VeriSilicon: Towards a Comprehensive Framework for Secure FPGA Development

Xiaoyan Liu Baojian Hua* Junmin Wu* Hao Zhu Zhizhong Pan
School of Software Engineering
University of Science and Technology of China
{sa20225316lxy, hhxk, sg513127}@mail.ustc.edu.cn {bjhua, jmwu}@ustc.edu.cn*

***Abstract*—Field Programmable Gate Array (FPGA) is a highly flexible and performant general-purpose programmable logic device, and has been successfully used in diverse scenarios such as digital signal or image processing, parallel computing, and low power systems. Recently, High Level Synthesis (HLS) has been proposed to develop FPGA applications at high level, thereby increasing productivity with less efforts. Unfortunately, HLS lacks security mechanisms, which may lead to vulnerabilities in the hardware logic, defeating the security guarantees of FPGAs.**

In this paper, we present VeriSilicon, a comprehensive framework and novel architecture for secure HLS development on FPGA. VeriSilicon consists of three novel techniques: 1) it advocates the use of security language as the developing language by leveraging Low*, a refinement type-based verification-oriented language; 2) it incorporates automated verification tools to protect the functional correctness of the program, by using Bedrock's automatic proof feature; and 3) It introduces a verified third-party library, `krmllib`, containing secure C library functions compatible with HLS. We have implemented a prototype for VeriSilicon and have conducted systematic experiments to evaluate the effectiveness of VeriSilicon in secure FPGA designs. Experimental results show that VeriSilicon is effective in detecting and reporting security vulnerabilities in source programs.

Index Terms*—Field Programmable Gate Array, High Level Synthesis, software security, Low

I. Introduction

Field Programmable Gate Array (FPGA) [1] [2] is a general-purpose, high performance, and programmable silicon device providing capabilities to program functionalities directly into a chip, thereby offering several technical advantages over general-purpose microprocessors. First, the reconfigurable nature of FPGA makes it highly flexible and scalable to deploy diverse functionalities on a single chip. Second, FPGA eliminates the lengthy IC manufacturing cycles [17] and thus significantly reduced Non Recurring Engineering (NRE) costs. Third, The fact that FPGAs do not need to be instruction driven makes them more efficient in execution and thus more suitable to perform computationally intensive tasks. Due to these technical advantages, FPGA chips are becoming more popular and are being deployed in productions such as 5G communications [3] [4], AI [5] [6], and autonomous driving [7] [8]. In the coming decade, a desire to pursue higher computing power with lower energy consumptions will offer more opportunities for FPGA.

While FPGA makes programming general-purpose chips *possible*, it does not make it *easy*. Traditionally, Register Transfer Level (RTL) [9] has been the dominant low-level programming language for FPGA development, proving capabilities to program digital circuits in a direct manner. While RTL is flexible to express lower-level design decisions and optimizations, programming in it is rather complex, laborious, and error-prone, due to its low-level nature. Furthermore, the rapid increase of System-on-Chip (SoC) design complexity has led even higher development costs and a lengthy process of using RTL [10]. To alleviate these problems, High-Level Synthesis [47] has been proposed for the FPGA domain. By developing FPGAs based on the HLS approach, designers are able to easily design functionally in a high-level language (e.g. C/C++, OpenCL) and automatically convert it into an efficient and accurate RTL circuit model using an HLS compiler, thus reducing the design and verification effort of RTL code and focusing more on system design. Due to this development cost and efficiency advantage, a variety of application areas including bioinformatics [19], image processing [21], scientific computing [22] and more are using HLS for FPGA synthesis design.

Unfortunately, although HLS offers advantages of flexibility and abstraction [18], it lacks security capabilities and thus may lead to exploitable vulnerabilities in the deployed FPGA binaries [71]. In particular, state-of-the-art HLS infrastructures still suffer from three problems: 1) unsafe languages, 2) no correctness guarantees, and 3) unreliable libraries. First, the source languages received by the HLS process (e.g. C/C++) are extremely risky in terms of security, as they not only do not provide security for manual memory management but also do not have automatic array bounds checking and rubbish collection, making such programs vulnerable to memory security vulnerabilities such as hanging pointer problems and buffer overflows. Second, the HLS process does not provide sufficient assurance of correctness, as existing HLS methods prove the correctness of a program solely on the basis of empirical tests, but no matter how sophisticated such tests are, if no incorrect behavior is triggered, the program cannot usually be judged to be correct. Third, reusable third-party libraries are also one of the most overlooked threats in an application, as there is

* Corresponding authors.

no guarantee that adequate security checks have been done during the building of the library and it is difficult to enforce security controls on third-party code. As a result, addressing this issue is challenging. Unfortunately, existing research has not been able to fully address all three of these issues without loss of performance.

Our work. In this paper, we propose VERISILICON, a novel framework for secure HLS development on FPGA. Specifically, VERISILICON consists of three key components. First, we introduce a modern security language oriented toward program verification as the source language for HLS processes. Second, we propose to utilize verified library functions for program development. Third, we leverage an automated validator to guarantee the correctness of programs accepted by HLS. We argue that this work stands for a novel contribution to the research field of how to build safe and correct FPGA designs.

We have implemented a prototype for VERISILICON and have conducted extensive experiments to evaluate it in terms of effectiveness. The experimental results show that VERISILICON can effectively avoid the memory errors that occur when developing directly in C and complete a safe and reliable FPGA IP design.

Contributions. This work represents the first *step* towards defining a comprehensive framework for *secure* HLS development on FPGA. To summarize, our work makes the following contributions:

- **A novel and comprehensive framework for secure HLS development on FPGA.** We studied the security of the input programs accepted by the HLS tool systematically, and then presented VERISILICON, a novel framework for secure HLS development on FPGA.
- **Prototype implementations.** We demonstrate a prototype implementation of VERISILICON to automate the protection of FPGA designs.
- **Extensive evaluations.** We have conducted extensive experiments to evaluate VERISILICON in terms of effectiveness. Experimental results demonstrated that VERISILICON can effectively avoid memory errors.

Outline. The rest of this paper is organized as follows. Section II presents the background for this work. Section III presents the security challenges of modern HLS processes and the threat model assumed in the work of this paper. Section IV and V present the design and implementation of VERISILICON, respectively. Section VI presents the experiments to evaluate VERISILICON. Section VII discusses directions for future work. Section VIII presents the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work: Field Programmable Gate Arrays (Section II-A), and High-Level Synthesis (Section II-B).

A. Field Programmalbe Gate Arrays

Brief history. Field Programmable Gate Array (FPGA) has a very long history in the field of integrated circuits. In 1986, William et al. [11] published the first paper on FPGAs, describing the first product in the field, the XC2064. The early FPGAs were based on a four-input lookup table (LUT) architecture, which had limited capacity and many of the LUTs in the architecture were not fully utilised. But by the 1990s, the FPGA field gained momentum with the invention and development of techniques such as LUT mapping algorithms [12], runtime reconfiguration [13], routing algorithms [14] and advances in integrated circuit processes. In 2000, the modern FPGA field has started to discuss cluster-based architectures [15], based on a solid theoretical foundation [16], FPGA architectures have gained richer diversity and complexity. In 2011, FPGA high-level synthesis technology successfully transitioned from prototyping to deployment [47] and FPGA programmability thus gained a breakthrough.

Development workflow. Due to its reconfigurable nature, the complete FPGA development consists of five typical phases: 1) design, 2) simulation, 3) synthesis, 4) place and route, and 5) chip debugging. First, in the design phase, circuit logics are developed using Hardware Description Language (HDL) or High Level Language (HLL). Second, the simulation phase verifies the logical functionality of the circuit logics before compilation, to guarantee that the logics are of the desired functionalities. Third, the synthesis phase converts the circuit into a netlist consisting of a series of basic logic cells. Fourth, the place and route process maps the synthesis-generated logic netlist onto specific hardware resources. This mapping process is often repeated several times to obtain the best possible logic structure. Finally, the chip debug phase downloads the generated FPGA configuration files (i.e., the bitstream files), into the FPGA chips to run it.

Advantages. FPGAs have significant advantages over non-reconfigurable traditional architectures in three aspects: performance, cost, and flexibility. First, FPGA is of high performance, due to their advanced nanoscale manufacturing processes, as well as the growth of logic cells in an FPGA chip. Second, FPGA's design cost was reduced significantly, due to the reusability of Intellectual Property (IP) cores [25]. Third, FPGA is highly flexible, because the logic functionality can be changed easily, even after deployment [26].

Wide applications. Due to its technical advantages, FPGA has wide applications in diverse scenarios, such as digital signal processing [27], image processing [28], cryptography [29], parallel processing [30], artificial intelligence [31], big data [33]. Furthermore, FPGA already accounts for more than 80% of the total demand in the automotive electronics area. By 2021, the global market for FPGA chips had reached US$6.86 billion, driven by 5G communications and AI applications.

B. High-Level Synthesis

HLS development. The High-level Synthesis (HLS) is essentially a design automation approach converting logic structures described by high-level languages (e.g., C/C++, OpenCL) into circuit models described by low-level Hardware Description Languages (HDL) (e.g., Verilog, VHDL), in an automated manner. A typical HLS-based FPGA development flow consists of three main phases: design input, simulation, and synthesis. First, in the design input phase, functional designs using HLL are delivered to the HLS tool. Second, the simulation phase verifies HLL programs to guarantee their functional correctness. Third, the synthesis phase converts the HLL programs into register transfer language (RTL) designs (i.e., HDL code), which will be further verified for functional correctness using co-simulation.

HLS advantages. HLS-based FPGA development approach has three technical advantages over the RTL-based approach: 1) efficiency; 2) performance; and 3) the agility. First, HLS-based approach is of high development efficiency, as little manual coding or tuning is required, saving development and debugging time. Second, FPGA programs developed with the HLS-based approach are normally of high performance, as HLS incorporates powerful optimizing algorithms enabling more optimization opportunities such as better parallelism, and higher acceleration. Third, the HLS-based approach is more agile to adapt to hardware changes, due to its key property of reconfigurability.

HLS tools. Due to its technical advantages, in the last decade, many HLS-based FPGA development tools have been proposed. First, from the input language point of view, many HLS language (e.g., C/C++ and OpenCL) have been proposed. Second, from the programming models point of view, HLS models have been developed including control flow-based models (e.g. Legup [48] and Bambu [49]) and data flow-based models (e.g. FAST-LARA [50] and OXiGen [51]). Finally, from a compiler transformation perspective, many HLS compilation strategies have been proposed, including pragma-driven [46] and automatic polyhedral compilation [52] [53].

III. Security Challenges and Threat Model

In this section, we present security challenges HLS faces (Section III-A), and the threat model for this work (Section III-B).

A. Security Challenges of HLS

Although HLS is a promising in programming FPGAs and has made significant successes, it, unfortunately, still faces three security challenges: 1) the usage of an insecure source language is prone to poor coding with security vulnerabilities; 2) the lack of correctness verification of the HLS development process does not adequately guarantee the correctness of the program; and 3) the calling of untrustworthy third-party libraries results in a very vulnerable design on the FPGA.

First, the source languages for HLS are usually low-level languages that lack memory security, such as C and C++. Although these languages can offer high performance, they forgo some basic security measures such as bounds checking and rubbish collection. The most common buffer overflow problem is caused by the lack of automatic bound checking, and this security vulnerability is ranked as the most dangerous software weakness in 2021. Moreover, to achieve low-level control over memory, developers need to perform memory management manually, which places a huge security burden on the development process, as the smallest memory operations in code can lead to security vulnerabilities. For example, `free()` in C does not guarantee that memory will be safely released, as the programmer can manually reclaim arbitrary heap data.

Second, the modern HLS development process is simulation-based and lacks rigorous proof of program correctness. On the one hand, to cover all functional behavior, developers need to write a large number of directed test cases, yet with the increasing complexity of the design, simulation-based techniques cannot prove that the design is correct on all inputs. On the other hand, the long simulation times also make it almost infeasible to use simulation alone for complete testing. Due to the vast number of chip-level scenarios to be covered, it is also impractical to perform chip-level formal verification after the HLS flow has been completed. Although studies have been proposed to perform formal verification at the beginning of the design, the non-automated formal verification they present is time-consuming and difficult to master for most engineers who are not trained in mathematics, making it difficult to utilize in the real world.

Third, third-party libraries are the easiest way for an attacker to gain access to sensitive data on a system and undermine the security of a program, and are potentially a significant factor in compromising code quality. A prime example of this is the recent discovery of a major security vulnerability in the GNU C library, that is a `getcwd()` function used to get the pathname of a working directory has incorrect boundary checks, leaving the program at risk of buffer overflows and underflows. If an attacker controls the input buffer and size parameters of `getcwd()` passed in the setuid program, then it is possible to execute arbitrary code on the system and elevate user privileges. Developers are unknowingly exposed to this vulnerability because vulnerabilities in third-party libraries are often not found and fixed in time when no serious security incidents occur. This also implies that developers must not only check their programs, but also ensure that there are no security threats to third-party libraries, otherwise the system on the FPGA remains very vulnerable.

B. Threat Model

The focus of this work is to present a comprehensive framework for secure HLS-based FPGA development. Therefore, we make the following assumptions in the threat model for this work.

We assume that the FPGA infrastructure hardware, including but not limited to memory, registers, and data buses, is secure and unexploitable. To secure these hardware, recent trusted execution environment technologies (e.g., Intel SGX [34] or ARM TrustZone [35]) can be leveraged, which are orthogonal to the work in this paper. Meanwhile, we assume that the FPGA binary bitstream cannot be maliciously forged by an adversary, as FPGA vendors already provide bitstream encryption and authentication features for confidentiality and integrity protection (e.g., the HMAC-SHA-256 on Xilinx 7-series, or SHA-256 on Altera's Arria 10).

We assume that the third-party IPs are secure and may not compromise the systems, because a myriad of techniques have been proposed (e.g., split manufacturing [36], functional locking [37], and hardware Trojan detection [38]). In addition, we assume that the advanced synthesis tools are secure and will not introduce malicious functionalities into the FPGA design tools, as many studies [39] have been proposed to check CAD tools guaranteeing their integrity.

We assume that programs written in the existing HLS source language are insecure. For example, if the developer forgets to manually check certain boundary conditions of a buffer, since the current HLS source language does not enforce boundary checking, even though it does not trigger an obvious error, in fact, an attacker is able to exploit this flaw to read or write to arbitrary memory locations in the process, causing security vulnerabilities such as buffer overflows.

We assume that the design mapped to the FPGA is unreliable. For example, for a simple function that calculates the sum of two 32-bit integers and outputs a 32-bit integer, if the program does not have constraints on the bounds of the input data and does not perform strict formal verification, there is an integer overflow risk in the function that an attacker can exploit to execute malicious code to infect the device, causing risks such as sensitive data leakage.

We assume that the third-party library functions called in the program are not trusted. For example, if the designer calls a third-party library function with a security vulnerability, such as `getcwd()`, the attacker is able to control the native code to perform arbitrary illegal operations such as privilege elevation. Therefore, we need to achieve safe and secure FPGA development by enhancing the security and correctness of the native code.

IV. VERISILICON DESIGN

In this section, we present the framework of VERISILICON by describing its design in detail. We first discuss the design goals (Section IV-A) and general architecture of VERISILICON (Section IV-B), followed by the secure input module (Section IV-C), the reliable translation module (Section IV-D), the trusted third party libraries model (Section IV-E), and the automated verification module (Section IV-F).

A. Design Goals

The design of VERISILICON should follow three important principles: 1) hierarchical security, 2) end-to-end, and 3) fully automation. First, VERISILICON applies a hierarchical view to security, guarantees security at each layer in a hierarchical perspective, allows modules in the framework to be easily extended, and emphasizes the holistic cross-layer philosophy of secure FPGA design. Second, VERISILICON is an end-to-end framework that provides assurance that flow complexity will not impact on layer interconnections and the viability of

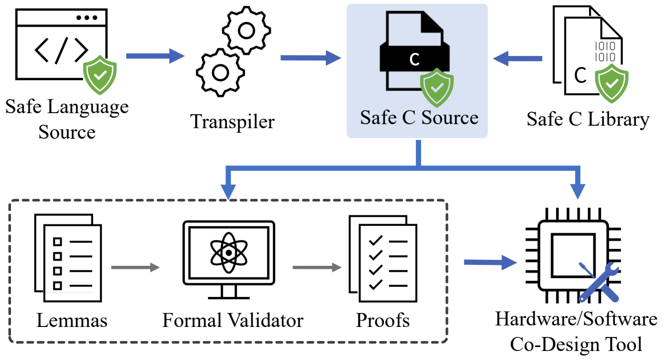

Fig. 1: The overall design of VERISILICON.

the flow from the coding design phase to the final deployment on FPGAs. Third, VERISILICON should be fully automated to minimize manual intervention and avoid security risks posed by human intervention, otherwise developers are prone to abandon inefficient secure development after weighing efficiency and security.

B. Overall Architecture

Fig. 1 illustrates the general architecture of VERISILICON. Specifically, VERISILICON consists of four modules: a secure input module, a trusted translation module, a validated third-party library, and an automated verification module. 1) The input module refers to a specification-compliant, security-vulnerable application written in a modern security language. 2) The translation module receives the source program of the input module, translates it source-to-source and converts it to the type of program that the HLS software tool can receive. 3) We provide a number of validated third-party libraries, and since third-party libraries are noted as untrustworthy in the threat model (Section III-B), developers can use verified third-party libraries to prevent threats from unreliable libraries. 4) The verification module will verify the programs received by the HLS tool to ensure functional correctness. This character of VERISILICON's abstract modularity allows it to be easily integrated into existing hardware/software co-design tools.

C. Secure Input

In the input module, developers use a secure source language to build secure applications. While these applications may be of various types, for example, from underlying algorithms for deep neural networks to high video quality encoders to dedicated accelerators, the overall development specification for the input module follows two overarching design goals: memory safety, and type safety.

First, we expect developers to use memory safe programming languages for program development, so as to prevent the introduction of errors related to memory usage, such as out-of-bounds reads and writes. Memory security vulnerabilities are one of the most common vulnerabilities; for example, Microsoft [40] estimates that 70% of the vulnerabilities in its products are memory security related, and this is the main type of vulnerability that VERISILICON protects against. Therefore, VeriSilicon uses memory-safe language to prevent memory leaks and maintain system stability.

Second, we expect developers to use programming languages with dependent types for program development. Unlike statically typed languages such as C/C++, languages with dependent types designed for verification allow the code to be more expressive, enabling the compiler to catch more errors and thus avoid a large number of errors such as null pointer exceptions, off-by-one errors, etc. By using a dependent type language VeriSilicon transfers the runtime checks in a statically typed language to the type system itself, thus avoiding program execution failures.

The development of the input modules is based on these two design principles, thus achieving the goal of ensuring highly reliable and secure FPGA development at the source code level.

D. Reliable Translation

The translation module takes the secure application generated by the input module and converts it into a program described in a source language that can be accepted by hardware/software co-design tools, such as a typical C/C++ language. While completing the translation, the module also needs to have three main characteristics: safe and secure, readable output, and fast translation speed.

First, the primary requirement of the translation module is to ensure that there are no changes in the functional and security characteristics of the program before or after translation. Therefore, it needs to complete the equivalent conversion of the source and target programs in such a way that the safety properties guaranteed in the input module are robustly retained intact in the output program, and this process requires the translation module to have already verified safety protection capabilities.

Second, the target program generated by the translation module needs to be highly readable. As the developer needs to check the translated program to ensure that the translation is correct and then debug the code as well as complete better code optimization, the readability of the output program is also a point that VeriSilicon must consider.

Third, the addition of translation modules should not impose a headache time burden on the FPGA development process. As FPGA developers need to expend much effort on the design of the hardware logic, we do not want the development process to be extremely time costly due to the introduction of security mechanisms.

E. Trusted Third Party Libraries

The trusted third-party libraries in the framework refer to a set of services that bring efficiency and reliability to application development. Broadly speaking, the "trusted" feature requires third-party libraries to guarantee three things: functional correctness, security and compatibility.

First, the most important thing for a third-party library to do is to be able to fulfil the functionality it promises, and the functions in the library need to be consistent and complete with the functionality it describes. Otherwise, it loses its basic usability as a service component.

Second, in order to guarantee the security properties of the VeriSilicon framework, the third-party library also needs to be secure. Therefore, the runtime libraries introduced during coding must be validated and free of security vulnerabilities to ensure that there are no exploitable security risks on the FPGA due to the introduction of insecure runtime libraries during development.

Third, the third-party libraries used by developers need to be compatible with the development environment. Compatibility here is not just the ability to work with HLS software development tools, but also with other libraries and system resources that are linked to the development.

F. Automated Validation

The verification module takes the target program generated by the translation module and the linked trusted third-party libraries, verifies them formally and ensures that the design is functionally correct. Essentially a rigorous proof of the program using mathematical methods, verifying that the program performs exactly according to the specifications in the design, i.e. that for all possible outputs, there is never an output that exceeds expectations. While satisfying the basic verification functions, we also expect the verification module to satisfy two design goals: utility and automation.

First, the verification tool should be as simple to use as possible and be supported by comprehensive documentation and a good visual interface. In general, it is not easy to master formal verifiers and overly complex tools only increase the cost of FPGA development, so verification tools need to have accompanying user guides to help with their use. Additionally, the tools should have detailed error prompts that clearly explain the root cause of the problem for troubleshooting purposes.

Second, verification tools need to verify the correctness of the program logic in an automated manner, reducing manual intervention by the developer. If the verification method is semi-automatic, we will get nothing if the proof of the theorem for a given problem fails. Therefore, a fully automated approach is needed to complete the verification process, with the necessary explanations to the user at the right time, to reduce the developer's task of understanding the verification process, and thus to complete an efficient FPGA development process.

V. VeriSilicon Implementation

In this section, we present a prototype implementation for VeriSilicon. We present the implementation of each module in Fig. 1 from top to bottom, including the implementation of the input module (Section V-A), the implementation of the translation module (Section V-B), the implementation of the verification module (Section V-C), the implementation of third party library (Section V-D).

A. Implementation of Input Module

VeriSilicon uses the modern security language Low* to finish the source programs in the input modules. Low* is a language for efficient low-level programming and verification, the core of which lies in its unique region-based memory model, Hyper-stacks. Hyper-stacks cover both stack and heap and are able to capture the allocation behaviour of functions, this memory model not only provides lightweight stateful verification but also indicates that the lifecycle of a particular set of regions follows specific rules. The stack regions are not only used as inference devices, but also provide efficient memory management mechanisms [41]. Low* is effective in ensuring memory security and type security, on one hand because the stack computations are ST computations, ensuring that all heap-allocated references are explicitly released before the program returns, and because the release and dereference of memory in the program requires that their parameters exist in current memory, and on the other hand because Low* uses type abstraction to ensure that memory access patterns are secretly independent and uses SMT solvers for type checking. Its utility and effectiveness have been demonstrated by implementing and proving ChaCha20 stream ciphers [42], Poly1305 MAC [43], and many other cryptographic algorithms. Therefore, VeriSilicon uses Low* to ensure that the input source program is secure.

B. Implementation of Translation Module

VeriSilicon uses the KaRaMeL compiler to automatically convert Low* programs to C programs. The C code generated by KaRaMeL is well-formed and highly readable. At the same time, the translation verification function of KaRaMel enables the compiled code to be statically re-checked to ensure the safety and reliability of the program, and the proof saving technique eliminates the risk of compilation failure. KaRaMeL first completes the conversion from Low* to C* and then from C* to C, where the syntax is close to that of C, and the whole process takes less than one second [41]. Furthermore, the correctness of the conversion from Low* to C is proven, i.e., if the source Low* program is safe, then the target program is also safe. Thus, VeriSilicon uses the KaRaMeL compiler to ensure that the converted C program has the same safety properties as the source program.

C. Implementation of Verification Module

VeriSilicon automates correctness verification of C programs using Bedrock, a tool for verifying low-level programming in Coq that combines generative metaprogramming and Hall logic. Bedrock enables highly automated correctness verification by proposing a low-level language with a macro concept that completely separates the interface from the implementation. This macro system is part of the Bedrock library, which provides a common integrated environment for program implementation, specification, and verification that supports automated proofs of programs built using macros extracted from different sources. Each macro contains a predicate converter and verification condition generator, which starts from assembly and builds structured code generators that are associated verification condition generators [44] [45]. In particular, one of the advantages that VeriSilicon gains from using the verified macro system provided by Bedrock is the ability to obtain a certain level of abstraction without sacrificing performance.

D. Implementation of Third Party Library

The verified third-party library called in VeriSilicon is the runtime support library provided in KaRaMel, `krmllib`, which includes the machine integer library, buffer library, data structures and other common C libraries. Among them, the integer library provides common arithmetic operations, and for unsigned integers, the `add_mod` function is able to detect overflow vulnerabilities. The buffer library is able to provide static boundary checking operations to ensure that the index position is within the legal range. The code in these core libraries has been verified to be secure and functionally correct. Therefore, VeriSilicon leverages the library functions in krmllib to accomplish efficient and secure source code development.

VI. Evaluation

In this section, we present experiments to evaluate VeriSilicon. We first present the research questions that guided the experiments, and then we experimentally verify the effectiveness of VeriSilicon on microbenchmark tests.

Research Question. By presenting the experimental results, our main research question is whether VeriSilicon can effectively guarantee that no memory vulnerabilities exist in the source programs accepted by HLS, thereby avoiding security threats in FPGA designs?

Experimental Setup. All the experiments are performed on the Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation kit shown in Figure 2, a development board equipped with an Arm® CortexTM-A53 processing system (PS), a dual-core Arm Cortex-R5 real-time processor, and multiple FPGA programmable logic (PL).

The design work of all the FPGA IP cores is completed on a server with Intel i7 core CPU and 32GB of RAM running Ubuntu 20.04.The integrated development environments used for IP design and synthesis are Xilinx Vitis HLS, Vivado (version 2022.2).

Fig. 2: Xilinx Zynq UltraScale+ MPSoC ZCU104 board

Experimental Design. To validate that VERISILICON can effectively guarantee that FPGA designs are secure, we build a set of microbenchmark tests written in C and Low* languages, respectively, and hand-inject common vulnerabilities into the test code, such as divided by zero errors and addition overflow errors. We use these typical vulnerability tests to verify that VERISILICON can detect and give feedback to developers on hidden security vulnerabilities in source code.

```
// code with division by zero error in C
int div_IP(int a, int b){
   return a / b;
}
```

(a) Function with implied division by zero error in C.

```
// compilation will terminate
let div_IP (x: Int32.t)(y: Int32.t): Int32
    .t =
  let open FStar.Int32 in
  x /^ y
```

(b) Function with implied division by zero error in Low*.

Fig. 3: Sample code illustrating C and Low* code with implied division by zero error.

```
// code with addition overflow error in C
int add_IP(int a, int b){
  return a + b;
}
```

(a) Function with implied addition overflow error in C.

```
// compilation will terminate
let add_IP (x y: FStar.UInt32.t): FStar.
    UInt32.t =
  let open FStar.UInt32 in
  x +^ y
```

(b) Function with implied addition overflow error in Low*.

Fig. 4: Sample code illustrating C and Low* code with implied addition overflow error.

Division by Zero. First, we write the source program containing the division by zero error in C, as shown in Fig. 3(a).The fact that there is no non-zero restriction on the operands in the program results in a possible zero input for the second parameter (i.e., the divisor). Handing this code to Vitis HLS and Vivado tools, the synthesis process did not prompt any errors and the simulation passed. The generated bitstream file was run on the FPGA and found that the program not only did not crash without any error hints, but also returned "-1" as the result of the calculation.

As a comparison, we complete the same source program containing the divide by zero error using the safety language in VERISILICON (i.e., Low*), as shown in Fig. 3(b). Since Low* is a verification-oriented safety language, it performs safety checks on the program before compilation, so it provides the developer with the error message `"Subtyping check failed"` without compiling, i.e., the program does not have a non-zero constraint on the divisor, thus efficiently avoiding the division by zero error.

Addition Overflow. First, we write the source program containing the integer addition overflow in C, as shown in Figure. 4(b). The length of the return value is not limited in the program, which leads to the possibility of the return value being longer than 32 bits when the input parameters of the program are too large, and thus producing an incorrect result. Similar to the divide by zero error, we handed this program to the Vitis HLS and Vivado tools, and they successfully completed the IP synthesis without giving any error hints to the developer. When this program was executed on the FPGA, the system did not crash and did not indicate any memory-related error messages.

As a comparison, we use the safe language in VERISILICON (i.e., Low*) to complete the same source program containing the additive overflow, as shown in Figure. 4(b). The code is compiled by the reliable translator in VERISILICON (i.e. KaRaMel), and the compiler gives a warning message to the developer to limit the `"size"` of the variables in the source code.

These results show that VERISILICON can effectively accomplish vulnerability detection, avoid implicit security vulnerabilities on source code level, and complete the highly secure and reliable FPGA development.

VII. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work. It should be noted that this work aims to complete the design of a safe and correct FPGA by circumventing user negligence during the coding phase.

Secure HLS Guidebook. Similar to the CERT-C [54], MISRA C [55] secure coding standard, we can design a set of guidebooks for secure coding based on modern languages (C/C++/Low*) for users who use HLS tools for FPGA development. Unlike CERT-C and MISRA C, this set of guidelines should favor sequential coding using high-level languages to complete hardware designs. It also needs to take into account the differences in user constraints (pragma, instructions, libraries) and coding styles due to the differences in HLS compilers.

Security verification of RTL code. The work in this paper focuses on vulnerabilities at the high-level language level, however, some vulnerabilities may have been introduced during the HLS translation process. Because HLS tools are not designed with security in mind at the beginning, it makes the underlying HLS algorithms such as scheduling and loop optimization not aware of the security assets in the design, which results in the security of the translation not being guaranteed. At the same time, we consider that the repair of such vulnerabilities may require a complete overhaul of the HLS compiler. Therefore, we can add automatic security verification of the converted RTL code to the current framework. For example, Eric et al. [56] [57] use Hall logic-based reasoning to verify the correctness of the RTL code and use Coq for implementation.

VIII. Related Work

In recent years, there has been a great deal of research into code security and the security of HLS compilers. However, the work in this paper represents a novel contribution to these fields.

A. Code Security

There is a lot of research related to code security in hardware products. The mainstream research includes two types of research, one is vulnerability checking of code in high-level languages such as C/C++, and the other is vulnerability checking of code at the RTL level.

Checks on high-level languages can help developers to identify hidden security vulnerabilities in their coding early. Arnaud et al. [58] have designed a tool called C Global Surveyor (CGS), a distributed implementation of static analysis algorithms on multiple processors, to perform boundary checks on arrays of large embedded C programs.Jacques-Henri et al. [59] proposed the static analyzer Verasco to determine the absence of runtime errors in the C programs being analyzed. John [60] and others presented the ITS4 tool for static vulnerability scanning of C and C++ code and providing real-time feedback to developers.

Security vulnerabilities can exist throughout the lifecycle of a silicon product, so there is also a lot of research focused on securing code at the RTL level. David et al. [61] used an information flow path approach to identify security-related logic in RTL designs to automate the identification of security vulnerabilities in RTL. Yu et al. [62] introduced RTL-ConTest, which identifies security vulnerabilities in RTL by extracting key processes from the RTL design and performing hybrid tests. Orlando et al. [63] presented the RTSEC framework to enhance the security of RTL code by automatically performing security analysis on HDL code and adding security features to it. Yang et al. [64] proposed an assertion-based automation mechanism that uses symbolic execution in the RTL model to activate assertions to generate directed tests to detect Trojan horses. Rui et al. [65] designed the Coppelia framework to perform vulnerability checking based on symbolic testing, after converting the hardware RTL design to C++ code using Verilator [66] and then using the KLEE [67] symbolic execution engine to check it for vulnerabilities.

While these methods can be combined with the FPGA development flow to complete vulnerability detection, however, we focus on completing secure coding at the design stage and reducing the burden of manual detection. Furthermore, our goal is not only system security but also functional correctness.

B. HLS compiler security

There are many studies planned to extend the HLS process to include security. Muttaki et al. [68] propose that HLS can be protected against hardware vulnerabilities by modifying the HLS configuration or by modifying the basic algorithms in the HLS tool, for example by modifying the scheduling optimization algorithms ASAP [69] and ALAP [70] to use minimum latency resources as a security constraint to accomplish security protection. Christian et al. [71] propose the idea of a security-aware HLS tool that, after analyzing the input descriptions, automatically identifies the sensitive data to be protected, and inserts a protection mechanism into the synthesis process to monitor the exchange of data between trusted and untrusted areas. Zheng et al. [72] developed the ASSURE framework, which enhances the security of HLS tools using an information flow control mechanism that not only enables the designer's security policy to be used as an HLS constraint, but also detects information flow violations in the design, performing highly secure operations with a low performance overhead.

However, a key limitation of these studies is that they do not consider whether the source programs received by the HLS compiler are safe and correct; on the contrary, we focus on the fact that the HLS needs to receive safe and correct FPGA designs.

David [24] proposed using Rust to accomplish secure FPGA development, supporting formal verification of the program. The advantage is that security and correctness are obtained at a certain level, but the disadvantage is that verification requires a lot of manual intervention, which affects FPGA design time and development costs.

IX. Conclusion

In this paper, we present VeriSilicon, a comprehensive framework for safe and reliable FPGA development. VeriSilicon uses the advanced type-safe language Low* as the source development language, combined with a verifier and accompanying compiler to generate programs that can be accepted by HLS tools. We have implemented a prototype for VeriSilicon and used it to experiment with the system. The experimental results demonstrate the effectiveness of VeriSilicon. The framework is also suitable for integration into HLS tools with only minor modifications. Overall, the work in this paper enhances the security of the HLS design process without sacrificing efficiency.

References

[1] Kuon I, Tessier R, Rose J. FPGA architecture: Survey and challenges[J]. Foundations and Trends® in Electronic Design Automation, 2008, 2(2): 135-253.

[2] Wollinger T, Paar C. How secure are FPGAs in cryptographic applications? [C]//International Conference on Field Programmable Logic and Applications. Springer, Berlin, Heidelberg, 2003: 91-100.

[3] Chamola V, Patra S, Kumar N, et al. Fpga for 5g: Re-configurable hardware for next generation communication[J]. IEEE Wireless Communications, 2020, 27(3): 140-147.

[4] Visconti P, Velazquez R, Soto C D V, et al. FPGA based technical solutions for high throughput data processing and encryption for 5G communication: A review[J]. TELKOMNIKA (Telecommunication Computing Electronics and Control), 2021, 19(4): 1291-1306.

[5] Qiu J, Wang J, Yao S, et al. Going deeper with embedded fpga platform for convolutional neural network[C]//Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays. 2016: 26-35.

[6] Yin S, Ouyang P, Tang S, et al. A high energy efficient reconfigurable hybrid neural network processor for deep learning applications[J]. IEEE Journal of Solid-State Circuits, 2017, 53(4): 968-982.

[7] Wu T, Liu W, Jin Y. An end-to-end solution to autonomous driving based on xilinx fpga[C]//2019 International Conference on Field-Programmable Technology (ICFPT). IEEE, 2019: 427-430.

[8] Uetsuki T, Okuyama Y, Shin J. CNN-based End-to-end Autonomous Driving on FPGA Using TVM and VTA[C]//2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). IEEE, 2021: 140-144.

[9] Habibi A, Tahar S. A survey on system-on-a-chip design languages[C]//The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings. IEEE, 2003: 212-215.

[10] Gurel M. A comparative study between rtl and hls for image processing applications with fpgas[M]. University of California, San Diego, 2016.

[11] Hsieh J Y J, Mahoney J E, Ngo L T, et al. A User Programmable Reconfigurable Logic Array[J]. 1986.

[12] Cong J, Hwang Y Y. Simultaneous depth and area minimization in LUT-based FPGA mapping[C]//Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays. 1995: 68-74.

[13] DeHon A. DPGA utilization and application[C]//Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays. 1996: 115-121.

[14] DeHon A. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization) [C]//Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays. 1999: 69-78.

[15] Ahmed E, Rose J. The effect of LUT and cluster size on deep-submicron FPGA performance and density[C]//Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays. 2000: 3-12.

[16] Lemieux G, Lewis D. Using sparse crossbars within LUT[C]//Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays. 2001: 59-68.

[17] Trimberger S. Trusted design in FPGAs[C]//2007 44th ACM/IEEE Design Automation Conference. IEEE, 2007: 5-8.

[18] Nane R, Sima V M, Pilato C, et al. A survey and evaluation of FPGA high-level synthesis tools[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2015, 35(10): 1591-1604.

[19] Chen Y T, Cong J, Lei J, et al. A novel high-throughput acceleration engine for read alignment[C]//2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2015: 199-202.

[20] Guo L, Lau J, Ruan Z, et al. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu[C]//2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2019: 127-135.

[21] Conficconi D, D'Arnese E, Del Sozzo E, et al. A framework for customizable fpga-based image registration accelerators[C]//The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2021: 251-261.

[22] Zohouri H R, Podobas A, Matsuoka S. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL[C]//Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2018: 153-162.

[23] Wang J, Guo L, Cong J. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga[C]//The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2021: 93-104.

[24] Hardin D. Hardware/Software Co-Assurance using the Rust Programming Language and ACL2[J]. arXiv preprint arXiv:2205.11709, 2022.

[25] Rodriguez-Andina J J, Moure M J, Valdes M D. Features, design tools, and application domains of FPGAs[J]. IEEE Transactions on Industrial Electronics, 2007, 54(4): 1810-1823.

[26] Serrano J. Introduction to FPGA design[J]. 2008.

[27] Hamouda M, Blanchette H F, Al-Haddad K, et al. An efficient DSP-FPGA-based real-time implementation method of SVM algorithms for an indirect matrix converter[J]. IEEE transactions on industrial electronics, 2011, 58(11): 5024-5031.

[28] Díaz J, Ros E, Pelayo F, et al. FPGA-based real-time optical-flow system[J]. IEEE transactions on circuits and systems for video technology, 2006, 16(2): 274-279.

[29] Rajagopalan S, Amirtharajan R, Upadhyay H N, et al. Survey and analysis of hardware cryptographic and steganographic systems on FPGA[J]. Journal of Applied Sciences(Faisalabad), 2012, 12(3): 201-210.

[30] Tsai C C, Huang H C, Chan C K. Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation[J]. IEEE Transactions on Industrial Electronics, 2011, 58(10): 4813-4821.

[31] Punitha K, Devaraj D, Sakthivel S. Artificial neural network based modified incremental conductance algorithm for maximum power point tracking in photovoltaic system under partial shading conditions[J]. Energy, 2013, 62: 330-340.

[32] Juang C F, Chen J S. Water bath temperature control by a recurrent fuzzy controller and its FPGA implementation[J]. IEEE Transactions on Industrial Electronics, 2006, 53(3): 941-949.

[33] Wang C, Li X, Zhou X. SODA: Software defined FPGA based accelerators for big data[C]//2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2015: 884-887.

[34] Xia K, Luo Y, Xu X, et al. Sgx-fpga: Trusted execution environment for cpu-fpga heterogeneous architecture[C]//2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021: 301-306.

[35] Gross M, Hohentanner K, Wiehler S, et al. Enhancing the Security of FPGA-SoCs via the Usage of ARM TrustZone and a Hybrid-TPM[J]. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2021, 15(1): 1-26.

[36] Rajendran J, Sinanoglu O, Karri R. Is split manufacturing secure?[C]//2013 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2013: 1259-1264.

[37] Roy J A, Koushanfar F, Markov I L. EPIC: Ending piracy of integrated circuits[C]//Proceedings of the conference on Design, automation and test in Europe. 2008: 1069-1074.

[38] Haider S K, Jin C, Ahmad M, et al. Advancing the state-of-the-art in hardware trojans detection[J]. IEEE Transactions on Dependable and Secure Computing, 2017, 16(1): 18-32.

[39] Benhani E M, Bossuet L, Aubert A. The Security of ARM TrustZone in a FPGA-based SoC[J]. IEEE Transactions on Computers, 2019, 68(8): 1238-1248.

[40] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," 2019.

[41] Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Verified low-level programming embedded in F*[J]. arXiv preprint arXiv:1703.00053, 2017.

[42] Nir Y, Langley A. ChaCha20 and Poly1305 for IETF Protocols[R]. 2018.

[43] Bernstein D J. The Poly1305-AES message-authentication code[C]//International workshop on fast software encryption. Springer, Berlin, Heidelberg, 2005: 32-49.

[44] Chlipala A. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier[C]//Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. 2013: 391-402.

[45] Chlipala A. Mostly-automated verification of low-level programs in computational separation logic[C]//Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011: 234-245.

[46] Xilinx Vitis HLS, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[47] Cong J, Liu B, Neuendorffer S, et al. High-level synthesis for FPGAs: From prototyping to deployment[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2011, 30(4): 473-491.

[48] Canis A, Choi J, Aldham M, et al. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems[J]. ACM Transactions on Embedded Computing Systems (TECS), 2013, 13(2): 1-27.

[49] Pilato C, Ferrandi F. Bambu: A modular framework for the high level synthesis of memory-intensive applications[C]//2013 23rd International conference on field programmable logic and applications. IEEE, 2013: 1-4.

[50] Grigoraş P, Niu X, Coutinho J G F, et al. Aspect driven compilation for dataflow designs[C]//2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors. IEEE, 2013: 18-25.

[51] Peverelli F, Rabozzi M, Del Sozzo E, et al. OXiGen: a tool for automatic acceleration of c functions into dataflow FPGA-based kernels[C]//2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2018: 91-98.

[52] Cong J, Wang J. PolySA: Polyhedral-based systolic array auto-compilation[C]// 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2018: 1-8.
[53] Liu J, Wickerson J, Bayliss S, et al. Polyhedral-based dynamic loop pipelining for high-level synthesis[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 37(9): 1802-1815.
[54] Seacord R C. The CERT C secure coding standard[M]. Pearson Education, 2008.
[55] Hatton L. Safer language subsets: an overview and a case history, MISRA C[J]. Information and Software Technology, 2004, 46(7): 465-472.
[56] Love E, Jin Y, Makris Y. Enhancing security via provably trustworthy hardware intellectual property[C]//2011 IEEE international symposium on hardware-oriented security and trust. IEEE, 2011: 12-17.
[57] Love E, Jin Y, Makris Y. Proof-carrying hardware intellectual property: A pathway to trusted module acquisition[J]. IEEE Transactions on Information Forensics and Security, 2011, 7(1): 25-40.
[58] Venet A, Brat G. Precise and efficient static array bound checking for large embedded C programs[J]. Acm Sigplan Notices, 2004, 39(6): 231-242.
[59] Jourdan J H, Laporte V, Blazy S, et al. A formally-verified C static analyzer[J]. ACM SIGPLAN Notices, 2015, 50(1): 247-259.
[60] Viega J, Bloch J T, Kohno Y, et al. ITS4: A static vulnerability scanner for C and C++ code[C]//Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00). IEEE, 2000: 257-267.
[61] Palmer D W, Manna P K. An efficient algorithm for identifying security relevant logic and vulnerabilities in rtl designs[C]// 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). IEEE, 2013: 61-66.
[62] Meng X, Kundu S, Kanuparthi A K, et al. Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021, 41(3): 466-477.
[63] Arias O, Liu Z, Guo X, et al. RTSec: automated RTL code augmentation for hardware security enhancement[C]//2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2022: 596-599.
[64] Lyu Y, Mishra P. Automated test generation for activation of assertions in RTL models[C]//2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2020: 223-228.
[65] Zhang R, Deutschbein C, Huang P, et al. End-to-end automated exploit generation for validating the security of processor designs[C]//2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018: 815-827.
[66] Snyder W. Verilator and systemperl[C]//North American SystemC Users' Group, Design Automation Conference. 2004.
[67] Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs [C]//OSDI. 2008, 8: 209-224.
[68] Muttaki M R, Ibnat Z, Farahmandi F. Secure by construction: addressing security vulnerabilities introduced during high-level synthesis[C]//Proceedings of the 59th ACM/IEEE Design Automation Conference. 2022: 1371-1374.
[69] Giamblanco N V, Anderson J H. asap: Automatic sizing and partitioning for dynamic memory heaps in high-level synthesis[C]//2019 International Conference on Field-Programmable Technology (ICFPT). IEEE, 2019: 275-278.
[70] Rhodes D L, Wolf W. RAGS-real-analysis ALAP-guided synthesis[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2001, 20(8): 931-941.
[71] Pilato C, Garg S, Wu K, et al. Securing hardware accelerators: A new challenge for high-level synthesis[J]. IEEE Embedded Systems Letters, 2017, 10(3): 77-80.
[72] Jiang Z, Dai S, Suh G E, et al. High-level synthesis with timing-sensitive information flow enforcement[C]//2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2018: 1-8.