HIVE: Scalable Hardware-Firmware Co-Verification using Scenario-based Decomposition and Automated Hint Extraction

Aruna Jayasena, *Student Member, IEEE,* and Prabhat Mishra, *Fellow, IEEE,*

***Abstract*—Hardware-firmware co-verification is critical to design trustworthy systems. While formal methods can provide verification guarantees, due to the complexity of firmware and hardware, it can lead to state space explosion. There are promising avenues to reduce the state space during firmware verification through manual abstraction of hardware or manual generation of hints. Manual development of abstraction or hints requires domain expertise and can be time-consuming and error-prone, leading to incorrect proofs or inaccurate results. In this paper, we effectively combine the scalability of simulation-based validation and the completeness of formal verification. Our proposed approach is applicable to actual firmware and hardware implementations without requiring any manual intervention during formal model generation or hint extraction. To reduce the state space complexity, we utilize both static module-level analysis and dynamic execution of verification scenarios to automatically generate system-level hints. These hints guide the underlying solver to perform scalable equivalence checking using proofs. The extracted hints are validated against the implementation before using them in the proofs. Experimental evaluation on RISC-V based systems demonstrates that our proposed framework is scalable due to scenario-based decomposition and automated hint extraction. Moreover, our fully automated framework can identify complex bugs in actual firmware-hardware implementations.**

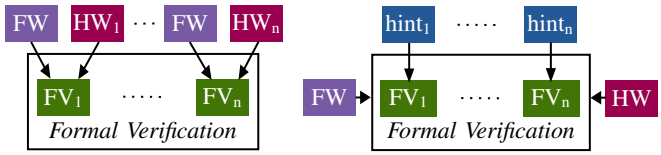
***Index Terms*—Simulation-based Validation, Formal Verification, Firmware Verification, Test Generation, Hint Extraction**

I. Introduction

System-on-Chip (SoC) architectures consist of components that are implemented as both hardware and firmware. External IPs such as cryptographic accelerators, and communication modules such as UART, I^{2}C and SPI modules use memory-mapped input/outputs (MMIO) to communicate with the processor. These devices are typically connected with the firmware applications and the device drivers. Both firmware applications and device drivers are usually written in *C* language. Once the application is ready, they are compiled with the device drivers using the toolchain corresponding to the processor in the SoC. Next, a customized linker script informs the compiler about the memory configurations used in the setup to obtain the final compiled binary. An efficient and scalable framework is needed to verify such a complex system across all these layers of hardware, drivers, and applications.

There are promising test generation-based validation techniques for functional and security verification of SoCs [1], [2], [3], [4]. These techniques either rely on the abstraction of the hardware-firmware interactions [3] or applicable on specific applications [5]. Due to exponential input space complexity, test generation based techniques are likely to miss corner cases, and therefore cannot provide a correctness guarantee. For example, we need to simulate a 64-bit adder (adds two 64-bit inputs) with 2^{128} test vectors to provide the verification guarantee. Clearly, it is infeasible to simulate a real-world design with all possible input vectors.

Formal verification methods [6], [7], [8], [9], [10], [11] can provide strong guarantees about the correctness of a system. While formal verification is not affected by input space complexity, it can lead to state space explosion when dealing with large and complex systems [12]. As a result, the scalability of formal verification is limited to small parts of the design due to the fact that symbolic expression term size grows exponentially with the increasing design size. There are promising efforts for system-level formal verification that rely on the designer's expertise to guide the proofs with manual hints [7], [8] to partition the state space and reduce the growth of expression term size.



(a) Manual Abstraction (b) Manual Hints

Fig. 1: Existing formal firmware verification avenues

Existing firmware verification efforts explore two promising avenues to reduce the state space as illustrated in Figure 1. The methods in the first category perform firmware verification on top of multiple manually abstracted versions of the hardware instead of register-transfer level (RTL) implementation. For example, Figure 1a shows that to perform formal verification (FV) of a firmware (FW) for n scenarios, a designer needs to manually construct the abstracted hardware model (HW_i) customized for the i-th scenario. Manual abstraction based on verification scenarios needs domain expertise and can be erroneous. Moreover, discrepancies between the actual implementation and the abstract model can lead to false positive results. The methods in the second category perform automated formal model generation for hardware, but it relies on manual extraction of hints from both firmware and hardware to guide formal proofs to avoid state explosion due to rapidly increasing term size. For example, Figure 1b shows that a designer needs to manually construct the hint ($hint_i$) customized for the i-th scenario. Manual development of hints can be time-consuming and error-prone, leading to incorrect proofs or inaccurate results.

A. State-of-the-Art in State Space Reduction

An inherent weakness of formal verification is state space explosion, where the backend solver has too many possibilities

A. Jayasena and P. Mishra are with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, Florida.

This work was partially supported by the grants from NSF (CCF-1908131) and Semiconductor Research Corporation (2022-HW-3128).

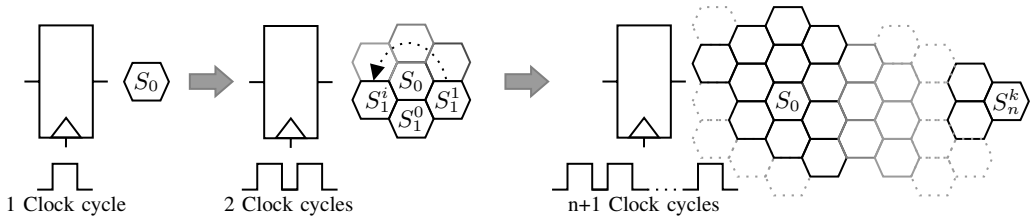


Fig. 2: State space expansion of a state register of a hardware implementation with respect to the evaluated clock cycles.

to explore at once. In a system that combines both hardware and firmware, there are multiple ways that state explosion can occur. Due to the concurrency of hardware, state registers can increase rapidly in each clock cycle in which it gets evaluated. Figure 2 illustrates an example scenario for a state register. When the number of unrolling cycles for the design increases, the number of possible paths that the state register can represent increases exponentially. Similarly, the number of symbolic variables also increase linearly when the implementation size increases. Moreover, firmware consists of branch statements which increases the number of possible states the system can have. These factors contribute to the symbolic state explosion and path explosion [13], [14] during formal verification.

In order to guide the formal proofs without getting into the state explosion problem, there are promising state space reduction techniques. These techniques aim to reduce the size of the state space by identifying and eliminating states that are either irrelevant or equivalent to other states for a given verification context. The goal is to make the verification process more efficient and effective by reducing the number of states that need to be analyzed. We outline four popular approaches for state space reduction.

1) Abstraction: Abstraction involves simplifying the system model by removing details that are not relevant to the properties being verified [15]. This can include abstracting away certain variables, simplifying the system topology, or reducing the level of details in the system model. Usually, abstraction is a manual process that relies on the expertise of the verification engineer. Quality and the similarity of the abstraction determine the guarantees and accuracy of the proofs. In other words, if the abstraction does not represent the system correctly, verification can lead to false positive results.

2) Symmetry Reduction: Symmetry reduction involves identifying and eliminating symmetric states in the system model [15]. Symmetric states are states that are equivalent to other states under certain transformations, such as swapping the values of variables or reordering events.

3) Partial-order Reduction: Partial order reduction involves analyzing only a subset of the possible orderings of events in the system model [15]. This can be achieved by identifying independent events that can be executed in any order, and then only analyzing a relevant subset of the possible orderings.

4) Decomposition: Decomposition enables state-space reduction by breaking a problem into multiple sub-problems [15]. Instead of going through the manual and tedious task of abstraction for system validation purposes, our proposed technique divides the main verification problem into sub-problems. Then symmetric reduction and partial order reduction techniques are directly applied to each sub-problem. Our sub-problem decomposition technique adheres the basic principles of assume-guarantee [16], [17] and compositionality [18], [17]. Assume-guarantee defines the requirements that should exist in a system to decompose them into multiple sub-problems while providing the system-level guarantee. Compositionality refers to the property of a system or proof that allows it to be divided into smaller, independent parts or sub-problems, which can be analyzed or verified separately, and then combined to form a complete solution.

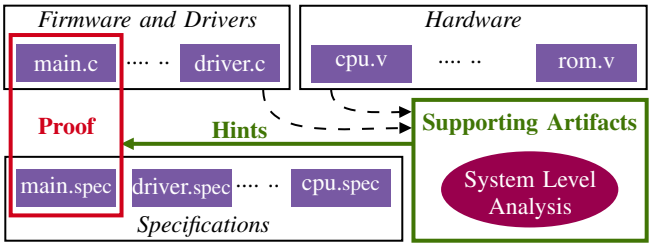


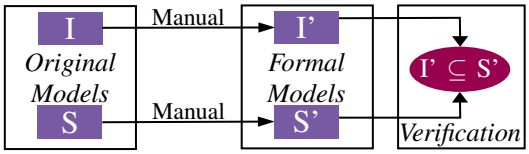
Fig. 3: Overview of our automated *Hint-based Verification* (*HIVE*) framework. Formal proofs are supported by hints.

B. Research Contributions

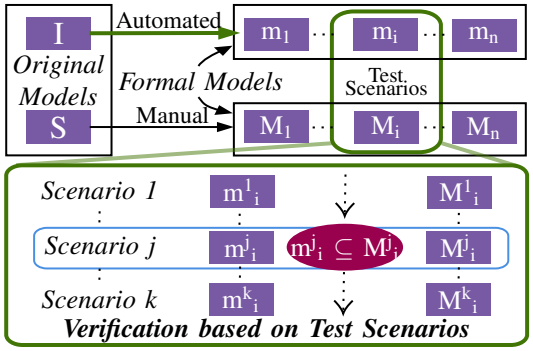
In this paper, we effectively combine the rigor and completeness of formal verification and the scalability of test generation based validation. We simplify the system validation problem by dividing it into sub-problems. For each sub-problem, we check the consistency of the implementation against the specification. This covers input output consistency of sub-problems and ensures that the implementation satisfies the specification. To further reduce the state space for each sub-problem, we utilize both concrete simulation and static analysis of the designs to extract proof supporting artifacts. Figure 3 presents an overview of our proposed framework, *HIVE* (*Hint-based Verification*), where proof supporting artifacts required for sub-problems are extracted through static analysis of the implementation as well as simulation of relevant test scenarios.

Figure 4 highlights the novelty of our methodology compared to the existing approaches. Figure 4a shows that the existing methods rely on manual abstraction of the hardware implementation, but it can lead to false positive results. As shown in Figure 4b, our proposed approach has two objectives: (i) automated formal model generation to avoid false positive results, and (ii) scenario-based verification to reduce the state space specific to the verification scenarios. Here, M_i and m_i represent the i-th component (module) in the specification (S) and implementation (I), respectively. Similarly, M_i^j and m_i^j represent the i-th module customized for the j-th scenario in the specification and implementation, respectively.

Specifically, this paper makes the following contributions.



(a) Existing approaches suffer from state explosion



(b) Proposed method utilizes automated scenario-based analysis.

Fig. 4: *HIVE* utilizes both module-level and test scenario-based decomposition for improving the scalability of proofs.

- It enables automated generation of formal models from the system-level implementation. We compile the firmware and combine it with the hardware to obtain the system model.
- To avoid state space explosion, it decomposes the system-level equivalence checking problem into sub-module level equivalence checking problems. We utilize function-level boundaries of firmware and module-level boundaries of hardware to perform module-level equivalence checking.
- We utilize both architectural and behavioral models through static analysis of modules and dynamic analysis of the implementation. This leads to the automated generation of module interactions and supporting artifacts.
- It reduces the state space further by scenario-based analysis. We utilize supporting artifacts to generate system-level hints and perform path prioritization and symbolic state reduction related to the scenario under validation. The generated hints guide the equivalence-checking proofs in the simplified state space specific to the scenario as illustrated in Figure 4b.

Our proposed framework leverages the strengths of both formal verification and simulation-based validation for scalable validation of firmware running on hardware (RTL) implementation. The design slicing feature of concrete simulation-based validation is combined with symbolic simulation to formally evaluate the implementation against target verification scenarios. Since *HIVE* generates the formal model automatically from the implementation, it is a complete model by construction relative to the implementation. Any issue found during verification can be corrected directly on the implementation and re-validated using our proposed method. This is in contrast with the existing methods that require fixes and re-validation of multiple (abstract and implementation) models.

The rest of the paper is organized as follows, First, we discuss the background and related work. Next, we discuss the *HIVE* methodology in detail. Then, we present experimental results with three case studies. Finally, we discuss the applicability and limitations of the proposed framework.

II. Background and Related Work

In this section, we first survey formal methods that utilize manual abstraction or manual generation of hints for firmware verification. Next, we discuss firmware validation methods using concolic testing.

A. Verification using Manual Abstraction

Blom et al. have identified overlapping states in a formal model based on the cone of confluence [19]. The authors propose a prioritization algorithm to identify and detect confluence properties of the models under verification. Yorav et al. have utilized the control-flow graph to simplify the transition diagram for validation of parallel programs [20]. Vasudevan et al. have reduced the state space through antecedent-conditioned slicing for hardware designs [21]. Their technique reduces the state space through abstraction, and it can be utilized for the verification of safety properties in the form of $G(A \Rightarrow C)$, where A (antecedent) and C (consequent) are propositional logic formulas. An improved version of the algorithm utilizes the extra information from the antecedent to prune and limit the state space further for a particular problem [22]. Analysis of Verilog designs with transaction-like protocol descriptions to reduce the state space for verification of protocols is discussed in [23]. The authors consider different SystemVerilog constructs with protocol-targeted assumptions for the abstraction of design and define and limit the state space of the verification problem. Another framework to validate *SystemC* models utilizing a *SystemC* intermediate representation is proposed in [24]. The authors perform static analysis on formal models and perform model slicing with abstraction to reduce the state space. None of these approaches consider state space reduction in the context of hardware-firmware verification.

Instruction Level Abstraction (ILA) is a popular technique used to formally verify firmware in System-on-Chip (SoC) designs [9], [10], [11], [25], [26]. ILA abstracts a processor design at the instruction level for verification purposes, which allows the verification of the design to be performed at a higher level of abstraction than the gate-level or RTL-level implementation. ILA involves modeling the processor's behavior as a set of instructions that can be executed by the processor, rather than modeling the behavior of the individual gates and wires that make up the processor. ILA can also enable the effective reuse of verification efforts across different processor designs, as the instruction-level model can be reused for different implementations of the same processor architecture. Note that ILA generation is a manual and tedious task that requires a thorough understanding of the particular processor design. Moreover, incorrect ILA will lead to inaccurate results. Furthermore, in order to verify external accelerators and modules that connects with the SoC, separate abstraction for their implementations are required (similar to Figure 1a).

B. Formal Verification using Manual Hints

RTLV [8] is a formal verification framework that utilizes hybrid symbolic execution to verify firmware that runs on

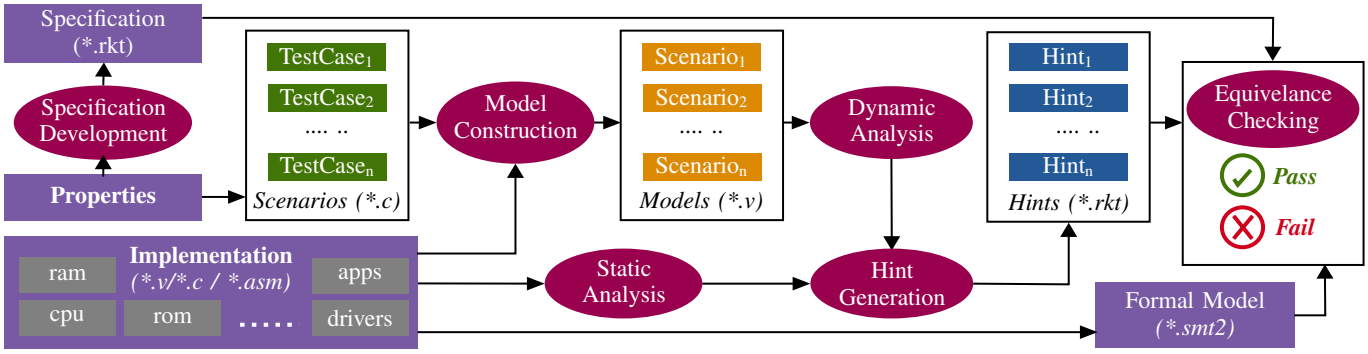


Fig. 5: Overview of *HIVE* framework that consists of five main steps: outlining test scenarios, system model construction, supporting artifact generation by static and dynamic analysis, hint generation, and equivalence checking.

the hardware. It utilizes *Rosette* solver-aided programming language to perform the symbolic evaluation. Compared to the existing frameworks, it can handle complex RISC-V-based CPUs with unrolling over 20,000 clock cycles. In order to guide the proofs, the authors have used manually developed hints that are specific to the design under verification (similar to Figure 1b). Athalye et al. have developed a framework for validation of hardware security modules (HSM) with information refinement [7]. The authors have performed equivalence checking on three simple HSM designs implemented with both hardware and firmware. The proposed technique has several limitations. To avoid the state explosion, the authors have manually provided hints to reduce the rapidly growing symbolic term space. Due to the fact that the verification problem is posed as one verification problem to the underlying solver, complex design aspects such as cryptographic calculations cannot be handled with the proposed technique due to solver timeouts.

The existing approaches rely on manual hints [7], [8], which requires significant expertise and can be error-prone. An incorrect hint would add more complexity to the solver in terms of state space and will eventually fail. This highlights the need for automated generation of proof supporting artifacts as well as system-level hints for scalable firmware validation.

C. Firmware Validation using Concolic Testing

Ahn et al. presented a concolic testing framework based on the concept of consumer-producer relationships between hardware and firmware [1]. Their technique combines the transaction-level models (TLM) of the implementation, which are implemented in *SystemC*, with the firmware implemented in *C* for the evaluation. The symbolic execution is performed with KLEE [27]. The authors remove the concurrent nature of the firmware by modeling the interactions as sequential events through mapping interactions to the consumer and producer models. Concolic testing approach based on virtual prototypes for verification of hardware and software models is proposed in [28]. The authors utilize virtual device models as the hardware model and run the software on the virtual device model while collecting traces. These traces are then utilized to guide the test generation process in the required direction. Although the concolic testing framework is effective in generating test cases to activate corner cases, these techniques also require virtual prototype models of the hardware implementations. Note that the generated tests are abstract tests, which may not be directly applicable to actual implementation. Moreover, the virtual prototype model needs to be in a form that can be analyzed with existing concolic testing tools that were developed for software verification. Therefore, these approaches are not suitable for firmware validation on top of hardware (RTL) implementation.

There are promising efforts that enable concolic testing on RTL-level hardware implementations [29], [30], [31]. These techniques operate on the RTL-level hardware models and the applicability is based on the controllability of the design with respect to the inputs of the designs. A major limitation of applying these techniques for hardware-firmware validation is that most of the hardware-firmware interactions are controlled through the firmware. Therefore, concolic testing needs to generate and modify the parts of the firmware, which is infeasible due to the complexity and the device-specific nature (memory layout and instruction set) of the firmware.

III. HIVE: Automated and Scalable Hint-based Formal Verification

Figure 5 provides an overview of our proposed automated Hint-based Verification (*HIVE*) framework. The major steps of the framework are outlined in Algorithm 1. First, we need to consider the verification scenarios and associated tests to cover these scenarios, this is the input to the algorithm. Next, the firmware tests are compiled and corresponding binaries are obtained (Line 1-7). Then, we construct the system model (Line 8). Next, we extract the proof supporting artifacts from the designs using dynamic analysis and perform automated hint extraction (Line 9-15). Finally, for each simplified sub-problem, we utilize the generated hints to perform formal verification (Line 16-21). The remainder of this section describes each step in detail. We use the SoC implementation shown in Figure 6 to describe each step of Algorithm 1 using illustrative examples.

Example 1 (SoC): *Consider a firmware configurable traffic light controller design implemented on a RISC-V-instruction-set-based SoC, as shown in Figure 6. The SoC design is implemented in Verilog and consists of CPU, ROM, RAM, UART, and TLC modules as illustrated in the figure. In this example, firmware is able to control timing parameter values for different states of the traffic light controller, and corresponding state machine transitions are defined in the hardware description.*

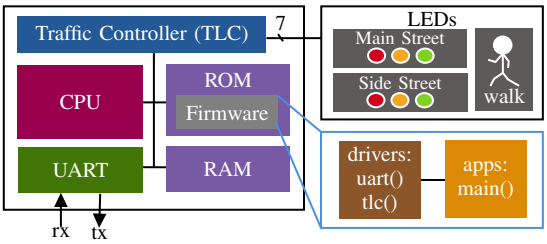


Fig. 6: SoC used for a traffic light controller.

The traffic light controller is connected to the processor of the SoC with memory-mapped I/O. There are two separate drivers for controlling the UART and TLC from the firmware. Further, all the sensor data is communicated to the SoC via an external device via the UART communication interface. Once the reset of the controller is triggered, the traffic light system starts to work with the newly configured values. As illustrated in Figure 6, the system output is represented by the LED signals which is a 7-bit vector. ■

Algorithm 1 *HIVE* Algorithm

Inputs: Scenarios: $T\{t_1, t_2, ..., t_n\}$, RTL: $R\{r_1, r_2, ..., r_n\}$, Drivers: d
Outputs: Verification: $V\{v_1, v_2, ..., v_n\}$

for each t in T **do** ▷ Section III-A
  $f \leftarrow compile(d + t)$
  $F.append(f)$
  for Each r in R **do**
    Spec.append([t,behavior]) ▷ Section III-B
  end for
end for
$M \leftarrow$ flattened(R) ▷ Section III-C
for each f in F **do**
  trace $\leftarrow simulate(f + M)$
  $S \leftarrow signalRanking(trace)$
  $a \leftarrow extractArtifacts(S, R, M)$ ▷ Section III-D
  $h \leftarrow hintGeneration(S, M(a), \tau)$ ▷ Section III-E
  $H.append(h)$
end for
I $\leftarrow smt(R, d)$ ▷ Section III-G
for each r in R **do**
  for each t in T **do**
    V.append($I(H_t^r) \subseteq Spec_t^r$) ▷ Pass/ Fail
  end for
end for
Return V

A. Test Cases for Verification Scenarios

In this step, we outline the functional scenarios that we want to provide guarantee with formal verification. For this purpose, we utilize the same test plan that is typically developed by designers to test the firmware. This may include unit tests for each of the components in the design under test, which are written in C or assembly (**.c/*.asm*) for testing the firmware. *HIVE* can effectively use these unit tests to identify the behavior of the model related to the specific test scenarios. Note that one test case would be enough to state one test scenario. For example, if the firmware uses a 64-bit adder (one scenario), we only need to specify one testcase (e.g., adding 2 and 3 should produce 5) for verifying the addition scenario instead of 2^{128} testcases. These tests should cover all the scenarios that are expected to get formally verified.

After writing each of the test cases, we compile the firmware with device drivers using the relevant toolchain of the processor instruction-set architecture. Note that we keep only the test case as the main function of the SoC during this process. This will generate a separate firmware for each of the verification scenarios. Next, we need to simulate the design to obtain the behavioral model of the system for each scenario.

Example 2 (Test Cases): *Let us verify the functionality of the traffic light controller. We have to write test cases for the major verification scenarios. In this example, we have considered four simple test scenarios: SoC requesting sensor data from an external device (**Scenario 1**), SoC reading sensor data from the external device (**Scenario 2**), firmware requesting the traffic light controller for the side street green light (**Scenario 3**), and firmware requesting a pedestrian crossing instance for the main road (**Scenario 4**). These verification scenarios and corresponding test cases are derived from the test plan. Listing 1 presents the test case written in the firmware in c language for the first scenario. These test cases will be compiled separately with the device drivers in order to obtain four separate binary/hex files corresponding to each test scenario.* ■

Listing 1: Firmware test case related to Scenario 1.

```
#include <stddef.h>
#include "driver.h"

int main(void) {
    char msg[] = "request()\r\n";
    uart_init(); // Initialize UART
    uart_write(msg, strlen(msg)); // Write msg to UART
    return 0;
}
```

B. Development of Specification

Once we have each of the test scenarios, we need to capture the expected behavior of the system in a formal model. We write the functionality of each of the components with respect to each scenario. We put them together to obtain the formal model of the specification.

Listing 2: Specification of UART module for Scenario 1.

```
(define (send-byte byte)
  ; Initialize sending process
  (send-bit #f)
  ; sending the data as bits
  (for-range 0 8 (lambda (i) (send-bit
        (bitvector->bool (extract i i byte)))))
  ; Ending the message
  (send-bit #t))
....
```

Example 3 (Specification): *Since Example 2 has four scenarios, we need to write four specifications for each of the module in the SoC (Example 1). Listing 2 shows a part of the specification that we have developed to capture the expected behavior of UART for the first scenario in the Racket programming syntax (*.rkt). In the example code, the* `send-byte` *function initiates a data transmission process, sends each bit of an input byte sequentially, and concludes the transmission by signaling the end of the message. The actual bit transmission is delegated to an external function named* `send-bit` *which describes the low level specification of the data transmission.* ■

C. Automated Generation of System Models

At this stage, we have compiled a binary for each of the verification scenarios, which is ready to be simulated with the implementation. In order to proceed with the simulation of each scenario, we construct the system model. For this, we include the binary in the hardware description (e.g., Verliog allows reading compiled binary as a memory image). At this point, if we simulate the design, all the tasks that are implemented in the firmware will be simulated. This system model (firmware+hardware) file is then flattened to remove the design hierarchy. This enables equivalence checking (via symbolic simulation) of the entire implementation at the final stage with both hardware and firmware (driver/software). It is important to keep the signal names preserved with the hierarchy during this stage since we need to align the dynamic simulation-related behavioral details with module-level architectural details. This process is repeated for all the firmware binaries that we have created in previous step. This results in multiple copies of system models with the same hardware, implementing each test scenario in the firmware separately.

Next, we need to perform dynamic analysis. Dynamic analysis is performed by analyzing the trace data after performing the concrete simulations. For this purpose, we automatically generate the testbenches to simulate the SoC. This also includes the template to include external inputs to the system, if required for the particular test case. The objective of this step is to create a functional system model that passes the particular test case.

Example 4 (System Models): *For the example SoC, we have four verification scenarios in Example 2. As a result, we will have four separate system models which contain relevant test case firmware for the relevant scenario. Figure 7 illustrates the individual system models for dynamic analysis (concrete simulation) while module-level implementation is utilized for static analysis.* ■

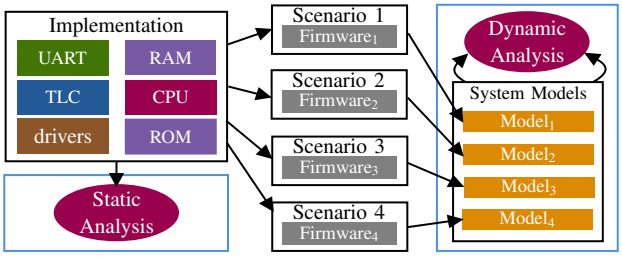


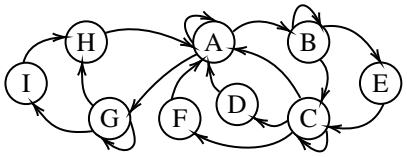
Fig. 7: System models generated for the example SoC.

D. Generation of Supporting Artifacts

We define proof-supporting artifacts as objects that can be derived from the system model. *HIVE* utilizes both module-level static analysis and system-level dynamic analysis to extract supporting artifacts. We first describe these two analysis techniques. Next, we discuss how to combine the extracted artifacts to infer meaningful details about the implementation.

1) ***Module-Level Static Analysis****:*
The purpose of the static analysis is to identify the structure of the implementation. This includes information such as signal dependencies, path conditions, and state register details. In order to identify these details, *HIVE* extracts the abstract syntax tree (AST) of each module. AST contains information about control flow and data flow related to the implementation of the module. Further, *HIVE* extracts the Finite-State-Machine (FSM) details present in the module implementation.

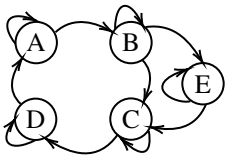
Example 5 (Static Analysis Artifacts): *For the SoC in Example 1, HIVE was able to automatically extract all the module level ASTs along with two main FSMs related to the validation scenarios. Figure 8a shows the FSM extracted from the UART module while Figure 8c shows the FSM extracted from the TLC module. Note that these figures show the visual representation of the FSMs. However, HIVE internally represent them as comma delimited ASCII text file (*.kiss) format. Figure 8b and Figure 8d provides the details extracted from the specification to identify each state in the FSM. For example,* $Stop_1$*,* $Stop_2$*, and* $Stop_t$ *refer to transmitter communication halt states at 1/2 byte, byte, and final stop, respectively. Similarly,* $Stop_r$ *refers to the receiver's final stop. Likewise,* $Shift_t$ *and* $Shift_r$ *refer to transmitter and receiver shift states, respectively. Finally,* $Parity_t$ *and* $Parity_r$ *refer to transmitter and receiver parity check states, respectively.* ■



(a) UART module state machine

Type		Transmitter					Receiver		
State	A	B	C	D	E	F	G	H	I
Desc.	Start	$Shift_t$	$Stop_1$	$Stop_2$	$Parity_t$	$Stop_t$	$Shift_r$	$Stop_r$	$Parity_r$

(b) UART module specification



(c) TLC module state machine

State		**Description**
#	**Value**	
A	0011000	Main St. green
B	0101000	Main St. yellow
C	1000010	Side St. green
D	1000100	Side St. yellow
E	1001001	Road Crossing

(d) TLC module specification

Fig. 8: Finite State Machine (FSM) extracted from the UART and TLC modules by *HIVE* and the relevant specifications.

2) ***System-Level Dynamic Analysis****:*
The purpose of the dynamic analysis is to automatically identify the behavioral model of the system for a specific scenario. *HIVE* simulates the designs with the system models generated in Section III-C and obtains the simulation traces related to each scenario separately. Next, the following procedure is applied to the simulation trace of each model separately.

Each of the signals in the model is analyzed and ranked according to their value change frequency. Algorithm 2 illustrates the main steps involved in the signal ranking process. First, it extracts the signal list in the execution trace and generates a data structure to store the history related to each signal. Next, it analyses the simulation trace while appending the concrete value of each signal observed in the simulation, followed by sorting the final signal list based on the increasing order of value change frequency observed for each signal. Note that all the unknown logic (uninitialized) value states are considered for weakening and ranked last, while high-impedance states are taken into account for the signal ranking

process. The ranking procedure serves as the heuristic for the hint generation process to identify the hint type corresponding to signals (*i.e.*, if a certain signal is changed less than a certain threshold (τ), consider for abstraction).

Algorithm 2 signalRanking(vcd)

Input: Value Change Dump (VCD) Traces: vcd
Output: Ranked Signals: S

```
1: Signals ← extractSignals(vcd)
2: for each trace in vcd do
3:     {signal, value} ← trace
4:     Signals<signal>.append(value)
5: end for
6: S ← sort Signals based on length(value)
7: Return S
```

Example 6 (Dynamic Analysis Artifacts): *After performing dynamic analysis on models in Example 4, HIVE obtains a data structure related to the signal history of each of the models and ranks the signals. Figure 9 illustrates the different percentages of signals for different frequencies of signal value changes found for each model separately.* ■

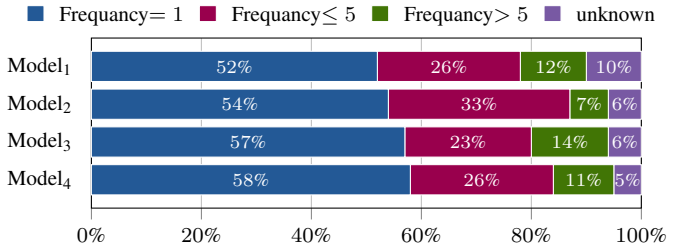


Fig. 9: Signal ranking calculated by *HIVE* on different test cases by analysis of simulation traces corresponding to each concrete simulation scenario. ($\tau = 5$)

*3) **Signal Alignment**:* After performing both static analysis and dynamic analysis, we need to combine the architectural model with the behavioral model of the design. *HIVE* aligns the control flow graph (CFG) of the flattened netlist with each sub-module CFG by referring to the signal names. From CFG, it extracts the term dependency graph for each of the state variables in sub-modules. Then from the sub-modules which contain FMSs, state register behavior is aligned with the FSM. Later this is used for path prioritization during formal proofs.

Example 7 (Signal Alignment): *Flattened implementation of Example 1 SoC contains all signal names with hierarchy. If we consider the signal "soc.uart.recv_buf_data" of the flattened netlist, this represents the signal "recv_buf_data" of the UART module. HIVE automatically decodes such information from the signal name and maps it with the corresponding module-level signal.* ■

E. Generation of System-level Hints

In *HIVE*, hint generation is a fully automated process, where the user does not need to know implementation details about the design. Firmware is typically developed on top of hardware design created by third parties, and verification of these system environments is made possible by our proposed technique. In this section, we discuss the process of generating effective hints to simplify proofs by utilizing the proof-supporting artifacts derived in Section III-D. This step of *HIVE* serves two purposes. First, it automatically identifies each variable in the design to be used as a state or symbolic variable, and generates abstract syntax for each of the hints by analyzing the proof supporting artifacts. Next, it translates abstract syntax into a form that can be used by the formal verification framework.

Algorithm 3 illustrates the main steps involved in the module-level hint generation process. First, it iterates through the modules available in the implementation of the SoC. If there is an FSM in the module, it aligns the corresponding state register with the ranked signal list. This signal is analyzed against proof supporting artifacts to perform path prioritization (discussed in Section III-E2). All the other signals in the module are also aligned with the corresponding signal from the ranked signal list. If there are multiple instantiations of the same component, those will have multiple occurrences corresponding to each alignment. Then the aligned signals are added to the hint list corresponding to the module and categorized based on the prospect hint type. First, it evaluates the signals whose value got changed only once during the concrete simulation. These are candidate signals to consider for concretization. These signals are analyzed with the FSM states and path conditions. Specifically, a signal is appended for concretization if it got its value assignment in a state/path that is not in the current state/path conditions for the scenario under test.

In *HIVE*, the user has the ability to control the threshold (τ) to consider a signal for overapproximation (default $\tau = 5$, and this gets verified on the system model). All the uninitialized signals are considered for weakening. Since the back-end solver only has access to a single sub-problem at a time, the hint synthesis step performs solver queries on the system model in order to verify hints before committing them during the equivalence checking proofs.

*1) **Elaboration of Hint Types**:*
Performing modifications to the symbols involved in the formal proofs such that they have more constraints can reduce the state space of the verification problem. For that purpose, *HIVE* performs the following types of symbol alterations:
Concretization: The process of concretization involves replacing symbolic values with concrete values that represent specific values that the system or component can take during execution. This allows verification of the system or component using specific input values relevant for the scenario under test, rather than analyzing the behavior of the system or component under all possible input values. Since we have the system model and the hint list generated by Algorithm 3, we perform system-level queries from the solver to verify the variables that can be concretized before using them in module-level proofs.
Weakening: Weakening a variable will effectively prune the formal model of the system by imposing constraints for that variable. This reduces the size of the symbolic execution tree and improves the efficiency of the symbolic execution by considering only the most relevant path to the specific scenario under consideration.

Algorithm 3 hintGeneration(S, M, τ)

Input: RankedSignals S, Modules $M(s, fsm)$, Threshold. τ
Output: Hints H

```
for each m in M do
    stateVars ← S{state(m< fsm >)}  ▷ get FSM state register
    vars ← S{m<s>}                  ▷ Signal alignment
    p ← pathPrioritization(S, stateVars)
    tempH<weaken>.append(p)
end for
tempH, H ← ∅
tempH<sRegs>.append(stateVars)
for each v in Vars do
    if length(v.value)==1 then
        if assignment(v) ∉ current path condition then
            tempH<concrete>.append(v)
        end if
    else if length(v.value) ≥ τ then
        tempH<overaprx>.append(v)
    else if length(v.value) < τ then
        d ← getDepends(v)
        if allowed(d) then
            tempH<abstract>.append(v)
        end if
    else if uninit(v.value) then
        tempH<weaken>.append(v)
    end if
end for
for each h in tempH do
    if verify(h) then           ▷ Verify hints on the system model
        H.append(h)
    end if
end for
Return H
```

Overapproximation and Abstraction: The overapproximation process will replace a symbolic variable that is propagated through the system with a new variable. This makes the analysis less precise but more tractable. Abstraction will replace the variable with a symbolic term if the variable only depends on allowed dependencies, which starts with the module inputs, and adds the symbol to the allowed dependencies.

Example 8 (Generated Hints): *Figure 10 illustrates the percentage of signals considered for different hint types for the four scenarios outlined in Example 2. After considering each of the signals as a hint for different modules, they are verified against the system model to check their validity.* ■

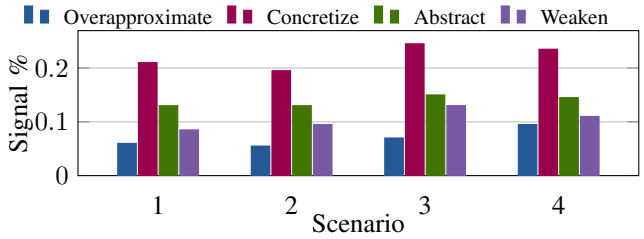


Fig. 10: Types of hints extracted and verified by *HIVE* on different test scenarios and % of hints on the system model.

*2) **Path Prioritization for Scenario Analysis**:*
The goal of path prioritization is to focus the verification effort on the most relevant parts of the design. Most important paths are determined based on the test scenarios that are provided in the firmware application in Section III-A. During symbolic evaluation, these paths will get priority in the state space. Further, each scenario will be evaluated as a separate sub-problem. Scenario-based decomposition is particularly useful when dealing with complex control flow or data structures, where it may be difficult to analyze all possible behaviors in a single case. By dividing the problem into multiple scenarios, the verification effort can be focused on specific parts of the system, making the analysis more tractable. Figure 11 illustrates two instances where scenario-based decomposition is applied to a state register based on two test scenarios.

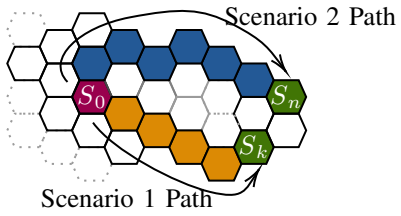


Fig. 11: Reduction of state space by execution path prioritization with respect to the scenarios outlined in firmware.

Algorithm 4 illustrates the major steps involved in the path prioritization process. First, we reconstruct the execution tree based on the test cases that the model was simulated on. Then traces that have different paths due to the test case difference are identified. For each case, we construct the hints such that path conditions for states that are not in the path get weakened. Path conditions are derived from the FSM that we extracted from the design during static analysis.

Algorithm 4 pathPrioritization(S, S_r)

Input: Traces S, State Registers S_r,
Output: Path Conditions p

```
for each v in S do
    if v ∈ S_r then
        v<exec>.append(v.value)      ▷ Generate Execution Tree
    end if
end for
for each r ∈ S_r do
    for each state ∈ r do
        if state ∉ v<exec> then
            conditions ← getConditions(State)
            p.append(conditions)
        end if
    end for
end for
Return p
```

Example 9 (Path Prioritization) : *Based on Algorithm 4, HIVE is able to generate hints to guide the state space through relevant conditions of the example SoC. Figure 12 illustrates the FSM after applying the weakening conditions. For Scenario 1, path conditions related to the receiver (States H, I, and G) are weakened. For Scenario 2, path conditions related to the transmitter (States B, C, D, E, and F) are weakened. Similarly, in Scenario 3 and 4 irrelevant state conditions are weakened by the hints. In this case, sample concretization hints that are related to the scenario 1 can be illustrated as follows. Let's assume that there is a variable X that gets its values assigned within the states that are related to UART receiver (States H,*

I, and G). Since scenario 1 is not interested in states H, I, and G , the variable X will be concretized to the value that was observed during the simulation. In this case, the SMT solver will process the variable X as a concrete value instead of a symbolic value. ■

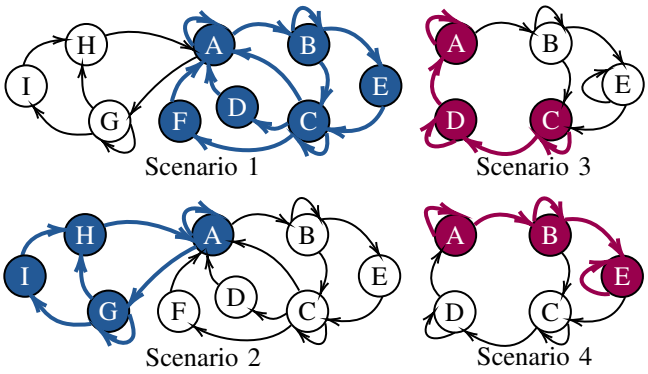


Fig. 12: Path prioritization for four scenarios outlined in Example 2 (colored paths get priority).

F. Scenario-based Decomposition

Once hint generation is complete, *HIVE* starts preparing for the hybrid symbolic execution. For this, natural boundaries of the designs such as module instantiations are used. Moreover, test scenarios are evaluated separately making them sub-problems of the original verification problem. For ease of development and human understanding, system designs are composed of component (module)-level designs. Following the design hierarchy, system models can be decomposed into components. There are promising efforts for proof by decomposition with conjoining specifications and assume-guarantee reasoning [16], [17]. The sub-problem decomposition technique of *HIVE* is based on assume-guarantee reasoning.

In assume-guarantee reasoning, each subsystem or component is viewed as a black box, where the assumptions specify the possible behaviors of the environment or other components that interact with the subsystem, and the guarantees specify the expected behaviors of the subsystem under all possible assumptions. The technique then uses logical reasoning to prove that the guarantees of each subsystem are preserved in the presence of the assumptions of other subsystems or the environment. In the paradigm of assume-guarantee, we define assumption (A) about the system, sub-component (C), and the property (P) that should hold within the system. The formula for the assume-guarantee relationship can be formulated as $f = \langle A \rangle\, C \langle P \rangle$. Here f evaluates to true, whenever C is a part of a system that satisfies A. In this case, the system must also guarantee P.

Consider a system S that is composed of two sub-components of C_1 and C_2 ($S = C_1 || C_2$). In order to check whether the S satisfies P, by analyzing C_1 and C_2 assume-guarantee reasoning principle in inference rule can be applied as follows.

$$\frac{\begin{array}{ll}\text{(Premise 1)} & \langle A \rangle\, C_1 \langle P \rangle \\ \text{(Premise 2)} & \langle true \rangle\, C_2\, \langle A \rangle\end{array}}{\langle true \rangle\, C_1 \,||\, C_2\, \langle P \rangle}$$

This indicates that if $\langle A \rangle\, C_1 \langle P \rangle$ and $\langle true \rangle\, C_2 \langle A \rangle$ holds, then $\langle true \rangle\, S\, \langle P \rangle$ hold true. In order for the above inference rule to be valid, A must be more abstract than C_2, and still it should describe C_2's behavior correctly while A should be strong enough for C_1 to satisfy P.

HIVE make assumptions on the system model, validates them on the system model, and provides validated hints to the sub-component wise proofs. Therefore, *HIVE* framework natively satisfies the definition of assume-guarantee. Equivalence checking proofs can be performed on the decomposed system by selecting one component at a time with the hints generated by the *HIVE*. Note that in situations that have the same set of hints in multiple cases, *HIVE* will concatenate them to one sub-problem, and hence repeated analysis is avoided. Figure 13 illustrates a situation where *HIVE* simplifies the formal model of the hardware module m_j for *scenario i* using the hints extracted from the system. Here, the dynamic analysis is performed with the simulation of *scenario i*. *HIVE* effectively removes unrelated components from the formal model using system-level hints, as discussed in Section III-E. The simplified model m'_j is used in the verification of *Scenario i*, as discussed in the next section.

Example 10 (Sub-Problems): *HIVE is able to decompose the verification problem by module boundaries and verification scenarios. For the SoC in Example 1, we have provided four test cases. Based on both factors, HIVE decomposes the entire system model into the following sub-problems.*

$$\begin{aligned} SoC_{scenario1} &= cpu_1 \,||\, rom_1 \,||\, ram_1 \,||\, uart_1 \,||\, tlc_1 \\ SoC_{scenario2} &= cpu_2 \,||\, rom_2 \,||\, ram_2 \,||\, uart_2 \,||\, tlc_2 \\ SoC_{scenario3} &= cpu_3 \,||\, rom_3 \,||\, ram_3 \,||\, uart_3 \,||\, tlc_3 \\ SoC_{scenario4} &= cpu_4 \,||\, rom_4 \,||\, ram_4 \,||\, uart_4 \,||\, tlc_4 \end{aligned}$$

■

G. Sub-Problem Proofs using Hints

So far we have discussed the process of automatically generating hints. In this section, we describe the steps to convert the implementation into a formal model and how the generated hints are utilized to address the path explosion. For this purpose, we convert a copy of the system model with inline firmware application (without test scenarios) to SMT format, which serves as our formal model for equivalence checking. Then, the proposed framework will construct the state machine of the implementation against the state machine of the specification. Effective utilization of hints can simplify extremely complicated proof scenarios to tractable problems for the underlying SAT solvers. In this section, we discuss three types of limitations on scalable proof with symbolic execution. Then we discuss how hints are used in HIVE to mitigate these limitations. There are three major factors that affect the scalability of formal proofs: path explosion, state space explosion, and exhaustive components in the design.

Path Explosion: Path explosion occurs in formal methods, such as model checking and symbolic execution, where the number of paths in a system's state space can grow exponentially with the size of the system. In other words, it becomes computationally infeasible to exhaustively check all possible paths for correctness or to explore the entire state

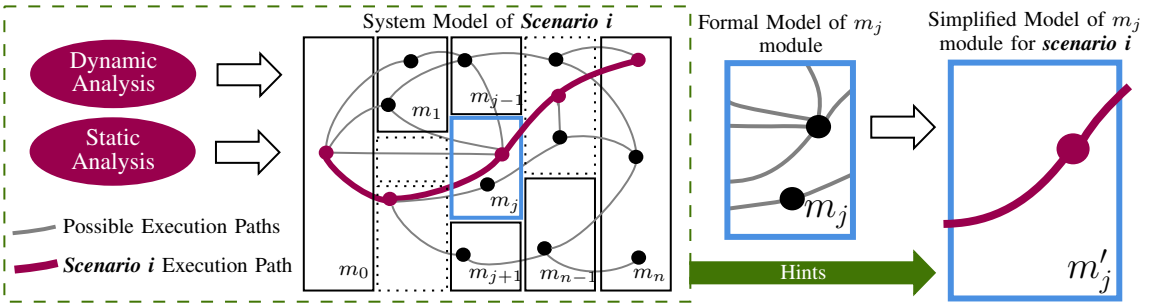


Fig. 13: Illustration of how *HIVE* simplify module level proofs utilizing system level hints. Here $m_0 - m_n$ represents different modules in the hardware implementation. In this instance, the system level hints simplify the formal model of the module m_j for the ***scenario i*** to m'_j using constraints. Simplified m'_j can be directly used in formal proof of ***scenario i***.

space. As a result, the verification process may introduce unacceptable time and memory overhead, and in reality, may unsuccessfully terminate after exceeding memory capacity of the computer. *HIVE* addresses path explosion by performing path prioritization relevant to the test scenarios outlined in Algorithm 4.

Symbolic Term Explosion: Another problem faced by formal proofs is that verification techniques that rely on symbolic execution can lead to a symbolic term explosion problem, which occurs when the number of possible program states (combinations of symbolic and concrete values) grow too large to be handled by the analysis tool. This can occur when the symbolic execution engine explores all possible states through the program, which can be exponential in the size of the input space. *HIVE* implements constraints on the symbolic variables in the form of concretization, weaken, overapproximation, and abstraction as outlined in Section III-E1. This process has the ability to control the term space from symbolic to concrete while making some variables opaque so that their values can be substituted later in the process.

Exhaustive Components: There are specific components in the design where a large run time can be anticipated during the proofs. Components such as memories and counters can contribute to the exponential state space expansion. This problem is also addressed by the path prioritization technique since all the unused areas of memory components will get lower priority by effectively abstracting such implementations to inputs and outputs with only relevant logic.

Note that *HIVE* generates the hint only for the scenario under test. Therefore, assumptions and properties are only valid for the specific scenario under consideration.

Listing 3: Sample set of overapproximate hints generated by the *HIVE* and import procedure to utilize hints in the proofs.

```
(define (overapproximate-case1 module)
    (match module
        [`uart (for ([field (list
                    "recv_buf_valid"
                    "recv_buf_data"
                    "recv_divcnt"
                    "recv_pattern"
                    ... )])
                (match-abstract! field field))]
        [`cpu (for ([field (list
                    ... )])
                (match-abstract! field field))]
        ....
        [default (displayln "Module not found!")]))
; Usage of hints in the proofs
(require "symcrete/dut/hints.rkt") ; import hints
(hint overapproximate-case1 `uart) ; use for UART proof
```

Example 11 (Hints in *Racket* Syntax) : *For four testing scenarios in Example 2, HIVE categorized signals into groups to be considered as hints in each scenario. Then these signals are adopted into pre-built Racket programming language templates. A sample set of overapproximate hints after verifying them on the system model and adopting them to Racket format are illustrated in Listing 3. Hints related to each case are constructed separately and generated hints can be directly imported into the proofs. The example code defines a function* `overapproximate-case1` *to generate hints for proofs related to scenario 1 of the system model. It uses pattern matching to handle different sub-problems related to each of the module (e.g., uart and cpu), iterating over their respective fields and employing the* `match-abstract!` *function.* ■

IV. Experiments

This section presents four case studies using hardware and firmware implementations to demonstrate the effectiveness of *HIVE*. First, we introduce the setup we have used to perform the experiments. Next, we discuss each case study and present results on hardware-firmware co-verification.

A. Experimental Setup

We consider four System-on-Chip (SoC) implementations. These implementations utilize *PicoRV32* processor [32] which is a CPU based on RISC-V instruction-set architecture configured with a ROM and RAM.

To perform symbolic execution and equivalence checking, we have used *Rosette* [33] language with *Z3* [34] as the back-end solver. Rosette is a solver-aided programming language extension for the *Racket* functional programming language. In order to compile the *C* and *assembly* programs into binary, *riscv-gnu-toolchain* [35] was used. To convert Verilog RTL implementations into *smt2* (*Satisfiability Modulo Theories Library v2*) format, *Yosys* [36] open synthesis tool was used. *Rosette* package *rtlv* [8] was used for lifting the smt2 output to racket supporting syntax and *rtlv/shiva* was used to feed hints to the equivalence checking problems. Specifications related to each of the test scenarios are manually written in the *Racket* language. Extraction of abstract syntax tree is performed by *Icarus Verilog Loadable Target API* [37]. Module level FSM extraction and data flow (**.blif*) are extracted through *Yosys*. All the simulations for obtaining Value Change Dumps (VCD) were performed with Icarus Verilog [38]. All the scripts related to hint generation, path prioritization, and symbolic state

reduction were implemented in *Python*. All the experiments were performed on a *ARMv8.5* based *Apple M1* system with 16GB RAM inside a *Docker* environment.

For the evaluation of performance, we have selected four benchmark circuits that contain considerably large FSM implementations. Table I presents the size and trace length-related details about the benchmark SoCs. After the validation process, each of the SoC was compiled with *Yosys* and *nextpnr* [39] to generate the necessary binary file to program the *Lattice ICE-Sugar-Nano* open-source Field Programmable Gate Array (FPGA). Finally, the usability of validated real-world SoCs were evaluated by running the firmware on the programmed FPGA.

TABLE I: Size comparison of RISC-V based SoC implementations considered for the four case studies.

Case Study SoC	**SEB**	**ECC**	**LCD**	**SATA**
Lines of Code (HW+FW)	4371	4999	5022	5628
Number of Signals (HW)	1471	1563	1494	1645
Average Trace Size (MB)	695	1874	1084	2027
Simulated Clock Cycles	50,000	100,000	100,000	50, 000

B. Case Study 1: Simple Encrypted Backup (SEB)

Simple Encrypted Backup (SEB) is a data logger SoC that reads data with a secret numeric pin from a host device and stores the result after performing a simple transposition cipher on the data on an external memory card. It uses the UART interface to communicate with the host device and uses an external memory card reader that communicates through the Serial Peripheral Interface (SPI) to store data as illustrated in Figure 14. The external device can request to read the data from the storage by providing the secret pin back. External devices access the SoC via the terminal environment as illustrated in Figure 15.

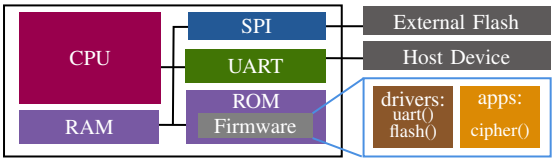


Fig. 14: Simple Encrypted Backup SoC.

Both UART and SPI are configured on the SoC such that they both have separate MMIO regions to communicate with the CPU. The transposition cipher logic is implemented within the firmware. In order to evaluate the correctness of the encrypted backup SoC, we have developed a test plan with several test scenarios. This includes verifying reading from the host machine, writing to the host machine, reading from the external memory card, writing to the external memory card, encrypting the data, and decrypting the data. First, we attempted to perform equivalence checking on the system for outlined properties without *HIVE*, and Z3 solver timeouts were observed for all scenarios. Therefore, we applied the proposed approach and validated the implementation.

Table II presents the results related to the verification process of the SEB SoC. The verification results illustrate that *HIVE* is able to decompose the SoC verification problem into tractable sub-problems for the underlying solver within an acceptable time and memory.

```
> $ screen /dev/tty.usbmodem102 115200
Welcome to Simple Encrypted Backup!
Select options
    1. Encrypt and Write Data
    2. Decrypt and Read Data
Your Option? _
```

Fig. 15: Simple Encrypted Backup terminal interface on a host machine, SoC is executed on the *Ice-Sugar-Nano* FPGA.

TABLE II: Time (in Minutes) and memory (in Megabytes) consumed by the validation scenarios of the SEB SoC. *We did not show comparison with state-of-the-art since existing methods without hints face timeouts for all scenarios.*

Property	**Test Case**	**Hint Generation**		**Equivalence Checking**	
		Time	**Mem**	**Time**	**Mem**
Read	*from SPI Memory*	13	674	19	639
	from Host	17	756	25	703
Write	*to SPI Memory*	15	698	20	655
	to host	18	789	24	697
Text	*Encrypt*	21	701	30	985
	Decrypt	21	701	32	946

C. Case Study 2: ECC Core Accelerator

Elliptic Curve Cryptography (ECC) functions are popularly used in public key cryptosystems that can be utilized for both authentication and authenticated encryption. ECC core accelerator provides functionality to perform calculations related to the Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve Integrated Encryption Scheme (ECIES) faster. Implementation of ECC core consists of *ECC_Arithmetic core* and *SHA256*. Users can use the ECC core as an IP in the SoC design as illustrated in Figure 16. Through the firmware driver, each of the functionalities can be accessed by the application. ECC core is connected with the SoC via the MMIO, and based on the request register status, it will perform the relevant calculation and the results will be written back to the results register. ECC core implementation consists of several nested FSMs in order to achieve the above functionalities. In order to evaluate the effectiveness of *HIVE*, we have selected several functional scenarios from the *ECC_Arithmetic core*. These functional scenarios include testing the field addition, field multiplication, field inverse, and fast reduction. Table III presents the elapsed time and consumed memory for hint generation as well as verification with *HIVE*.

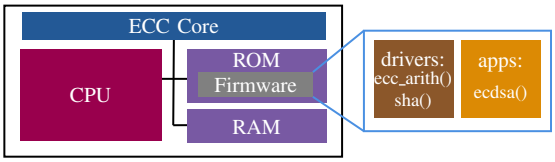


Fig. 16: ECC Core Accelerator SoC.

D. Case Study 3: LCD Controller

This SoC implementation consists of a hardware Liquid Crystal Display (LCD) controller as illustrated in Figure 17. It can be configured by the firmware and then display data can be sent to the display. Configure parameters are sent as command packets from the firmware and data packets are

TABLE III: Time (in Minutes) and memory (in Megabytes) consumed by the verification process of the SoC with the ECC Core for different verification scenarios of *Arithmetic Core*. *We did not show comparison with state-of-the-art since existing methods without hints face timeouts for all scenarios.*

Test Cases for Arithmetic Core	Hint Generation		Equival. Checking	
	Time	Memory	Time	Memory
Field Addition	33	2109	207	1876
Field Multiplication	47	2967	239	1908
Field Inverse	41	2479	211	1856
Fast Reduction	28	1795	242	2064

sent as data packets to the controller. This implementation is designed to work with ST77XX family display drivers. The controller consists of an FSM with 33 states and 30 transitions. ST77XX family device drivers have particular delay periods between different command parameters and data packets.

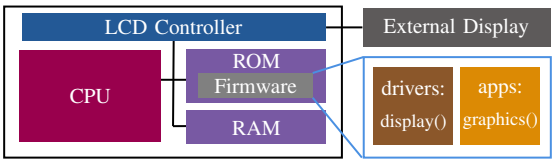


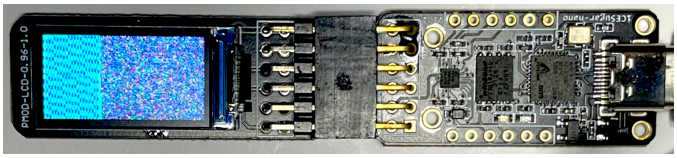
Fig. 17: LCD Controller SoC.

According to the specification, implementation requires satisfying several basic conditions to properly initialize the external display and to show graphics on the display. These requirements can be outlined as follows, 1) display delay values are within the specification range, 2) correct commands are sent to the external display from the firmware, and 3) data packets are in sync with the coordinates of the display. Based on the requirements, we formulated several test scenarios for this experiment. For each of the test scenarios, we have written one test case in the firmware. Then we used the original specification [40] of the display driver to write the specification in *Racket* language.

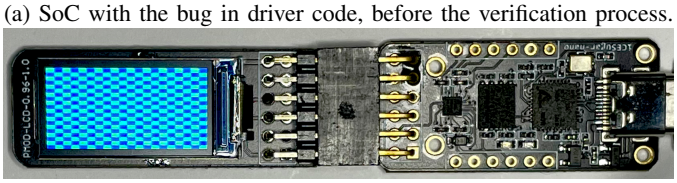
TABLE IV: Time (in Minutes) and memory (in Megabytes) consumed by the verification process of the SoC with the SPI LCD Controller for different verification scenarios. *We did not show comparison with state-of-the-art since existing methods without hints face timeouts for all scenarios.*

Property	Test Cases	Hint Generation		Equivalence Checking	
		Time	Mem	Time	Mem
Delay Check	*SWRESET*	23	1649	37	1489
	SLPOUT	29	1720	33	1348
	NORON	27	1644	32	1394
	DISPON	27	1679	42	1489
Command Mode	*CASET*	19	976	23	1296
	RASET	19	981	22	1301
	COLMOD	19	967	21	1290
	MADCTL	20	970	26	1310
Data Mode	*NEW_FRAME*	17	979	31	1358
	WAIT_FRAME	19	963	33	1380
	INIT_FRAME	18	958	33	1367

After applying the *HIVE* on the test scenarios, we performed equivalence checking to provide a guarantee for all the scenarios. Table IV illustrates the verification scenarios for the LCD controller SoC and elapsed time and memory used for each of the verification sub-problem with *HIVE* generated hints. We were able to reveal a functional bug related to *CASET* and *RASET* test cases that were causing the external ST7735 driver to set up X and Y coordinate values incorrectly, this was fixed in the firmware device driver. Figure 18 illustrates the design running on the FPGA with *checkered_Flag.c* benchmark. Figure 18a illustrates the SoC with the driver bug, and here the external display displays graphics incorrectly. Figure 18b presents the validated SoC after fixing the driver bug, displaying data on the external display properly.



(a) SoC with the bug in driver code, before the verification process.



(b) SoC after the verification process and fixing the driver bug.

Fig. 18: SoC with SPI LCD controller running *checkered_Flag.c* benchmark. Firmware is sending the display content to the external display with the ST7735 driver on *Lattice Ice-Sugar-Nano* FPGA.

E. Case Study 4: Mass Storage (SATA) Controller

Serial Advanced Technology Attachment (SATA) controller is a hardware module that manages the communication between a SoC and mass storage devices (such as hard disk drives and solid-state drives). In this case study, we have utilized a SATA controller that connects to the SoC using the AXI (Advanced eXtensible Interface) as illustrated in Figure 19. The controller is configured and managed by the firmware driver, which has the ability to change the state of the data transfer between the mass storage device attached to the controller based on the requirement of the application code. For simulation purposes, we have used the Verilog `readmemb` function as the mass storage device.

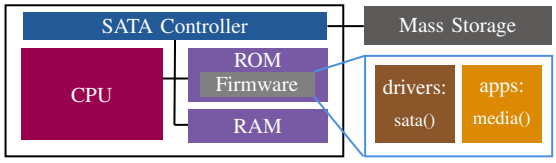


Fig. 19: SoC that integrates SATA controller to communicate with the mass storage device.

The firmware driver is responsible for managing the SATA controller to perform handshake, establishing and maintaining the communication protocol between the system and the storage device, thereby ensuring reliable and synchronized data exchange. SATA controller implementation consists of several nested FSMs in order to achieve the above functionalities. In order to evaluate the effectiveness of *HIVE*, we have selected several functional scenarios from the implementation. These

TABLE V: Time (in Minutes) and memory (in Megabytes) consumed by the verification process of the SoC with the SATA device connected via SATA controller for different verification scenarios of *SATA controller*. *We did not show comparison with state-of-the-art since existing methods without hints face timeouts for all scenarios.*

Test Cases for SATA Driver	Hint Generation		Equival. Checking	
	Time	Memory	Time	Memory
Initialization Sequence	39	3109	274	2850
Handshake Phase	41	3190	269	2689
Command Phase	43	3019	255	2503
RX Transaction	34	3100	270	2476
TX Transaction	34	3100	270	2476

functional scenarios include testing the controller initialization, SATA handshake, controller commands, and read and write transactions. Table V presents the elapsed time and consumed memory for hint generation as well as verification with *HIVE*.

V. Applicability and Limitations

In this paper, we focused on automated generation of hints to simplify equivalence checking during functional verification by reducing the state space. In this section, we discuss the applicability and limitations of *HIVE*.

Usability and Tools: *HIVE* usability is not limited to functional verification. This framework can be used in other scenarios (e.g., security validation) to reduce the state space and make the problem more tractable. Note that Rosette was used in the experiments due to seamless integration with the backend SMT (Z3) solver for symbolic execution to aid formal proofs. However, other formal tools also can be used with minor modifications to the framework.

Simulation versus Formal Guarantee: Simulation does not provide any verification guarantee since it is infeasible to simulate using all possible input sequences. The contribution of *HIVE* is to translate one concrete test case to a symbolic test case and perform equivalence checking to obtain formal guarantee. For example, consider the addition functionality (scenario) of a 64-bit calculator. In order to verify addition, *HIVE* only needs one test case, such as $2+3=5$. By looking at the simulation trace, Algorithm 3 figures out which paths are relevant for the hint generation, which then facilitates symbolic simulation of the addition scenario. During the symbolic simulation, symbolic values will propagate through the design covering all possible inputs (instead of concrete values like $2+5$). This provides guarantees for addition functionality for all the inputs of $2^{64} \times 2^{64}$. If our framework is not used, a designer has to simulate the addition scenario of the calculator $2^{64} \times 2^{64}$ times to get the same level of verification guarantee.

Effort for Implementation Abstraction: The manual specification requirement remains the same for both *HIVE* and existing (state-of-the-art) approaches. The fundamental difference is that existing approach requires manual abstraction of the implementation. For example, "C=A+B" would be the specification of an adder. The abstraction of the adder implementation will look different depending on whether it was implemented as ripple-carry adder or carry look-ahead adder. In contrast, our framework eliminates the need for manually writing a per-scenario abstraction of the implementation. In other words, the automatically generated hints slices the implementation to its smallest possible size during symbolic simulation, giving the same effect as manually abstracted implementation.

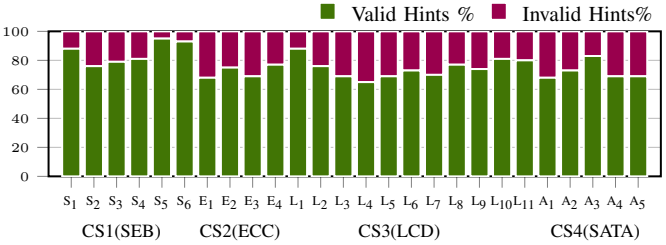


Fig. 20: Percentage of valid and invalid hints generated by *HIVE* framework for different scenarios of four case studies. Here S_1 to S_6, E_1 to E_4, L_1 to L_{11} and A_1 to A_5 denotes scenarios corresponding case studies CS1, CS2, CS3, and CS4.

Invalid Hints: Since *HIVE* is focused on automating the manual hint generation procedure, it also has some of the inherent drawbacks of the manual method, such as the generation of invalid hints. However, the percentage of invalid hints is typically small. Note that if hints are not available at all, SMT will face state space explosion. Figure 20 shows the percentage of valid and invalid hints generated by HIVE for each of the validation scenarios. It can be observed that the scenarios evaluated in the four case studies (CS1:S1-S6, CS2:E1-E4, CS3:L1-L11, and CS4:A1-A5), the vast majority of the hints generated by *HIVE* are valid hints. Note that the proof is still complete since there will be no false positives, and the false negatives can be eliminated by utilizing the counterexamples.

False-Negative Results: *HIVE* does not produce false-positive results since the hardware implementation is directly converted to an SMT model and it has the exact SMT model of the implementation. Both *HIVE* and state-of-the-art manual abstraction methods may produce false negatives if the specification is incorrect or incomplete. The false-negative results will not directly affect the verification guarantees of the implementation since the symbolic evaluation results will provide a counterexample. A designer can analyze the counterexample to find it as a false negative and can modify the specification to fix the problem.

VI. Conclusion

Hardware-firmware co-verification is an important requirement in designing reliable systems. While there are promising formal verification approaches, they rely on manually written hints or abstracted versions of the hardware to deal with the state space explosion. Manual intervention requires design expertise and can be time-consuming and error-prone. In this paper, we presented an automated hint-based verification (*HIVE*) framework that does not require any manual intervention during model generation or hint extraction. We effectively utilize both static analysis and concrete simulation of the implementation to extract proof supporting artifacts from the system model. *HIVE* can automatically generate system-level hints to guide formal proofs to avoid state space

explosion. Extensive evaluation using real-world hardware-firmware implementations demonstrates the effectiveness and scalability of the proposed framework. In all the case studies, *HIVE* was able to provide a guarantee for the scenarios in a reasonable time, while state-of-the-art methods failed to verify them (timeout). Moreover, *HIVE* is able to identify several faulty scenarios in the hardware-firmware implementations.

REFERENCES

[1] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *CODES+ISSS*, 2014, pp. 1–10.

[2] S. Ahn and S. Malik "Modeling firmware as service functions and its application to test generation," in *Haifa Verification Conference*. Springer, 2013, pp. 61–77.

[3] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[4] A. Jayasena, B. Kumar, S. Charles, H. Witharana, and P. Mishra, "Network-on-chip trust validation using security assertions," *Journal of Hardware and Systems Security*, vol. 6, no. 3-4, pp. 79–94, 2022.

[5] Z. Gao, W. Dong, R. Chang, and Y. Wang, "Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 16, p. e5756, 2022.

[6] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Notary: A device for secure transaction approval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 97–113.

[7] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying hardware security modules with {Information-Preserving} refinement," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 503–519.

[8] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "rtlv: push-button verification of software on hardware."

[9] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ila) a uniform specification for system-on-chip (soc) verification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.

[10] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in socs using instruction-level abstraction for hardware," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[11] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik, "Ilang: a modeling and verification platform for socs using instruction-level abstractions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 351–357.

[12] A. Jayasena and P. Mishra, "Directed test generation for hardware validation: A survey," *ACM Computing Surveys*, vol. 56, no. 5, pp. 1–36, 2024.

[13] M. Dobrescu and K. Argyraki, "Software dataplane verification," *Communications of the ACM*, vol. 58, no. 11, pp. 113–121, 2015.

[14] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 314–327.

[15] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[16] M. Abadi and L. Lamport, "Conjoining specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 3, pp. 507–535, 1995.

[17] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh, "Assume-guarantee verification of source code with design-level assumptions," in *Proceedings. 26th International Conference on Software Engineering*. IEEE, 2004, pp. 211–220.

[18] M. Abadi and L. Lamport, "Composing specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 1, pp. 73–132, 1993.

[19] S. Blom and J. van de Pol, "State space reduction by proving confluence," in *Computer Aided Verification (CAV)*. Springer, 2002, pp. 596–609.

[20] K. Yorav and O. Grumberg, "Static analysis for state-space reductions preserving temporal logics," *Formal Methods in System Design*, vol. 25, pp. 67–96, 2004.

[21] S. Vasudevan, E. A. Emerson, and J. A. Abraham, "Efficient model checking of hardware using conditioned slicing," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 279–294, 2005.

[22] S. Vasudevan, A. Emerson, and J. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *International Journal on Software Tools for Technology Transfer*, vol. 9, pp. 89–101, 2007.

[23] R. Stuber, "Formal verification: Not just for control paths," *Verification Horizons*, vol. 13, no. 2, pp. 62–66, 2017.

[24] P. Herber, "The rescue approach-towards compositional hardware/software co-verification," in *IEEE HPCC*. IEEE, 2014, pp. 721–724.

[25] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for soc verification," in *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2015, pp. 160–167.

[26] H. Zhang, C. Trippel, Y. A. Manerkar, A. Gupta, M. Martonosi, and S. Malik, "Ila-mcm: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–10.

[27] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[28] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2019, pp. 1–8.

[29] Y. Lyu and P. Mishra, "Scalable concolic testing of rtl models," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2020.

[30] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *DATE*. IEEE, 2018, pp. 1538–1543.

[31] Y. Lyu, A. Ahmed, and P. Mishra, "Automated activation of multiple targets in rtl models using concolic testing," in *DATE*. IEEE, 2019, pp. 354–359.

[32] PicoRV32, "A size optimized risc-v cpu," https://github.com/YosysHQ/picorv32, 2023, accessed: 2023-4-10.

[33] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *ACM SIGPLAN*, 2013, pp. 135–152.

[34] Z. solver, "A theorem prover from microsoft research," 2023. [Online]. Available: https://github.com/Z3Prover/z3

[35] GCC:RISC-V, "Risc-v gnu compiler toolchain," 2023. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain/

[36] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/, 2023.

[37] IVerilogAPI, "Icarus verilog loadable target api," 2023. [Online]. Available: https://steveicarus.github.io/iverilog/usage/ivl_target.html

[38] "Iverilog," 2023. [Online]. Available: http://iverilog.icarus.com/

[39] Nextpnr, "a portable fpga place and route tool," https://github.com/YosysHQ/nextpnr, 2023, accessed: 2023-4-10.

[40] st77xx Datasheet, "St77xx specifications," 2023. [Online]. Available: https://www.displayfuture.com/Display/datasheet/controller/ST7735.pdf

Aruna Jayasena is a Ph.D student in the Department of Computer & Information Science & Engineering at the University of Florida. He received his B.S. in the Department of Computer Science and Engineering at the University of Moratuwa, Sri Lanka, in 2019. His research focuses on systems security, hardware-firmware co-validation, test generation, trusted execution, side-channel analysis, and system-on-chip debug.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received his Ph.D. in Computer Science from the University of California at Irvine. His research interests include embedded systems, hardware security, formal verification, system-on-chip validation, machine learning, and quantum computing. He currently serves as an Associate Editor of ACM Transactions on Embedded Computing Systems. He is an IEEE Fellow, an AAAS Fellow, and an ACM Distinguished Scientist.