTurtle constructs a miter circuit that compares the execution time of two instances of the RTL design when providing matching random data to public inputs and differing values to secret inputs. TurboTurtle automatically generates test benches and uses random simulation to detect timing side channels. We discuss our method in Section 6 and evaluate the usage of formal verification instead of simulation in Section 8.1, finding that simulation finds vulnerabilities more efficiently.

As our last challenge, we want to know what HLS compiler inputs have caused the generation of designs with timing side-channel vulnerabilities.

C3. Determining the cause of timing side channels introduced by HLS compilers.

For large HLL programs, manually determining which statement or HLS configuration causes a timing side channel is very time consuming, as the HLS transformations are complex and sometimes proprietary. Section 7 describes how TurboTurtle automatically and iteratively reduces the vulnerable HLL programs, without breaking the CT rules or removing the vulnerability, while keeping the programs valid. We further automatically determine the vulnerable HLS configurations for a program, allowing us to specify HLS tool-specific CT rules (see Section A.2.3).

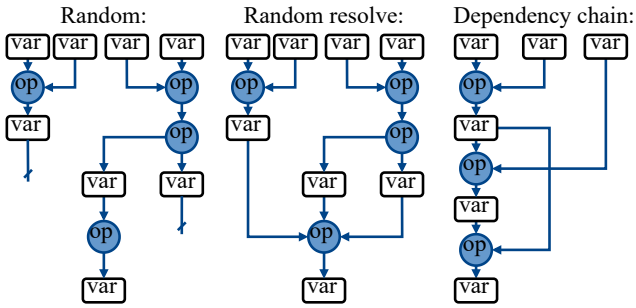

Figure 3: Overview of our *data flow generation strategies* var = variable, op = operation. The last 'var' represents the return value.

5 Generating Constant-Time C Programs

This section describes CTurtle, the first C program generator that produces random C programs, which adhere to software constant-time rules, including configurable rule extensions. CTurtle is a grammar-based program generator [83] that generates syntactically valid programs by adhering to the C grammar [2], as well as to customizable rules.

5.1 Data flow generation strategies

To avoid biasing the generated programs with our design decisions, we implement various data flow generation strategies, as shown in Figure 3, and randomly pick one for each program generation. The random strategy (left) generates completely random programs and returns a randomly chosen variable. Variables that do not influence the return value can be optimized out by the front-end compiler and are never synthesized, wasting generation time and skewing program length. The 'random resolve' strategy avoids these problems by adding a return statement that combines all variables in a single addition expression. Yet, such programs have potentially short dependency chains between statements. The *dependency chain* strategy assumes that many of these chains could be synthesized into parallel code with few CF transitions, or even into combinational programs; it generates sequentially-chained, data-dependent variable assignments, thus producing long dependency chains in the data flow, which require consecutive hardware states. It returns the last assigned variable.

CTurtle constructs a secret and public data flow in the HLL program. To test potential influences between the secret and public data flow, CTurtle can control their interaction during data flow generation. With the *separated* strategy, the public data flow never merges with the secret data flow. However, we let the public data flow influence the secret data flow indirectly through control-flow statements. Using the *mixed* strategy, variables from the public data flow can be merged into the secret data flow, but not vice versa. Section 8.2 evaluates the performance of the generation and mixing strategies.

5.2 Program generation

We now discuss the CTurtle's program generation algorithm outlined in Algorithm 1.

Algorithm 1 Random Program Generation

```
1:  struct Program:
2:    List<Argument> Arguments; List<Declaration> Declarations;
3:    List<Statement> Statements; List<Rule> Rules;
4:    Integer MaxStatementCount; Strategy strategy;
5:  endstruct
6:  method Program::GenerateProgram()
7:    GenerateArguments()
8:    GenerateSubfunctions()
9:    GenerateVarDeclarations()
10:   AssignArgumentsToVariables()
11:   while Statements.size ≤ MaxStatementCount do
12:     Statements.Append(GenerateStatement())
13:   GenerateReturnStatement()
14: endmethod
15: method Program::GenerateStatement()
16:   StatementType = GetLegalTypeRandom()
17:   if StatementType is ASSIGNMENT then
18:     lhs, lhs_secrecy = PickVariableRandom()
19:     rhs = GenerateExpression(lhs_secrecy)
20:     return AssignmentExpression(lhs, rhs)
21:   else if StatementType is FUNCTION_CALL then
22:     Args ←PickArgumentsRandom()        ▷ Considering secrecy
23:     Func ←PickSubFunctionRandom()
24:     return GenerateFunctionCall(Args, Func)
25:   else if StatementType is IF_ELSE then
26:     Cond ←GenerateConditionExpression()
27:     If ←PopulateScope()          ▷ Calling GenerateStatement()
28:     Else ←PopulateScope()
29:     return IfElseStatement(Cond, If, Else)
30:   ▷ Other constructs are omitted for the sake of brevity.
31: endmethod
```

(Line 7–10): CTurtle generates a top function with randomly chosen secret and public function arguments through GenerateArguments(). CTurtle then randomly declares a set of secret and public variables through GenerateVarDeclarations(), and assigns them with function arguments of matching secrecy through AssignArgumentsToVariables(). The variables can be of random data types, which can be constrained by configurable rules, and can include one-dimensional arrays. For example, after the steps above, we can obtain the following code:

```
int foo(int arg0_secret, bool arg1) {
  int var1_secret = arg0_secret;
  bool var2_secret = (bool) arg0_secret;
  int var3 = (int) arg1;
  int var4_secret = arg0_secret; ... }
```

(Lines 11–12): Following the variable declarations, CTurtle propagates the variables through a random number (up to the configured MaxStatementCount) of randomly generated statements via GenerateStatement(). Specifically, GenerateStatement() is responsible for generating assignments, control-flow constructs, and function calls. When generating statements, CTurtle can be configured to adhere to customizable CT rules. For example, it can avoid generating forbidden operations, prevent the usage of secret data when computing public data, or prevent secret data from being

Statements	Expressions / Operators	Data Types
compound, if, if else, switch, while, do while, for loop, expression, return, break	empty, function-call, cast, *, /, %, +, -, <<, >>, &&, =, [], <, >, <=, >=, !=, &, ^, |, ||	(u) char, (u) short int, (u) long int, (u) long long int, float, double, long double

Table 1: All the statements, expressions, and data types CTurtle supports. The integer data types can be signed or unsigned. (u): unsigned variant exists. For each data type, there exists a derived array type.

used to determine the control flow. Additionally, the user can supply arbitrary custom rules to avoid tool-specific unsafe constructs. The following details how CTurtle generates each statement type.

Generating assignment statements. (Line 17–18): When CTurtle generates an assignment statement, it first randomly picks a left-hand side (LHS, the one that gets a value assigned) variable or array access location via PickVariableRandom(). (Line 19–20): Then, in GenerateExpression(), it chooses the expression type (see expressions/operators in Table 1; e.g., +, −) to be used on the right-hand side (RHS) and populates its operands. If the *dependency chain* program generation strategy is active, at least one operand must be the one that was most recently assigned. Since operations can operate on a combination of public and secret data, when populating the RHS operands, CTurtle chooses variables depending on the LHS's secrecy to exercise the potential influences between public and secret data. CTurtle can follow different strategies during the RHS variable selection. The *separated* strategy forbids any mixing between the secret and public data flows, that is: when LHS is a secret variable, then all RHS variables must be secret. The *mixed* strategy allows public variables to be used as operands when the LHS is secret. However, in this case, at least one of the RHS operands must be secret data to maintain the secret data flow. For both strategies, if LHS is a public variable, then none of the RHS operands can be a secret. This avoids inadvertently introducing secrets into the CF in case *Rule #1* is active. For example, the *mixed* strategy permits the following kinds of expression, whereas the *separated* strategy forbids them:

```
// This assignment is legal in the "mixed" strategy
// but illegal in the "separated" strategy
var4_secret = (bool) arg0_secret + var3;
```

Adhering to CT rules in assignment statements: For array accesses, CTurtle randomly selects a variable as the index. When *Rule #2* is active, CTurtle will not use secret variables as array subscript operators so that there will be no memory accesses based on secret data in the generated hardware. There currently exists no *Rule #3* for HLS flows. Because our goal is to identify cases of *Rule #3* for HLS compilers and enable fuzzing of future HLS compiler versions, we equip CTurtle with the capability of accepting customized rules. Combinations of data types, operators, pragmas, directives, and HLS configurations can be configured as a custom CT rule. (Line 21–24): CTurtle generates function calls by assigning a variable with a call to the previously generated subfunctions. As with expressions, it randomly picks arguments following the currently active configuration.

Generating control-flow (CF) statements. (Line 25–29): CF statements like if-else, while and for loops, and switch statements consist of a condition evaluation and the conditionally executed statements.

CTurtle iteratively calls GenerateStatement() when populating the conditionally executed statements via PopulateScope(); potentially, it could introduce nested control-flow structures, capping at 3 nested levels in our evaluation. To generate loops with data-dependent numbers of iterations, CTurtle places early exit conditions inside the loop body, which can depend on secret or public data, depending on the configured CT rules.

Adhering to CT rules in CF statements: (Line 26) When *Rule #1* is enabled, no secret data is allowed to be used to evaluate the condition expression. For example, the following code would **not** be generated by CTurtle if *Rule #1* is enabled:

```
if (var4_secret) { ... }
```

Generating linearization statements. (Line 30) CTurtle can optionally generate linearization statements like those introduced in that mimic linearized, masked, or conditional selections. Such statements enable decisions based on secrets without introducing secret-dependent CF into the C code. For example, CTurtle can generate linearized branches using a masking technique as shown below:

```
result = (secret_condition) * (true_expr) +
         (1 - secret_condition) * (false_expr);
```

Variable `result` is assigned with `true_expr`, i.e., the value that should be assigned in case `secret_condition` is true, and with `false_expr` otherwise.

Equipping HLS code with pragmas and directives. TurboTurtle and CTurtle randomly configure the HLS process by annotating functions and loops with HLS pragmas or adding directives to the HLS compiler configuration (discussed in Section 2.1). Section A.3 reports the supported pragmas and directives.

Optimizing fuzzing throughput. Undefined behaviors of syntactically correct C programs may cause C-RTL mismatches or compiler crashes. To avoid this, CTurtle rules out programs with undefined behaviors. For example, to avoid invalid programs containing out-of-bound array accesses, CTurtle uses a modulo operator to reduce the index value to the size of the array. Alternatively, CTurtle can be configured to limit the index variables to specific data types (e.g., 8-bit) and let arrays be large enough (e.g., 256 elements). To ensure a program can terminate, CTurtle generates loop statements with constant bounds and allows data-dependent early exit as mentioned above.

6 Detecting Timing Side-Channel Vulnerabilities in HLS-Generated Designs

We describe how TurboTurtle detects the timing side channels in the HLS-generated RTL design. For a given CT C program, TurboTurtle invokes an HLS compiler with a randomly-generated configuration to obtain an RTL design. Then, TurboTurtle generates a testbench to search for timing side channels in the generated RTL design. The following describes the procedure for creating this testbench and detecting timing side channels.

6.1 Preparing the testbench

Identifying the ports. TurboTurtle parses the generated RTL design using an open-source Verilog parser, Slang [70], and automatically identifies the necessary control ports of the top module such as clock, reset, start, and done signals. It maps the data ports of the top module to the arguments of the top C function to identify which of the HDL data ports carry secrets, and which public data. Having identified the data ports, TurboTurtle collects their names, directions, and bitwidths, and stores them for later processing.

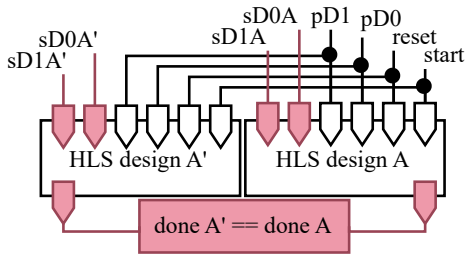

Figure 4: A miter composed of two copies of the same HLS-generated design (A and A'). Public input pairs are driven with the same value (e.g., pD0 is passed to A and A'), while secret input pairs receive differing values (e.g., sD0A and sD0A'). A difference in the done signals indicates a timing side channel.

Building the miter testbench. TurboTurtle constructs a miter testbench [20, 23, 92] to monitor the timing difference between circuit executions with different secret values. We chose this setup since it can be easily plugged into a model checker, and compared to sequentially simulating the same design with different inputs, this setup minimizes simulation time by allowing early termination. Figure 4 illustrates such a miter testbench generated by TurboTurtle: It instantiates two identical copies of the generated HLS design, named A and A', in the testbench. The testbench drives each corresponding public primary input (i.e., the global input of the circuit) pair (e.g., `pD0` and `pD0'`) with the same signal (shown as `pD0` in the figure), but uses different signals for corresponding secret input pairs (e.g., `sD0A` and `sD0A'`). This means that circuits A and B receive matching random public inputs and different random secret inputs.

6.2 Detecting timing variations

Simulating the testbench. When TurboTurtle starts the simulator, the testbench simultaneously resets the two circuits with identical initial states, applies random data inputs to the data ports, and starts their execution. The testbench then waits for the two RTL designs to assert the `done` signals, which indicate the end of execution.

Detecting the side channel. If the `done` signals (e.g., `done A`, `done A'` in Figure 4) assert in different clock cycles, then the two executions have different latencies. In this case, since the only discrepancy between the two executions is caused by the secret inputs, the RTL design's execution time depends on their values and thus contains a timing side-channel vulnerability.

After identifying the vulnerability, TurboTurtle performs post-processing to facilitate the developer in addressing the vulnerability. We discuss this in the following section.

7 Test Program Reduction

Even when an RTL design is synthesized from a program with only a few C statements, the RTL output can be large and the exact

Algorithm 2 Program Reduction Algorithm

```
1:  method ReduceProgram(Program)
2:      while NotEmptyAndVulnerable do
3:          NotEmpty = Program.ReduceStatements(Tail)
4:          Vulnerable = ReVerify()
5:          if NotEmptyAndVulnerable then
6:              NotEmpty = Program.ReduceStatements(Head)
7:              Vulnerable = ReVerify()
8:      return Program
9:  endmethod
10: method ReduceStatements(Head)          ▷ Head = 1, Tail = 0
11:     Start = Head ? 0 : Program.Middle
12:     End = Head ? Program.Middle : Program.End
13:     for all i in Start..End do
14:         Program.ReduceStatement(i)  ▷ Calls ReduceStatement on body
15: endmethod
```

cause of a vulnerability can be hard to distinguish. To determine which parts of the inputs to an HLS compiler are responsible for the synthesis of a timing side-channel vulnerability, we enhance TurboTurtle with a reduction mechanism. This reduction mechanism is split into two main parts: C statement reduction and HLS compiler configuration reduction. The goal of the C statement reduction is to automatically reduce a randomly-generated C program into a reduced form that still triggers the vulnerability. We gradually omit statements, re-synthesize and re-verify and thus detect whether they had contributed to the generation of vulnerable HLS output. The HLS compiler configuration reduction determines which HLS pragmas, directives or arguments contribute to the generation of vulnerable HLS output.

7.1 C statement reduction

Algorithm 2 describes our reduction algorithm to reduce the size of a vulnerable program. (Line 2–7): We iteratively remove the tail and head half of the program, re-synthesize and re-verify. When removing a part of the program also removes the vulnerability from the HLS output, we add this part back and continue reducing the other half until no further reductions are possible without also removing the vulnerability. We demonstrate the algorithm with an example.

Head reduction. (Line 6): Program.ReduceStatements(Head), calls the method ReduceStatements (Line 10–15), which iterates through the first half of the non-declarative statements (e.g., assignment and CF statements), removes them, and uses the function arguments to replace the identifiers whose assignments got removed. In case it encounters a compound statement, e.g., a loop, it recursively calls the Program.ReduceStatements(Tail) for the body. Consider the original program that has not yet been reduced, shown in Listing 1. This program has a vulnerable `if` statement whose condition is a secret variable. Calling Program.ReduceStatements(Head) returns the reduced program shown in Listing 2.

In this example, the assignments to `var1`, `var2`, `var3_secret` are removed by Program.ReduceStatements(Head). To maintain the secrecy of the variables used in the statements, for those variables whose assignments are removed by the head reduction, the references to public variables are replaced with public function

```
int original (int arg0, int arg1, int arg2_secret,
    int arg3_secret) {
  int var1 = arg0; int var2 = arg1; int var4 = arg0;
  int var3_secret = arg2_secret;
  int var5_secret = arg3_secret;
  var1 = arg0 + arg1;
  var2 = arg1 * var1;
  var3_secret = arg2_secret + arg3_secret;
  // Vulnerable line that we aim to automatically locate:
  if (var3_secret) {var4 = var1 + var2;}
  var5_secret = var3_secret - arg3_secret;
  return var4 + var5_secret; }
```

Listing 1: Original vulnerable program.

```
int head_reduced (int arg0, int arg1, int arg2_secret,
    int arg3_secret) {
  int var4 = arg0; int var5_secret = arg3_secret;
  /* The upper half of the statements is removed */
  // Vulnerable line that we aim to automatically locate:
  if (arg2_secret) {var4 = arg1 + arg1;}
  var5_secret = arg2_secret - arg3_secret;
  return var4 + var5_secret; }
```

Listing 2: Head-reduced program is still vulnerable.

```
int tail_reduced (int arg0, int arg1, int arg2_secret,
    int arg3_secret) {
  int var4 = arg0; int var5_secret = arg3_secret;
  /* The upper half of the statements is removed */
  // Vulnerable line that we aim to automatically locate is gone
  /* The lower half of the statements is removed */
  return arg0 + arg2_secret; }
```

Listing 3: Tail-reduced program is not vulnerable anymore.

arguments (e.g., all the occurrences of `var1` are replaced by `arg1`), and all the secret variables are replaced with secret function arguments (e.g., all the occurrences of `var3_secret` are replaced with `arg2_secret`). Explicit casts are inserted to match the type correctness. Additionally, all the dead variable declarations (no uses) are removed. Note that after this reduction, Algorithm 2 obtains a legal program that still contains a vulnerable statement that consumes a secret variable. Since the program is still vulnerable, we continue to search for an even smaller program.

Tail reduction. Similarly, Program.ReduceStatements(Tail) removes the lower half of the statements in the function. The tail reduction is not allowed to remove the return statement. When a variable assignment and its declaration are removed, Program.ReduceStatements(Tail) replaces all its references with function arguments, while preserving syntactic correctness and the configured CT rules. For example, Program.ReduceStatements(Tail) generates the program shown in Listing 3 from the program in Listing 2. In this case, although the resulting program still compiles, the timing side-channel in the genrated RTL no longer exists, and Algorithm 2 stops searching this half of the program. Thus, the reduced program shown in Listing 2 is returned.

7.2 HLS compiler configuration reduction

We further determine vulnerable HLS argument combinations. Given a reduced C program and a random HLS compiler configuration that results in vulnerable HLS output, TurboTurtle determines the compiler arguments that contribute to the generation of a timing side channel. We call these arguments *effective arguments*. To

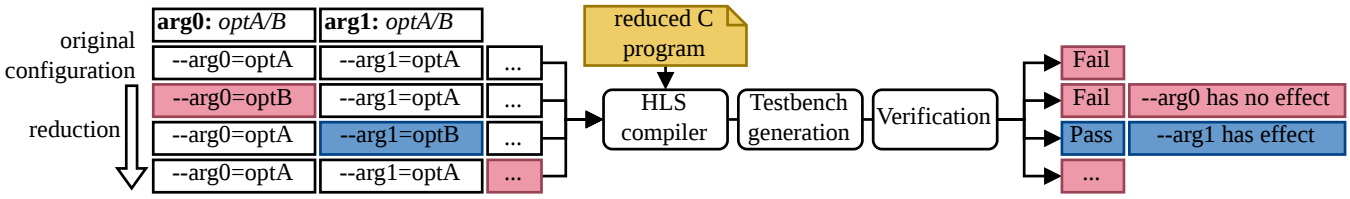

Figure 5: Given a reduced C program and a random HLS compiler configuration, TurboTurtle determines effective HLS configuration options by iteratively changing the options. If a changed option for an HLS argument results in a change in the test status, the argument has an effect. Here, `--arg0` has no effect, as the changed option did not change the test status. `--arg1` has an effect, as changing the option results in a test status change. `optB` is necessary for the side channel.

achieve this, TurboTurtle iterates through all arguments one-by-one and changes their values. After each change, TurboTurtle restarts the HLS process with the same reduced C program and re-tests the RTL output to see if the update causes a change of the test status. If so, TurboTurtle determines that the altered HLS argument affects the side channel introduction. If no *effective* argument was found, TurboTurtle finally tests the default arguments.

Exploiting bug locality. Based on the idea of locality of bugs [66], we assume that, if we found an effective argument value that influenced the generation of a timing side channel, it is likely that other values of the same argument could cause a side channel as well. While further fuzzing would eventually reveal other effective argument values, we can speed up the process by reusing the already minimized C program and brute forcing all effective argument value combinations. This allows us to identify and merge overlapping CT rules if differing HLS compiler configurations together with the same input program lead to vulnerable HLS compiler output.

8 Evaluation

Our evaluation answers the following questions: (1) What is the most effective verification method to find many vulnerabilities? (Section 8.1); (2) What is an optimal test program length? (Section 8.2); (3) Are CT *Rule #1* and *Rule #2* effective at mitigating timing side channels in HLS output? (Section 8.3); (4) Do HLS compilers preserve CT properties of CT programs? (Section 8.4); (5) Do timing side-channel vulnerabilities introduced by HLS occur in real-world scenarios? (Section 8.5)

Supported HLS compilers. TurboTurtle supports four state-of-the-art HLS compilers: Bambu 2024.10 [28], Dynamatic `#9204faf2` [1], commercial tool A, and commercial tool B. We anonymize the commercial tool names due to license agreements.

Simulators and tools. We use Verilator v5.040 [82] as our default simulator, because it is open source. However, we use a commercial simulator when testing proprietary HLS tool chains since they do not provide pre-compiled simulation binaries for their intellectual property (IP), which are needed by Verilator. When parsing the generated RTL designs for their input and output ports or detecting missing modules, we rely on Slang v8.1.0 [70].

Fuzzing configuration. We configure TurboTurtle to reset the RTL design, start simulation, and then invoke the design's behavior four times without a reset. As long as TurboTurtle does not detect a timing side-channel vulnerability, it restarts the testbench 5000 times with a fresh reset and new random inputs. Upon detecting a timing side channel, it stops the verification cycle early and outputs the simulation traces.

Description	Formal	Simulation
Time limit	6h	6h
Total RTL designs verified	68	1092
Average verification duration for one design	5min 17s	20s
Total vulnerable designs	12	114

Table 2: The results of verifying HLS designs using formal verification and simulation in the same amount of total verification time.

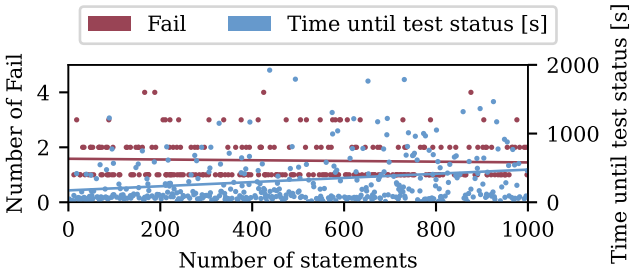

Figure 6: Determining effective program lengths. We compare the number of detected vulnerable designs (Fail) (left y-axis) and the time until test status, which is the sum of HLS compilation, verilation/building, and verification duration (right y-axis) for different numbers of statements (x-axis).

8.1 Simulation vs. formal verification

We first compare the performance of verifying the generated RTL designs using formal verification versus simulation with random data inputs. Table 2 compares how many designs we can test either with formal verification or simulation, given a time limit of 6 hours. With simulation, we can evaluate roughly 16× more designs and find roughly 10× more bugs. Thus, we decided to continue our evaluation using simulation. Simulation is also advantageous concerning design initialization and simulation libraries, as HLS flows typically provide artifacts for these cases for simulation only.

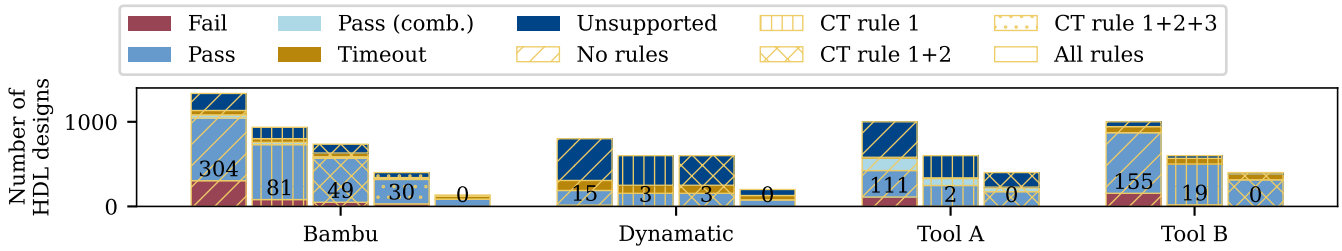

Figure 7: We configure CTurtle to generate random C programs with progressively more CT rules enabled. *Fail*: TurboTurtle detected a vulnerable RTL design. *Pass*: TurboTurtle did not discover a vulnerability during verification. *Pass (comb.)*: the HLS output is combinational. *Timeout*: one of the tasks in the HLS to verification workflow required more than 30 minutes to complete. *Unsupported*: TurboTurtle encountered an error during HLS or RTL elaboration because of unsupported HLS input or output. All data types are integer types.

8.2 Program generation strategies and length

To identify a good C program length for discovering timing side channels, we configure CTurtle to generate 1800 non-CT C programs with lengths of 1 to 1000 statements. TurboTurtle synthesizes each C program with Bambu, Tool A and Tool B. Figure 6 evaluates the impact of the program length. The data indicates a small trend towards smaller programs resulting in more vulnerable RTL designs, and longer input programs tend to take longer to synthesize and verify. Thus, we choose to fuzz with a tendency for small programs to increase the fuzzing throughput. In Appendix Section A.4, we compare the effects of the different program generation strategies discussed in Section 5.1. Surprisingly, we discover that the random data flow generation strategy together with separated secret and public data flows produces the most RTL designs with a timing side-channel vulnerability. However, these designs had no CT rules active and thus the results might not be representative for all cases. To increase program diversity, we fuzz with randomly selected data flow strategies, with a slight bias towards the 'random, separated' strategy.

8.3 Effectiveness of CT rules

We launch multiple fuzzing campaigns with progressively more rules enabled and present our results in Figure 7. We use CTurtle to generate 800 C programs with integer data types only and synthesize them with Bambu, Tool A and Tool B. We generate and test a separate set of 800 C programs for Dynamatic, because at the time of writing Dynamatic does not support `long` and `long long` integer data types, sub-function calls, arrays defined inside the top-function body, and the modulo operation. Further, it only supports division for the `int` data type. Interestingly, in many cases, non-CT input programs lead to CT RTL (based on fuzzing), which could be of interest to study further when designing mitigations. This is our first takeaway:

T1. In many cases, HLS compilers may convert non-CT input programs into CT RTL automatically.

We notice that avoiding secret-dependent control flow (second bar per tool in Figure 7) significantly reduces timing side-channel vulnerabilities in the design output for each of the four HLS flows. The number of side channels caused by secret-dependent memory accesses is lower (third bar per tool in Figure 7), likely because TurboTurtle currently does not include external memories, as discussed in Section 9. We conclude that for two of the four HLS compilers, following *Rule #1* and *Rule #2* is an effective measure to avoid generating vulnerable hardware. However, when fuzzing Bambu and Dynamic with these rules active, we *still find RTL outputs that exhibit secret-dependent timing variation*. These cases constitute timing side-channel *vulnerabilities* that we discuss in Section 8.4. After finding vulnerabilities 1 and 2, we implement a custom CT *Rule #3* for Bambu that forbids integer division and modulo in combination with the arguments that we found to be vulnerable (see Table 5) and re-run TurboTurtle. We find another 30 vulnerable designs. In total, we generate additional 1200 C programs that gradually adhere to more of our discovered Bambu CT rules. When verifying the designs generated from programs that adhere to all our *newly discovered rules* for a tool, as listed in Section A.2.3, Table 5 for Bambu and Table 6 for Dynamatic, we find *no more side channels*.

Programs with floating point types. We test the C float data types separately, as only Bambu and Tool A support them, and present our results in Figure 8. We generate 600 programs that we synthesize with both tools. Similar to the integer-data-type fuzzing results, these results show that following *Rule #1* and *Rule #2* helps mitigate many timing side-channel vulnerabilities. In a third fuzzing run, we generate 200 C programs that adhere to our Bambu-specific *Rule #3*, plus a *Rule #4* that forbids linearization expressions (causing vulnerability 9). This run surfaces the floating point vulnerabilities. Finally, when we activate rules that exclude all vulnerable code constructs and arguments as presented in Table 5 , we find no more side channels.

8.4 Detected vulnerabilities

We present the vulnerabilities we discovered through fuzzing. As there are no established specifications for data-independent timing in HLS flows (i.e., no established *Rule #3 or more*), we consider as vulnerable all non-CT RTL designs synthesized from C programs following *Rule #1* and *Rule #2*. Table 5 and 6 in Appendix A summarize the vulnerabilities and list necessary HLS compiler arguments.

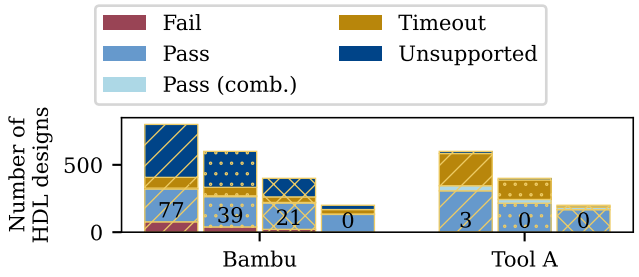

Figure 8: Results with floating point data types. The bars, left to right, are used C programs with: *no rules*, *CT rule 1*, and *CT rule 1 and 2*. Bambu's fourth run uses custom rules that exclude already found vulnerabilities.

```
double secret_array[...] = ...;
float f = (float) secret_array[...];
```

Listing 4: Vulnerability 8 (Bambu): Reading a value from a double array into a float variable, array content leaks.

```
return var_0 * (varSECRET || (var_1 * var_0));
```

Listing 5: Vulnerability 9-10 (Bambu+Dynamatic) - short-circuit conversion: Following the C standard, the execution of the RHS of the logical OR depends on whether the LHS is 1. varSECRET leaks. All variables are int.

Vulnerability 1-2, Bambu integer division/modulo operators. C division and modulo operators with integer data type in combination with certain HLS argument values for `--hls-div` and `compiler` are vulnerable operators in Bambu. A middleend HLS pass of Bambu, called `SoftIntCGExt`, replaces division/modulo operators with function calls to a software division implementation called `divsi3` (or `modsi3`). These functions implement input-dependent CF and are then converted into hardware with a timing side-channel vulnerability.

Vulnerability 3: Bambu float/double casts. *Type casts* from or to float or double are vulnerable in many different compiler configurations, including in the default configuration. In Section 8.5 we show that type punning is equally affected. According to the maintainers, adding the '–pipelining' flag removes the vulnerability. We confirm that we have not found side channels with this flag active.

Vulnerability 4-7 + 12: Bambu float/double division/addition/-subtraction/multiplication/comparison operators. The middleend replaces float/double divisions/additions/multiplications/-subtractions with C functions that operate on `unsigned long long int types`. While these functions do not contain any CF statements, the middleend also inserts code after the function calls, which contains an `if` statement conditioned by the function's result.

Vulnerability 8: Bambu - reading from double array into float. When reading from a secret array of `double` data type into a `float` variable, Bambu's middleend generates secret-dependent CF. Interestingly, this vulnerability is only generated when a specific combination of HLS arguments is used, listed in Table 5. This set of arguments does not introduce any CF statements when compiling only a float-to-double cast, thus it differs from vulnerability 3.

Vulnerability 9-10: Bambu+Dynamatic (CVE-2026-23895) masking. Listing 5 shows a HLL program that triggers

```
return (var_0 / var_1) * (varSECRET < 3232);
```

Listing 6: Vulnerability 11 (Dynamatic): Since the RHS value of the multiplication (i.e., the comparison operator result) is binary, LLVM converts the multiplication into a select instruction dependent on varSECRET. Dynamatic synthesizes the select into secret-dependent CF. All variables are int.

short-circuiting compiler optimizations in GCC and CLANG. If `varSECRET == 0/1`, according to the C standard [2], the right-hand side of a logical AND/OR is not supposed to be evaluated at all, so compilers insert a secret-dependent branch. The branch is synthesized into secret-dependent FSM transitions, causing a timing side channel. This snippet is vulnerable with Dynamatic, and in a longer version also with Bambu. When compiling the snippet for x86, a branch instruction is emitted similarly. Thus, this code construct may even break CT software programs targeted for CPUs.

Vulnerability 11: Dynamatic backend vulnerability (CVE-2026-23894). Listing 6 shows the root-cause of an interesting vulnerability. LLVM's 'instcombine' pass understands that the RHS (comparison with varSECRET) value of the multiplication can only be binary and converts it into a select that chooses LHS or 0 depending on the value of the RHS. This LLVM behavior has been described by Schneider et al. [79] and was found to introduce timing side channels into real-world CT cryptographic code, when compiled to x86 instructions. The LLVM 'select' is, however, not necessarily vulnerable. For example, we do not observe a timing side channel for this code with Bambu or Tools A/B. This is most likely because these tools statically schedule basic blocks. Conversely, Dynamatic aims for higher performance by introducing variable latency. We find that Dynamatic's backend propagates the selected input to the output before the non-selected input arrives. A latency variability can thus be observable if the data inputs arrive at different times, resulting in a timing side channel. Interestingly, the HLL code construct that leads to a side channel (multiplication with a conditional's result) follows a state-of-the-art pattern suggested to linearize the CF of a program to make it CT [19].

Takeaways. Vulnerabilities 1-2 and 4-7 and 9-10 establish unsafe operators for Bambu HLS, leading us to the following takeaway:

T2. Non-CT C operators can exist for HLS compilers.

Vulnerabilities 1-10 are introduced when the HLS compiler front- or middle-end alter the input program's CF. Vulnerability 11 was introduced without modifying this high-level CF. Instead, Dynamatic generates hardware with different execution paths for the same software-level CF. This leads us to our next takeaway:

T3. HLS compilers can insert data-dependent decisions at any stage of the HLS flow, thereby breaking CT programming unexpectedly.

8.5 Case studies

8.5.1 Masking / LLVM constant-time pass. Listing 7 in Section A.2 shows a typical code used by cryptographic libraries for constant-time lookup [3, 4]. LLVM lowers this code into a 'select' statement,

making it vulnerable to Dynamatic's vulnerability 11. As this snippet can also introduce side channels when compiled for CPUs [79], a CT LLVM pass has been proposed that compiles the 'select' into architecture-specific CT instruction sequences. However, this is not an option for HLS, as the C code is not compiled into instructions. Thus, HLS users have no control over the insertion of such side channels. Listing 3 in [79] shows another case where a 'select' is inserted. We find that an adapted version of this listing (shown in Section A.2, Listing 8) is equally vulnerable in Dynamatic. The original listing was based on code found in the Botan crypto library's implementation of the AES-GCM hashing algorithm [79].

8.5.2 Neural network inference operator. We find that floating point conversions cause timing side channels with Bambu and searched for real-world code that performs such conversions on sensitive data. For example, Dynamic Range Quantization is a method where floating-point layer activations are converted to integer during inference [50]. Google's XNNPACK [33] library provides highly optimized floating-point neural network inference operator implementations for various CPU models, and generic C versions of them. We find that the `xnn_f32_raddstoreexpminusmax_ukernel__scalar_rr2_lut64_p2_u1` function performs type punning from float to integer in a statement like shown in Listing 9 in Section A.2 . Our verification setup confirms that this statement causes a similar side channel as float to integer conversion in Bambu. HLS could be used to synthesize and accelerate XNNPACK operator C implementations, for example, as custom RISC-V instructions, or for targeting an embedded FPGA [51, 58]. Accelerating this statement with HLS could lead to timing side channels for the operators.

9 Discussion

We discuss limitations, mitigations and future work.

C language and HLS language features. CTurtle currently supports the majority of the C language. Some C constructs, like, `goto` statements, are implicitly fuzzed because they are used in library functions inserted by the HLS middleend. CTurtle does not yet support structs, single bit manipulation and pointer arithmetic. CTurtle currently does not support HLS language features such as arbitrary precision data types or custom code libraries offered in commercial HLS tools, or domain specific languages such as SystemC [6, 26]. Extending the fuzzer to support C++ and tool-specific language features are interesting directions for future work.

Proving security. If TurboTurtle detects no vulnerability, we consider the verification status of the design as 'passed'. However, as with any fuzzing-based method, non-exhaustive simulation cannot guarantee that no timing side-channel vulnerability exists. Formal methods would be able to prove that a design is side-channel free, but they come with a runtime cost as was shown in Section 8.1.

External Memories. TurboTurtle currently does not support arrays on the top-function interface, because they would be synthesized into HLS tool-specific memory interface protocols. Extending TurboTurtle to support these could be interesting for findnig more memory-related side channels. For example, Dynamatic features a Load-Store Queue (LSQ) for managing memory operations[46]. During manual analysis, we found that secret-dependent accesses to array arguments, i.e., external memory, cause timing side channels due to the LSQ.

Mitigations. While our findings suggest that applying CT rules to the HLL input program can reduce the chance of a side channel in the output RTL, we note that more than half of our generated programs with secret-dependent control flow did *not* lead to a timing side channel in the generated RTL. These examples could give hints for future mitigations. TurboTurtle's results allow the definition of tool-specific CT rules consisting of sets of code constructs and HLS configurations (see Table 5) that should be avoided.

Future research. Compilers have been tested for CT violations in existing codes [31, 79], but have not yet been tested with randomly generated input programs. To find more corner cases, future research could use CTurtle to fuzz CPU compilers regarding breaking CT code. Dynamatic plans to implement speculation into their HLS flow [48], and Elastic HLS [87, 88] implements memory speculation. These cases would be specially interesting to test for timing side channels, given that speculation can cause many CPU vulnerabilities [14, 34, 52, 77, 91, 93–96].

10 Related Work

We discuss related work in the field of high-level synthesis security research and compiler fuzzing.

Random C program generation. Csmith [97] and GrayC [27] generate random C/C++ programs intended for compiler fuzzing. Focusing on functional correctness, they lack various capabilities that we need for fuzzing HLS compilers for timing side channels. They do not support CT rules, do not track secret and public data flows and thus also do not provide enough information for our CT-preserving program reduction. Revizor generates random binaries for x86 CPUs, applying some CT rules [64]. Whisperfuzz and MileSan generate random constant-time RISC-V assembly programs [18, 56]. Grammar-based program generation has been used to test programming languages [83]. Methods exist for automatic HLS pragma insertion, aiming for optimal placement [13, 72]. As we aim for bug detection, we place them randomly instead.

Verifying HLS compilers. Herklotz et al. [36] fuzz four HLS compilers to check for functional equivalence between the HLL input and RTL output. They exclude floats and do not fuzz HLS compiler arguments. Others formally verify HLS source-to-source transformations [71] and the full HLS process [37]. All these methods focus on behavioral correctness and are unable to detect timing side channels because they verify against an untimed HLL input code.

Detecting timing side channels in HLS output. Steffen et al. [69] manually crafted an example where a software timing side-channel is avoided in the input program using branch balancing, but the HLS compiler introduces a timing side channel into the generated hardware design. Hu et al. [41] augment HLS-generated designs with Register Transfer Level IFT (RTLIFT) logic [8] and use formal verification to detect data-dependent timing.

Software compilers breaking CT. Software compilers have been tested against breaking the CT property of cryptographic libraries [29, 31, 79, 100]. In many cases, compilers broke the programs' CT property. We manually tested some of their listings with Bambu and Dynamatic and found that Dynamatic introduced vulnerability 11 into an expanded version of listing 3 [79]. Often,

previous work generates CPU-specific instruction sequences with secret-dependent timing [79]. As in the case of HLS, since the C code is not compiled into CPU instructions, we did not find the same violations.

Mitigations. Peter et al. [68] propose mitigations that balance the hardware control flow for labeled control signals in hardware generated from non-CT input programs. Jiang et al. [45] track information flow from secret inputs through the design and balance all secret-dependent branches, thus enforcing constant timing behavior at public output ports. They currently support only non-pipelined designs [45]. [69] use bounded model checking on the generated schedule to detect timing side channels. SecHLS implements security constraints into scheduling and binding algorithms [80].

Other HLS security research. The following relates to HLS security but does not aim at detect timing side-channels. Pundir et al. [73], Konigsmark et al. [55], and Zhang et al. [99] detect power side-channel vulnerabilities when synthesizing cryptographic hardware such as AES-128 using industry standard HLS tools. [73] finds that CT programming does not protect against intermediate value leakage. Muttaki et al. [62, 63] and Pundir et al. [73] discover security vulnerabilities in HLS related to unbalanced pipelines, uncleared I/O registers, and information leakage for combinational circuits based on manual analysis of output designs. Ibnat et al. [42] present examples of non-secure resource sharing in HLS-generated designs.

11 Conclusion

We performed the first automated security analysis of HLS with respect to timing side channels for two open-source (Bambu and Dynamatic) and two commercial HLS flows. Our results show that Bambu and Dynamatic can introduce 12 distinct timing side channels, even when the input C programs use generic CT rules that avoid secret-dependent memory accesses or branching decisions. We discover eight insecure operators with data-dependent timing in Bambu or Dynamatic, which should not be used with secret arguments. To obtain these results, we designed and implemented TurboTurtle, the first fuzzer that is capable of automatically detecting timing side channels produced by HLS flows. TurboTurtle achieves this by randomly generating valid C programs that are constant-time with respect to input data that is marked as secret, following configurable CT rules. To detect timing side channels, TurboTurtle relies on a miter construction that can distinguish when two instantiations of the same hardware design, generated by the target HLS flow, exhibit variable latency under differing inputs. We will release TurboTurtle as open source to enable automated testing of HLS flows against timing vulnerabilities.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, as well as Carmine Rizzi and Emmet Murphy for their advice and input regarding HLS tools and benchmarks. This work has been supported by a Qualcomm Innovation Fellowship and the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE), as well as by the Swiss National Science Foundation (grant number 215747).

12 Ethical Considerations

12.1 Stakeholders

High-level hardware designers. Respect for Persons: Following a deontologist view, HLL programmers who use HLS tools have the right to know about the potential security vulnerabilities introduced by HLS tools. Beneficence: Thus, they benefit from our research, which makes them aware of the risks and enables them to evaluate the hardware generated from their code. Justice: Our research neither favors nor discriminates any HLL programmer, because we use randomly generated code to identify vulnerabilities and thus do not blame any HLL programmer. Respect for Law and Public Interest: We do not investigate any source code that we obtain via license agreements. All code that we publish has been published as open source or was generated by our tool or the HLS tools.

HLS tool developers/vendors. Respect for Persons: Our research does not affect individuals, but may have an impact on organizations that sell or develop HLS tools. We obtained our results without requiring the use of proprietary and protected information. Beneficence: HLS tool developers and vendors benefit from our research, because we make them aware of the security risks produced by their products. Since HLS tool vendors do not claim to provide security guarantees yet, they will not experience financial harm through our research. Furthermore, we reported all vulnerabilities before publication and thus HLS tool developers were able to respond as desired. Tool developers/vendors might see our results as bad for their reputation. However, since they did not make any security claims so far, they cannot be held responsible for the vulnerabilities, and thus there is no reputational damage. Taking our research into account, developers/vendors are now able to decide whether to provide security guarantees or not. Justice: To avoid blaming a single HLS tool developer/vendor, we conduct our research for four tools. Respect for Law and Public Interest: To the best of our knowledge, our method does not break the law or compromise public interest in any way.

Security research community. Respect for Persons/Beneficence: Researchers benefit through the open sourcing of our results. Justice: Open sourcing also avoids any future advantage that we could obtain over other researchers by withholding our tools and results. Respect for Law and Public Interest: We publish everything that licenses allow.

Society at large. Respect for Persons/Justice: Our research does not involve data of individuals. Beneficence: It benefits society at large, because we open source tools for testing any HLS generated code against timing side channels. Respect for Law and Public Interest: We publish everything that licenses allow.

12.2 Mitigations

Malicious actors may make use of our findings to write malicious C libraries targeted for HLS. However, malicious actors may have found such vulnerabilities already without publishing them.

We mitigate harm through our research through responsible disclosure of all vulnerabilities to the respective tool vendors or maintainers, giving them time to respond and address the issues that we have discovered.

12.3 Decision

Weighing the ethical harms against ethical benefits of our research led us to decide the following: Not conducting our research, despite having obtained initial results that show that HLS tools might generate vulnerable RTL, would cause harm to the society at large, because vulnerable hardware might be distributed unknowingly. With our tool we enable HLS programmers, tool vendors, users of HLS-generated hardware and security researchers to find side channels that were knowingly or unknowingly introduced into HLS generated circuits.

Generative AI Usage

This paper was partially checked for grammar using ChatGPT [65]. GitHub Copilot in VS Code [32] was used for auto-completion of the code implemented by some of the authors. The randomized backtracking algorithm for argument selection was generated by Claude Sonnet 4.5, based on a hand-written functional version that implemented rejection sampling. We manually inspected, adapted and tested all generated code for correctness.

References

[1] 2024. Dynamatic. EPFL LAP.

[2] 2025. ISO/IEC 9899:1999. https://www.iso.org/standard/29237.html.

[3] Julius Alexandre. [n. d.]. *[ConstantTime][LLVM] Add llvm.ct.select intrinsic with generic SelectionDAG lowering*. Accessed: 2026-01-14. https://github.com/llvm/llvm-project/pull/166702

[4] Julius Alexandre. 2025. *Introducing constant-time support for LLVM to protect cryptographic code*. Accessed: 2026-01-14. https://blog.trailofbits.com/2025/12/02/introducing-constant-time-support-for-llvm-to-protect-cryptographic-code/

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 53–70.

[6] AMD. 2025. AMD Vitis™ HLS. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 623–639.

[8] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. [n. d.]. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*.

[9] Various authors. [n. d.]. *Csmith*. Accessed: 2025-11-13. https://github.com/csmith-project/csmith

[10] Various authors. 2025. Vitis Libraries. Accessed: 2025-11-10. https://github.com/Xilinx/Vitis_Libraries/tree/main

[11] Various authors. 2025. *WolfSSL*. Accessed: 2025-11-10. https://github.com/wolfSSL/

[12] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. 2013. Formal Verification of Side-Channel Countermeasures Using Self-Composition. *Science of Computer Programming* 78, 7 (July 2013), 796–812. doi:10.1016/j.scico.2011.10.008

[13] Yunsheng Bai, Atefeh Sohrabizadeh, Zijian Ding, Rongjian Liang, Weikai Li, Ding Wang, Haoxing Ren, Yizhou Sun, and Jason Cong. 2024. Learning to compare hardware designs for high-level synthesis. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. 1–7.

[14] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 971–988.

[15] Claudio Barone, Rishika Kushwah, Ankur Limaye, Vito Giovanni Castellana, Giovanni Gozzi, Michele Fiorito, Fabrizio Ferrandi, and Antonino Tumeo. 2024. To Cache or Not to Cache? Exploring the Design Space of Tunable, HLS-generated Accelerators. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24)*. Association for Computing Machinery, New York, NY, USA, 210–218. doi:10.1145/3695794.3695815

[16] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1267–1279. doi:10.1145/2660267.2660283

[17] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying Constant-Time Implementations by Abstract Interpretation. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 260–277.

[18] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. WhisperFuzz: White-box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors. (2024).

[19] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 715–733. doi:10.1145/3460120.3484583

[20] Katharina Ceesay-Seitz, Flavien Solt, Alexander Klukas, and Kaveh Razavi. 2025. Pathfinder: Constructing Cycle-accurate Taint Graphs for Analyzing Information Flow Traces. In *ICCAD 2025*.

[21] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. (2024).

[22] Sudipta Chattopadhyay and Abhik Roychoudhury. 2018. Symbolic Verification of Cache Side-Channel Freedom. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2812–2823. doi:10.1109/TCAD.2018.2858402

[23] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2022. Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, New York, NY, USA, 727–732. doi:10.1145/3489517.3530981

[24] S. Dinesh, M. Parthasarathy, and C. Fletcher. 2024. CONJUNCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety against Microarchitectural Timing Attacks. In *IEEE S&P '24* (Los Alamitos, CA, USA, 2024-05). IEEE Computer Society, 176–176. https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00180

[25] S. Dinesh, Y. Yhu, and C. Fletcher. [n. d.]. H-HOUDINI: Scalable Invariant Learning. In *ASPLOS 2025*.

[26] Siemens EDA. 2025. Catapult High-Level Synthesis and Verification. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[27] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. doi:10.1145/3597926.3598130

[28] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1327–1330. doi:10.1109/DAC18074.2021.9586110

[29] Antoine Geimer and Clementine Maurice. 2025. Fun with flags: How Compilers Break and Fix Constant-Time Code. *arXiv preprint arXiv:2507.06112* (2025).

[30] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1690–1704. doi:10.1145/3576915.3623112

[31] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. 2025. Do Compilers Break Constant-time Guarantees? https://cispa.de/de/research/publications/84593-do-compilers-break-constant-time-guarantees-.

[32] GitHub. 2026. *GitHub Copilot in VS Code*. Accessed: 2026-01-05. https://code.visualstudio.com/docs/copilot/overview

[33] Google. [n. d.]. *XNNPACK*. Accessed: 2026-01-14. https://github.com/google/XNNPACK

[34] Jean-Claude Graf, Sandro Rüegge, Ali Hajiabadi, and Kaveh Razavi. 2026. VMScape: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments. In *2026 IEEE Symposium on Security and Privacy (SP)*.

[35] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. [n. d.]. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS 2020*.

[36] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of High-Level Synthesis Tools. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 219–223. doi:10.1109/FCCM51124.2021.00034

[37] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Vericert* 5, OOPSLA (Oct. 2021), 117:1–117:30. doi:10.1145/3485494

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

[39] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P. Mulligan, Gustavo Petri, Christopher W. Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *MICRO*.

[40] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Comput. Surv.* 54, 4 (May 2021), 83:1–83:39. doi:10.1145/3447867

[41] Wei Hu, Armaiti Ardeshiricham, Lingjuan Wu, and Ryan Kastner. 2022. Integrating Information Flow Tracking into High-Level Synthesis Design Flow. In *Behavioral Synthesis for Hardware Security*, Srinivas Katkoori and Sheikh Ariful Islam (Eds.). Springer International Publishing, Cham, 365–387. doi:10.1007/978-3-030-78841-4_16

[42] Zahin Ibnat, Hadi Mardani Kamali, and Farimah Farahmandi. 2023. Iterative Mitigation of Insecure Resource Sharing Produced by High-level Synthesis. In *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 1–6. doi:10.1109/DFT59622.2023.10313550

[43] Intel Corporation. 2025. *Data Operand Independent Timing ISA Guidance*. Accessed: 2025-11-10. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[44] Jan Jancar, Marcel Fourne, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. [n. d.]. "They're Not That Hard to Mitigate": What Cryptographic Library Developers Think about Timing Attacks. In *IEEE S&P 2022*.

[45] Zhenghong Jiang, Steve Dai, G. Edward Suh, and Zhiru Zhang. 2018. High-Level Synthesis with Timing-Sensitive Information Flow Enforcement. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1145/3240765.3240813

[46] Lana Josipović, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Transactions on Embedded Computing Systems* 16, 5s (Sept. 2017), 125:1–125:19.

[47] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. doi:10.1145/3174243.3174264

[48] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 162–171. doi:10.1145/3289602.3293914

[49] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 7 (July 2022), 2142–2155. doi:10.1109/TCAD.2021.3105574

[50] Alan Kelly. [n. d.]. *Faster Dynamically Quantized Inference with XNNPack*. Accessed: 2025-11-14. https://blog.tensorflow.org/2024/04/faster-dynamically-quantized-inference-with-xnnpack.html

[51] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. 2021. FABulous: An Embedded FPGA Framework. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 45–56. doi:10.1145/3431920.3439302

[52] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. arXiv:1801.01203 doi:10.48550/arXiv.1801.01203

[53] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.). Springer, Berlin, Heidelberg, 104–113. doi:10.1007/3-540-68697-5_9

[54] David Kohlbrenner and H. Shacham. 2017. On the Effectiveness of Mitigations against Floating-Point Timing Channels. In *USENIX Security Symposium*.

[55] S.T. Choden Konigsmark, Deming Chen, and Martin D.F. Wong. 2017. High-Level Synthesis for Side-Channel Defense. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 37–44. doi:10.1109/ASAP.2017.7995257

[56] Tobias Kovats, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2025. MileSan: Detecting Exploitable Microarchitectural Leakage via Differential Hardware-Software Taint Tracking. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*.

[57] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[58] Jindong Li, Tenglong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. 2025. Pushing up to the Limit of Memory Bandwidth and Capacity Utilization for Efficient LLM Decoding on Embedded FPGA. In *2025 Design, Automation & Test in Europe Conference (DATE)*. 1–7. doi:10.23919/DATE64628.2025.10993087

[59] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: A Survey. *Cybersecurity* 1, 1 (June 2018), 6. doi:10.1186/s42400-018-0002-y

[60] ARM Ltd. [n. d.]. *DIT, Data Independent Timing*. Accessed: 2025-11-10. https://developer.arm.com/documentation/ddi0601/2022-06/AArch64-Registers/DIT--Data-Independent-Timing

[61] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Replication Package for "Compiler Fuzzing: How Much Does It Matter?"* 3, OOPSLA (Oct. 2019), 155:1–155:29. doi:10.1145/3360581

[62] Md Rafid Muttaki, Zahin Ibnat, and Farimah Farahmandi. 2022. Secure by Construction: Addressing Security Vulnerabilities Introduced during High-Level Synthesis: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1371–1374. doi:10.1145/3489517.3530674

[63] M. Rafid Muttaki, Nitin Pundir, Mark Tehranipoor, and Farimah Farahmandi. 2021. Security Assessment of High-Level Synthesis. In *Emerging Topics in Hardware Security*, Mark Tehranipoor (Ed.). Springer International Publishing, Cham, 147–170. doi:10.1007/978-3-030-64448-2_6

[64] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *ACM ASPLOS '22* (2022).

[65] OpenAI. 2026. *ChatGPT*. Accessed: 2026-01-05. https://chatgpt.com/

[66] Thomas J. Ostrand and Elaine J. Weyuker. 2002. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 55–64. doi:10.1145/566172.566181

[67] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES *(CT-RSA'06)*.

[68] Steffen Peter and Tony Givargis. 2016. Towards a Timing Attack Aware High-Level Synthesis of Integrated Circuits. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 452–455. doi:10.1109/ICCD.2016.7753326

[69] Steffen Peter and Tony Givargis. 2022. Generation and Verification of Timing Attack Resilient Schedules During the High-Level Synthesis of Integrated Circuits. In *Behavioral Synthesis for Hardware Security*, Srinivas Katkoori and Sheikh Ariful Islam (Eds.). Springer International Publishing, Cham, 343–363. doi:10.1007/978-3-030-78841-4_15

[70] Michael Popoloski. 2025. MikePopoloski/Slang.

[71] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/3626202.3637563

[72] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. 2025. Automatic Hardware Pragma Insertion in High-Level Synthesis: A Non-Linear Programming Approach. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 26 (Feb. 2025), 44 pages. doi:10.1145/3711847

[73] Nitin Pundir, Sohrab Aftabjahani, Rosario Cammarota, Mark Tehranipoor, and Farimah Farahmandi. 2022. Analyzing Security Vulnerabilities Induced by High-level Synthesis. *J. Emerg. Technol. Comput. Syst.* 18, 3 (Jan. 2022), 47:1–47:22. doi:10.1145/3492345

[74] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2016. Secure, Precise, and Fast {Floating-Point} Operations on x86 Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. 71–86.

[75] S. Ravi and M. Joseph. 2016. Open Source HLS Tools: A Stepping Stone for Modern Electronic CAD. In *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. 1–8. doi:10.1109/ICCIC.2016.7919615

[76] RISC-V. 2024. *RISC-v Cryptography Extension*. Accessed: 2025-11-10. https://github.com/riscv/riscv-isa-manual/releases/tag/20240411

[77] Sandro Rüegge, Johannes Wikner, and Kaveh Razavi. 2025. Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions. In *USENIX Security*.

[78] Alexander Schaub. 2020. *Formal Methods for the Analysis of Cache-Timing Leaks and Key Generation in Cryptographic Implementations*. Ph. D. Dissertation. Institut Polytechnique de Paris.

[79] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. 2025. Breaking Bad: How Compilers Break Constant-Time Implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications*

Security. 1690–1706. arXiv:2410.13489 [cs] doi:10.1145/3708821.3733909
[80] Shang Shi, Nitin Pundir, Hadi M Kamali, Mark Tehranipoor, and Farimah Farahmandi. 2023. SecHLS: Enabling Security Awareness in High-Level Synthesis. In *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 585–590.
[81] Frans Sijstermans and Jc Li. 2019. NVIDIA Closes Design Complexity Gap with HLS. https://resources.sw.siemens.com/en-US/white-paper-working-smarter-not-harder-nvidia-closes-design-complexity-gap-with-hls/.
[82] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. 2025. Verilator.
[83] Mathias Soeken and Rolf Drechsler. 2013. Grammar-based program generation based on model finding. In *2013 8th IEEE Design and Test Symposium*. 1–5. doi:10.1109/IDT.2013.6727084
[84] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security 2024*.
[85] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. {CellIFT}: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in {RTL}. In *31st USENIX Security Symposium (USENIX Security 22)*. 2549–2566.
[86] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *USENIX Security 2025*.
[87] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. A High-Frequency Load-Store Queue with Speculative Allocations for High-Level Synthesis. In *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 115–124.
[88] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2025. Compiler Support for Speculation in Decoupled Access/Execute Architectures. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*. 192–204.
[89] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 109–120. doi:10.1145/2528521.1508258
[90] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. 2022. Fuzzing Hardware Like Software. In *USENIX Security 2022*.
[91] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7303–7320.
[92] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2128–2142. doi:10.1145/3576915.3623192
[93] Sander Wiebing and Cristiano Giuffrida. 2025. Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*. 3599–3616.
[94] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3825–3842.
[95] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 89–89. doi:10.1109/SP61157.2025.00089
[96] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto ON Canada, 2023-10-28). ACM, 49–61.
[97] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. doi:10.1145/1993316.1993532
[98] Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. 2024. Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures. *ACM Comput. Surv.* 56, 7 (April 2024), 178:1–178:40. doi:10.1145/3645109
[99] Lu Zhang, Wei Hu, Armaiti Ardeshiricham, Yu Tai, Jeremy Blackstone, Dejun Mu, and Ryan Kastner. 2018. Examining the Consequences of High-Level Synthesis Optimizations on Power Side-Channel. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1167–1170. doi:10.23919/DATE.2018.8342189
[100] Zhiyuan Zhang and Gilles Barthe. 2025. CT-LLVM: Automatic Large-Scale Constant-Time Analysis. *Cryptology ePrint Archive* (2025).

A Appendix

A.1 Existing software CT verification tools

Table 3 compares existing tools that evaluate C code against constant-time rules at C level and discusses the challenges of using them for our purpose of CT C code generation. Even the most complete CT checker, STAnalyzer [78], can only check a C program against CT *Rule #1* and *Rule #2* which is not sufficient when considering HLS-specific rules.

A.2 Vulnerabilities and bugs

A.2.1 Masking. Listing 7 implements a constant-time lookup typically used by cryptographic libraries to avoid secret-dependent CF [4]. Listing 8 is an adapted version of Listing 3 of [79]. For demonstration, in both listings we add a multiplication at the beginning of the snippet to add latency for the calculation of one of the selected values. Without this additional multiplication, we do not see the side channel, but we still see the select in Dynamatic's MLIR dump. Note that integer multiplication with non-binary operands is not vulnerable in Dynamatic.

A.2.2 Type punning. With Bambu, type punning, which interprets the bits of one data type as another data type, causes a similar timing side channel like floating point conversion. Listing 9 shows a vulnerable snippet taken out of Google's XNNPACK library [33].

A.2.3 Vulnerabilities. Table 5 details the vulnerable combinations of operators and HLS compiler configurations for Bambu. Table 6 details the vulnerable combinations of operators and HLS compiler configurations for Dynamatic.

A.2.4 Reported functional bugs. We report crashes and bugs not related to timing side channels to the respective developers and vendors of the HLS compilers we test, with the hope of helping the continuous improvement of the tools. For Bambu 2024.10, we report seven issues, and for Verilator v5.040 we report a single issue. For Dynamatic, we report more than fifteen issues.

A.3 Directives and Pragmas

In Table 4, we list all HLS arguments TurboTurtle supports for Bambu `v2024.10`. Due to licensing restrictions we cannot list the pragmas and directives of the commercial tools. TurboTurtle supports all of them.

A.4 Program generation strategies

Figure 9 compares the different program generation strategies, using the same 1800 designs as in Figure 6. Against intuition, separate data flows and completely randomly generated programs lead to the most discovered side channels.

Table 3: Comparison of software constant-time verification tools that determine whether C code is CT. We list the reasons why we opted against using the tool and instead built a CT C program generator. STAnalyzer is the only tool that might have been extendable.

Tool	Evaluated in	Reason for not using it
ABPV13 [12]	[5],[44],[30]	manual, unscalable, unavailable
BPT17 [17]	[44],[30]	unscalable, unavailable
VirtualCert [16]	[30]	unscalable
CacheFix [22]	[30]	memory CT checks (Rule 1) only
STAnalyzer [78]	[30]	Rule 1 and 2 only
ABSynthe [35]	[30]	requires timing information per instruction

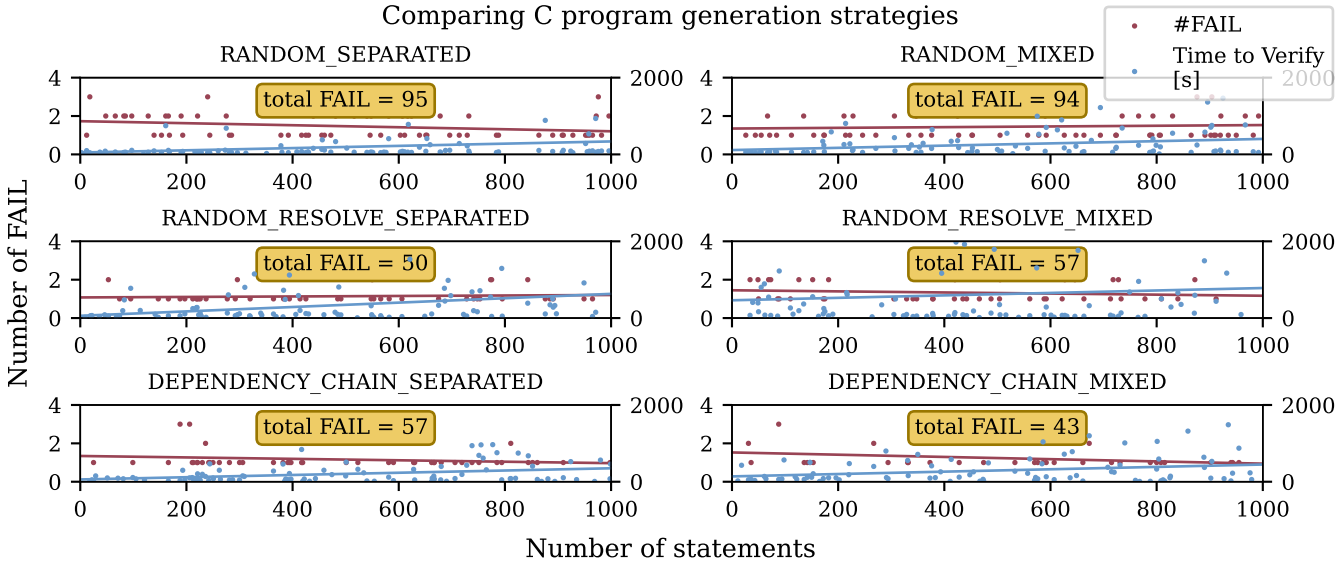

Figure 9: Program generation strategies. We compare the different data flow generation and mixing strategies of CTurtle, as discussed in Section 5.1. We plot the number of statements on the x-axis, the number of detected vulnerable designs (CEX) on the left y-axis and the fuzzing throughput in terms of number of statements per second on the right y-axis. No CT rules are active.

```
uint64_t table, result;
for (uint64_t i = 0; i < 8; i++) {
    table = table * var_0_secret; // added for latency
    const bool cond = i == var_0_secret;
    const uint64_t mask = (-(int64_t)cond);
    result |= (table & mask) | (0 & ~mask); // leaks
}
```

Listing 7: CT select. Fallback implementation for CT LLVM pass when no assembly implementation is available[4]. Triggers vulnerability 11.

```
// originally: const uint64_t R = 0xE100000000000000;
uint64_t R = var_0 * var_0; // added for latency

const uint64_t carry = R * (var_1_secret & 1); // leaks
return (var_0 >> 1) ^ carry;
```

Listing 8: LLVM IR 'Select' generated from code based on Botan's AES-GCM code[79]. Triggers vulnerability 11.

```
inline static uint32_t float_as_uint32(float f) {
    union {
      float as_float;
      uint32_t as_uint32;
    } bits = {f};
    return bits.as_uint32;
  }
  uint32_t ve = (float_as_uint32(vn) & UINT32_C(0
     xFFFFFFC0)) << 17;
```

Listing 9: Type punning. Type punning interprets the binary representation of a float as integer. It leads to vulnerable RTL, similar as float conversion (vulnerability 3). This code from XNNPACK [33] leaks 'vn', derived from machine learning model parameters.

Name	Options
scheduling	parametric-list-based=0,1,2, speculative-sdc-scheduling, no-chaining
register-allocation	WEIGHTED_TS, COLORING, WEIGHTED_COLORING, CHORDAL_COLORING, BIPARTITE_MATCHING, TTT_CLIQUE_COVERING, UNIQUE_BINDING
shared-input-registers	
module-binding	WEIGHTED_TS, WEIGHTED_COLORING, COLORING, TTT_FAST, TTT_FAST2, TTT_FULL, TTT_FULL2, TS, BIPARTITE_MATCHING, UNIQUE
memory-control-type	D00, D10, D11, D21
sparse-memory	on, off
do-not-use-asynchronous-memories	
serialize-memory-accesses	
rom-duplication	
bram-high-latency	3, 4
device-name	5SGXEA7N2F45C1, EP2C70F896C6, EP2C70F896C6-R, EP4SGX530KH40C2, LFE335EA8FN484C, LFE5U85F8BG756C, LFE5UM85F8BG756C, asap7-BC , asap7-TC, asap7-WC, nangate45, nx1h140tsp, nx1h35S, nx2h540tsc , xc4vlx100-10ff1513, xc5vlx110t-1ff1136, xc5vlx330t-2ff1738, xc5vlx50-3ff1153, xc6vlx240t-1ff1156, xc7a100t-1csg324-VVD, xc7vx330t-1ffg1157, xc7vx485t-2ffg1761-VVD, xc7vx690t-3ffg1930 -VVD, xc7z020-1clg484, xc7z020-1clg484-VVD, xc7z020-1clg484-YOSYS-VVD, xc7z045-2ffg900-VVD, xcku060-3ffva1156-VVD, xcu280-2 Lfsvh2892-VVD, xcu55c-2Lfsvh2892-VVD
libm-std-rounding	
fp-subnormal	
fp-exception-mode	ieee, saturation, overflow
fp-rounding-mode	nearest_even, truncate
hls-div	none, nr1, nr2, NR, as
fsm-encoding	auto, one-hot, binary
DSP-fracturing	16, 32
timing-model	EC, SIMPLE
aligned-access, unaligned-access	
O	0, 1, 2, 3, s, fast, g
floating point implementation	soft-float, flopoco, soft-fp, fp-subnormal
hls-fpdiv	SRT4, G, SF
compiler	I386_GCC7, I386_GCC8, I386_CLANG6, I386_CLANG7, I386_CLANG8, I386_CLANG9, I386_CLANG10, I386_CLANG11, I386_CLANG12

Table 4: All HLS compiler arguments TurboTurtle supports for Bambu v2024.10.

Vulnerability	C operator/construct	HLS compiler configuration	Mitigation
1	<int> / <int>	--compiler=I386_GCC7,8 and --hls-div=nr1,nr2,NR,as	--pipelining
1	<int> / <int>	--compiler=I386_CLANG6,7,8,9,10,11,12 and --hls-div=NR,as	
1	<long long int> / <long long int>	--parametric-list-based=0,1,2 and --hls-div=nr1,nr2,NR,as	
1	<long long int> / <long long int>	--no-chaining and --hls-div=nr1,nr2,NR,as	
1	<long long int> / <long long int>	--speculative-sdc-scheduling and --hls-div=NR,as,nr1	
1	<unsigned long long int> / <unsigned long long int>	--hls-div=NR,as	
1	<unsigned long int> / <unsigned long int>	--hls-div=as	
2	<int> % <int>	--compiler=I386_GCC7,8 and --hls-div=NR,as	--pipelining
2	<int> % <int>	--compiler=I386_CLANG6,7,8,9,10,11,12 and --hls-div=NR,as	
2	<long long int> % <long long int>	--parametric-list-based=0,1,2 and --hls-div=nr1,nr2,NR,as	
2	<long long int> % <long long int>	--no-chaining and --hls-div=nr1,nr2,NR,as	
2	<long long int> % <long long int>	--speculative-sdc-scheduling and --hls-div=NR,as,nr1	
3	<float> -> <unsigned int>	--parametric-list-based=0,1,2 or --no-chaining	--pipelining
3	<float> -> <unsigned long long int>	--parametric-list-based=0,1,2 or --no-chaining	
3	<unsigned long int> -> <float>	-	
3	<double> -> <short int>	--parametric-list-based=0,1,2 and --soft-float	
3	<double> -> <short int>	--parametric-list-based=0,1,2 and --fp-subnormal	
3	<double> -> <short int>	--no-chaining and --soft-float	
3	<double> -> <short int>	--no-chaining and --fp-subnormal	
3	<double> -> <unsigned long long int>	--parametric-list-based=0,1,2	
3	<double> -> <unsigned long long int>	--no-chaining	
3	<double> -> <float>	-	
3	<unsigned char> -> <double>	--parametric-list-based=0,1,2 and --soft-float	
3	<unsigned char> -> <double>	--parametric-list-based=0,1,2 and --fp-subnormal	
3	<unsigned char> -> <double>	--no-chaining and --soft-float	
3	<unsigned char> -> <double>	--no-chaining and --fp-subnormal	
3	<unsigned short int> -> <double>	--parametric-list-based=0,1,2	
3	<unsigned short int> -> <double>	--no-chaining	
3	<unsigned long long int> -> <float>		
4	<float> / <float>	--parametric-list-based=0,1,2 or --no-chaining	
4	<double> / <double>	--fp-exception=ieee,saturation and -O0,1,2,s,g and --hls-fpdiv=G	
5	<float> - <float>	--soft-float	--pipelining
5	<float> - <float>	--fp-subnormal	
6	<float> + <float>, <float> + <double>	--soft-float	--pipelining
6	<float> + <float>	--fp-subnormal	
6	<double> + <double>	--soft-fp	
7	<float> * <float>	--soft-float, --fp-subnormal	--pipelining
8	double secret_array[]; float f = (float) secret_array[i];	--channels-type=MEM_ACC_11 --distram-threshold=2048 --memory-allocation-policy=ALL_BRAM --compiler=I386_CLANG12 --speculative-sdc-scheduling --register-allocation=WEIGHTED_COLORING --module-binding=COLORING --memory-ctrl-type=D00 --sparse-memory=on --do-not-use-asynchronous-memories --rom-duplication --bram-high-latency=4 --device-name=EP2C70F896C6-R --fp-exception-mode=saturation --fp-rounding-mode=nearest_even --fsm-encoding=auto --DSP-fracturing=32 --timing-model=EC --hls-div=nr2	default arguments
9	<bool> && <bool>, <bool> || <bool>		--pipelining
12	<float> <= <float>, <dobule> <= <dobule>	--compiler=I386_CLANG7	--pipelining or --compiler=I386_GCC8
	<dobule> < <dobule>	--compiler=I386_CLANG10	

Table 5: Vulnerable C code and HLS configuration combinations for Bambu. The operators in the left column, combined with the HLS argument combination in the right column in the same line lead to vulnerable hardware in Bambu. '-' signifies default arguments. Column 'Mitigations' lists Bambu arguments that produce secure RTL.

Vulnerability	C operator	HLS compiler configuration
10	`<bool> && <bool>, <bool> || <bool>`	
11	`<anything> * <boolean value>`	-
11	`(value & masked_secret) | (0 & ~ masked_secret);`	-

Table 6: Vulnerable C statement and HLS configuration combinations for Dynamatic. The operators in the left column, combined with the HLS argument combination in the right column in the same line lead to vulnerable hardware in Dynamatic. '-' signifies default arguments.

B Open Science Appendix

B.1 Enumeration of artifacts

All our artifacts are provided in one repository, which contains the following elements. We describe them in more detail below.

- CTurtle code
- TurboTurtle code
- Vulnerable code and case studies
- Documentation, including Docker setup for installing tools and reproducing everything that can be reproduced with open source tools.

B.1.1 Contribution 1: CTurtle. CTurtle generates the random C programs that follow customizable constant-time rules. The rules can be configured in a the <repositoryname>/config.toml file. We provide all the configuration examples that we used to obtain the results. The Python code is available in <repository-name>/turboturtle/cturtle. It is invoked via start.py.

B.1.2 Contribution 2: TurboTurtle. TurboTurtle generates the necessary scripts to synthesize the generated (or provided custom) programs with the various HLS tools and adds pragmas and directives. It further generates the scripts for simulating the generated RTL design in a miter testbench. TurboTurtle is invoked via start.py.

B.1.3 Contribution 3: HLS tool evaluation. For transparency and reproducibility, we provide the generated programs, scripts and generated hardware with which we observed the vulnerabilities reported in Section 8. We further provide some designs that we found to be secure for reference. New fuzzing runs (including the generation of new programs) may be performed with the tools mentioned earlier.

B.1.4 Documentation and setup. The repository contains a file called README.md that lists the dependencies, describes how to install the tools and how to configure and launch CTurtle and TurboTurtle. There are several config*.toml files for configuring CTurtle and TurboTurtle. The possible configurations are described as comments in the file. As described in the README, we provide Docker files and installation instructions.

B.1.5 Artifacts that cannot be shared due to licensing restrictions. Due to licensing restrictions we cannot reveal the names of Tool A and Tool B. We further cannot provide any scripts that contain proprietary synthesis or simulation commands. However, our core contributions, CTurtle and TurboTurtle, are generic tools where only a small portion is HLS tool or simulator specific. This includes tool-specific HLS arguments and invocation commands, as well as simulation commands. CTurtle is completely independent from any other tools. TurboTurtle is able to work with various open and closed-source tools. Thus, we provide a representative fuzzing flow by evaluating two open-source HLS tools Dynamatic and Bambu, using the open-source tool Verilator as simulator. When fuzzing Tool A and B only their tool specific HLS arguments and invocation commands differ from the open-source tool evaluation. Further, we use QuestaSim as a simulator in some cases. We provide all corresponding testbenches, except the proprietary launch scripts and library names.

B.2 Availability

Artifacts are maintained here: https://github.com/comsec-group/turboturtle. A permanent initial version can be found here: https://zenodo.org/records/20754126.