RTSEC: Automated RTL Code Augmentation for Hardware Security Enhancement

Orlando Arias, *University of Florida*
Zhaoxiang Liu, *Kansas State University*
Xiaolong Guo, *Kansas State University*
Yier Jin, *University of Florida*
Shuo Wang, *University of Florida*

***Abstract*—Current hardware designs have increased in complexity, resulting in a reduced ability to perform security checks on them. Further, the addition of any security features to these designs is still largely manual which further complicates the design and integration process. In this paper, we address these shortcomings by introducing RTSEC as a framework which is capable of performing security analysis on designs as well as integrating security features directly into the HDL code, a feature that commercial EDA tools do not provide. RTSEC first breaks down HDL code into an Abstract Syntax Tree which is then used to infer the logic of the design. We demonstrate how RTSEC can be utilized to automatically include security mechanisms in RTL designs: watermarking and logic locking. We also compare the efficacy of our analysis algorithms with state of the art tools, demonstrating that RTSEC has capabilities equal or superior to those of state of the art tools while also providing the means of enhancing security features to the design.**

I. INTRODUCTION

As the complexity of hardware designs increase, automatic design checks become necessary. As such, EDA vendors have introduced a series of tools to aid hardware designers in their checks [1], [2]. However, these tools are unable to automate the insertion of state of the art security features into designs. This is left to the designer to perform manually.

For example, a designer may wish to watermark a finite state machine (FSM) within a larger HDL IP core in order to identify it. For this, the designer must find the FSM in the IP core, then detect and modify both the transition and output functions while ensuring that no syntax or logic errors are introduced. The designer must ensure that the watermark function introduced does not generate any conflicts with existing logic. Any auxiliary inputs and outputs to the watermarked FSM need to be manually propagated through the module hierarchy, while resolving signal conflicts if multiple instances of the watermarked module are used. This process becomes excruciating for large designs.

Another example is the obfuscation of logic within the HDL design. The designer is tasked with introducing new unlock signals and combinational logic around signals of interest, changing their value lest the proper key is furnished. Performing this task manually involves not only the introduction of these signals, and logic, but also the propagation of the signals into the toplevel module resolving any conflicts in the module dependency tree of the design. This proves to be cumbersome and error-prone for large designs.

Despite that hardware security features such as watermarking and logic locking have become critical features for circuit designs, commercial EDA tools still focus on performance optimization but leave the security feature insertions and verification to the users, a gap leaving many of the hardware security research outcomes not implemented in commercial designs. To address these issues we introduce RTSEC as a framework which can perform security analysis and integrate state of the art security features directly in HDL code. RTSEC first breaks down the provided HDL into an abstract syntax tree (AST) and performs a series of inference checks in order to extract features of interest in the design. RTSEC can then add user-specified security enhancements to the design. It then regenerates new HDL code for the design with the enhancements included *without losing the original HDL code's semantics*. While RTSEC can be used for different hardware security feature insertion, in this paper, we mainly demonstrate how RTSEC is able to fully recover finite state machines (FSMs) in the design, including both the transition and output maps. We then show how we can use the information to watermark FSMs in the design. Moreover, we show how we can use our framework to automatically perform logic locking in HDL designs. We further compare RTSEC's inference capabilities to that of existing tools, showing that our inference algorithm have capabilities equal or superior to existing tools (including those commercial tools), while also providing the means to automatically insert security features into the design.

II. METHODOLOGY FOR HDL ANALYSIS AND INSTRUMENTATION

A. RTL Analysis Basics

The basis for our RTL analysis tools is the parsing of the HDL source files into an Abstract Syntax Tree (AST). An AST is a deterministic data structure with no loops which contains a representation of the source file in terms of nodes. Each node in the graph is representative of a token or series of tokens in the source file. For proper parsing of source files, a lexer and a parser are needed. The lexer converts the source file into a stream of tokens. This stream serves as the input to the parser which employs the rules of the grammar of the language in order to generate an abstract representation of the source code. Once the AST is generated, we are able to traverse the graph using traditional graph navigation routines. For example, to locate a particular node type, such as those containing declarations, we can perform a breadth-first search over the AST. This particular type of search is preferred since declarations are often close to the root node of the AST.

978-3-9819263-6-1/DATE22/©2022 EDAA

Using the AST we can further infer the scope for certain operations, such as assignments. This is helpful when dealing with tasks such as data-flow analysis. For this purpose, we can search for locations where assignments to a certain l-value of interest such as a signal take place. When performing the search we also record if we have traversed any conditional statements. This allows us to derive the commonly named φ-nodes when representing assignments to variables/signals.

B. Flip-Flop Inference Rules

When representing flip-flops in Verilog and SystemVerilog HDL, these are normally written as part of a procedural block that is edge triggered on a clock. Flip-flops may have a synchronous or asynchronous reset signals. In the latter case, the reset signal is also added as an edge trigger to the procedural block. The reset signal is checked for a level within the procedural block. Within the procedural block, whenever the signal reset signal is asserted, a zero constant is assigned to a signal of type `reg` which serves as the storage element. Enable signals, if used, are not added to the trigger list of the procedural block. The signal is synchronously checked within the procedural block for its active logic level. If asserted, a non-constant assignment is made into storage element.

Algorithm 1 Flip-Flop inference algorithm. We iterate over the AST finding procedural blocks that are edge triggered. We examine all candidate blocks and check assignment.

```
1:  procedure FF_INFERENCE(ast)
2:      ℙ ← procedural_blocks(ast)
3:      𝔽 ← ∅
4:      for p ∈ ℙ do
5:          if edge_triggered(p) then
6:              𝔽 ← 𝔽 ∪ {p}
7:          end if
8:      end for
9:      𝔸 ← get_assigns(𝔽)
10:     𝔽 ← ∅
11:     for a ∈ group_targets(𝔸) do
12:         (res, set, en, q, d) ← infer_data_flow(a)
13:         𝔽 ← 𝔽 ∪ {(res, set, en, q, d)}
14:     end for
15:     return 𝔽
16: end procedure
```

We summarize the inference procedure in Algorithm 1. After obtaining the AST of the module, we look for procedural blocks that are edge triggered. We extract all assignments that occur within the procedural blocks. We group all assignments based on their target signal. We check the scope of the assignment. If a signal level is necessary to store a *constant* value into the target that signal is added as a reset if storing the value 0 or set if storing a non-zero value. If a signal level is necessary to store a *non-constant* value, that signal is treated as the enable signal with the r-value as the data input terminal. The l-value on the store is the output of the flip-flop.

C. Finite State Machine Inference Rules

A finite state machine (FSM) can be defined as the 6-tuple $(\mathbb{S}, s_0, \mathbb{\Sigma}, \mathbb{\Lambda}, T, G)$, where $\mathbb{S}$ is a finite set of states with $s_0 \in \mathbb{S}$ being the initial state. The set $\mathbb{\Sigma}$ corresponds to the input alphabet, that is, the symbols or values that serve as inputs to the FSM. $\mathbb{\Lambda}$ is the output alphabet of the FSM, the symbols or values that are returned by the FSM. T defines the transition function of the FSM, with $T : \mathbb{S} \times \mathbb{\Sigma} \rightarrow \mathbb{S}$. Lastly, G is the output function of the FSM with $G : \mathbb{S} \rightarrow \mathbb{\Lambda}$ for a Moore FSM, and $T : \mathbb{S} \times \mathbb{\Sigma} \rightarrow \mathbb{\Lambda}$ for a Mealy FSM.

1) Transition Functions in FSMs: The transition function T of an FSM maps the Cartesian product of the set of inputs and set of states to the set of states. That is, the expression $(s_i, \sigma_j) \rightarrow s_k$ where $s_i, s_k \in \mathbb{S}$ and $\sigma_j \in \mathbb{\Sigma}$ is interpreted as *when the FSM is in state s_i and receives input σ_j, it will transition to state s_k*. When translated to HDL, this implies checking the current state of the FSM, and the input that is being given to determine transitions to the next state.

When implementing an FSM in hardware, we necessitate a storage for the current state of the FSM. This is normally performed using a register. There are multiple ways transition information can be encoded in HDL, however HDL synthesis tools provide recommendations on how to write FSMs for the purposes of aiding with the optimization of the logic [3], [4], [5], [6]. The recommended procedure is by employing a `case` statement over the state register, then selecting the new state based on the input within a procedural block. Of importance is that we have a data-flow loop with the state register and what holds the next state. Synthesis tools may also accept a direct assignment into the state register in the transition function, in which case a clock-triggered `always_ff` block is used.

2) Output Functions in FSMs: The output function G has one of two forms. A Moore FSM maps the current state directly into an output, whereas a Mealy machine maps the output as a combination of both the current state and input. Translation of these equations into HDL imply checking the current state of the FSM to generate an output in the case of a Moore machine, or checking the current state of the FSM as well as the current input of the FSM in the case of a Mealy machine. Synthesis tools recommend a construct similar to that of a transition function when defining this function.

3) FSM Inference Algorithm: We present an overview of the method used to detect transition function candidates in Algorithm 2. We start by generating the AST of the module to be examined. We then infer all flip-flop candidates using the rules described in Section II-B. The inferred flip-flops are made into state register candidates. Then, for each procedural block we check that it has some form of a select statement. We check the dispatch variable for the select statement, recording those that utilize signals in the inferred flip-flop list. We record the compare values for the variable in the select statement are matched against in the body of the statement.

At this point, we are ready to determine the candidates for the transition function and the output function. For the transition function we check for data-flow in the procedural blocks. If all compare values are assigned to the state register candidate

we record the procedural block as the transition function. In the event where the state register is not directly assigned from within the procedural block, we follow the signal which is the target of the assignment and ensure that its value is eventually stored in the state register candidate. We finish the process by examining the conditions which are required to be met for the compare values to be assigned to the state register. These conditions become the possible input for the state machine. To determine the output function, we look at the remaining procedural blocks with select statements that use the state register as dispatch variable. We check which signals are being driven in the design within the procedural block. We ensure that assignment to these signals are with respect to the state register, and for Mealy machines with respect to the computed input. Assignments to the signals in the procedural block are recorded as the output of the state machine.

Algorithm 2 FSM transition function detection algorithm. The algorithm returns the set $\mathbb{M}$ of transition function candidates. The requirement that all case labels must be assigned to the state register candidate can be loosened to account for unused states/transitions.

procedure FSM_INFERENCE(ast)
  $\mathbb{F} \leftarrow ff\_candidates(ast)$
  $\mathbb{P} \leftarrow procedural\_blocks(ast)$
  $\mathbb{M} \leftarrow \varnothing$
  for $(p, f) \in \mathbb{P} \times \mathbb{F}$ **do**
    $\mathbb{S} \leftarrow select\_statements(p)$
    if $\exists s \in \mathbb{S} \wedge select\_uses(s, f)$ **then**
      $\mathbb{L} \leftarrow select\_labels(s)$
      if $\forall l \in L, f \leftarrow l$ **then**
        $\mathbb{M} \leftarrow \mathbb{M} \cup \{(s, f)\}$
      end if
    end if
  end for
  return $\mathbb{M}$
end procedure

We determine the reset state of the FSM by examining the state register. From our flip-flop inference algorithm, we determine the reset signal candidate as well as the reset value. If the initial state for the FSM is a non-zero value, then the flip-flop inference algorithm returns a set value instead. This is used as the reset state candidate of the FSM. To ensure proper inference, we check this value against the compare values for the variable in the select statement. Matching one of the select values verifies the validity of our candidate.

III. Application Towards Security

A. Towards Watermarking Finite State Machines

We extended our analysis framework to perform the automatic watermarking of finite state machines [7]. In particular, both our ability detecting finite state machines and performing edits to the HDL through the abstract syntax tree of a module gives us the power to *automatically* insert watermarks in a design. Furthermore, for hierarchical designs we are able to propagate the necessary signals driving the watermark of the FSM into an arbitrary module automatically.

Algorithm 3 Watermarking algorithm. Using the ability of RTSEC to detect FSMs and to edit RTL, we are able to automatically generate and add a watermark to our design, as well as to propagate any signals to a specific module.

procedure WATERMARK_FSM(ast)
  $\mathbf{F} \leftarrow fsm\_inference(ast)$
  if $|\mathbf{F}| > 1$ **then**
    $f \leftarrow select\_fsm(\mathbf{F})$
  else
    $f \leftarrow \mathbb{F}_0$
  end if
  $T \leftarrow transition\_fn(f)$
  $G \leftarrow output\_fn(f)$
  $T \leftarrow T + \bigcup_{i,j,k} [(w_i, \sigma_j) \rightarrow w_k]$
  $G \leftarrow G + \bigcup_{i,j,k} [(w_i, \sigma_j) \rightarrow \lambda_k]$
end procedure

We provide an overview of the watermarking process in Algorithm 3. We start by inferring all the FSMs in the design. If more than one FSM is found we select one, otherwise we utilize the only FSM in the design. We obtain its transition function as well as the output function. We utilize the user provided constraints to generate an extension to the transition function which become the watermark states. Then using the initial state information we identify the node in the abstract syntax tree which contains the select statement. This select statement drives the initial set of transitions. We modify this node to add a new transition to the watermarking state. Extending the output function uses a slightly different process. If the output function is that of a Moore FSM, we add a new node to the AST of the module body which contains a new procedural block. This procedural block implements the transition function of a Mealy FSM which drives all of the watermark output signals. We ensure that the original Moore procedural block contains a default state for the original set of output signals, otherwise a default statement will be added. If the original output function is that of a Mealy FSM, we extend the select statement in the procedural block for the output function to also include watermarking states.

B. Towards Logic Locking

We further employ our framework to perform logic locking on RTL designs [8]. We implement a traditional logic locking scheme where combinational logic is added to signals internal to the design. Our implementation propagates signals to the top level module exposing them as part of the input.

We describe our methodology for signal obfuscation in Algorithm 4. We start by obtaining a module dependency graph by scanning the top level module for instances. We flag the modules as the children, then proceed to scan the body of those modules for module instances. Any modules found this way are flagged as children of the respective children of the top level module, then scanned for instances. This process is recursively repeated until no more instances are found. We iterate over each

Benchmark	Quartus		Vivado		DC		Yosys		Pyverilog		This Work	
	FSM	Time	FSM	Time	FSM	Time	FSM	Time	FSM	Time	FSM	Time
aes128	●	6 s†	●	7 s†	○	0.13 s	●	0.91 s	○	–‡	●	0.392 s
apb2spi	●	7 s†	●	5 s†	○	0.13 s	●	0.09 s	●	2.5 s	●	0.012 s
crcahb	●	6 s†	●	5 s†	○	0.14 s	●	0.33 s	●	110.5 s	●	0.015 s
ima_adpc_enc	●	6 s†	●	5 s†	◐	0.14 s	●	0.17 s	○	–‡	●	0.032 s
pid_ctrl	●	7 s†	●	5 s†	◐	0.14 s	●	0.42 s	○	–*	●	0.070 s
prep3_binary	●	7 s†	●	5 s†	◐	0.13 s	●	0.80 s	●	2.2 s	●	0.003 s
prep3_onehot	○	6 s†	○	5 s†	○	0.13 s	○	0.08 s	○	–*	●	0.003 s
prep4_binary	●	6 s†	○	5 s†	◐	0.13 s	●	0.13 s	●	4.6 s	●	0.005 s
prep4_onehot	○	7 s†	○	5 s†	○	0.12 s	○	0.18 s	○	–‡	●	0.004 s
spi	○	6 s†	●	5 s†	○	0.14 s	○	0.13 s	○	–‡	●	0.012 s
uart	●	6 s†	○	5 s†	○	0.13 s	○	0.11 s	○	–‡	●	0.007 s
xtea	●	7 s†	○	5 s†	○	0.14 s	●	0.22 s	●	5.5 s	●	0.030 s

† time information mixed with all synthesis analysis; ‡ tool timed out in analysis; * tool crashes on analysis

TABLE I: Finite state machine finding characteristics. We record the capabilities of a tool at inferring finite state machines and how much information it can recover. Complete analysis is given by a ●, partial analysis by a ◐, and no analysis by ○. Some tools do not report the time spent on FSM analysis on its own and report an overall analysis time and are flagged as such.

Algorithm 4 Logic locking algorithm.

```
1: procedure LOGIC_LOCK(ast)
2:     T ← module_tree(ast)
3:     𝕄 ← modules(ast)
4:     𝕊 ← signals(𝕄)
5:     for (m, s) ∈ 𝕄_i, 𝕊_i do
6:         k ← obfuscate(s)
7:         m ← new_input(k)
8:     end for
9:     percolate_module_io(T)
10: end procedure
```

module body finding signals of interests. Once located, we use the ability to edit the AST node for the signal to expand it into an `xor` or `xnor` operation depending on the desired key bits. We add the newly introduced obfuscating signals to the module's input. When all modules have been explored, we propagate the new input signals into the top level module using the module tree, exposing the key signals as inputs.

IV. Experimental Results

We evaluate RTSec with respect to existing commercial tools and state of the art open-source tools. Our evaluation capabilities are limited to testing whether or not the tools can infer finite state machines. Note that all tools are capable of inferring flip-flop candidates, thus we do not report those. We summarize our findings in Table I. For testing we utilize Intel Quartus 20.1, Xilinx Vivado v2020.2, Synopsys Design Compiler J-2014.09-SP5-3, Yosys 0.9, and Pyverilog 1.3.0 when comparing them against our tools. The benchmarks utilized in our testing were obtained from OpenCores [9] and were selected for their broad scope as components of modern System on Chips (SoCs).

We were unable to test other tools for their instrumentation process, as available tools were not designed with that goal in mind. We believe that RTSec *is the first tool of its class, which is able to perform automatic analysis and logic enhancement for security purposes directly on HDL code*. In our experimentation, RTSec was capable of identifying and watermarking any FSMs in the tested designs and generating test fixtures capable of simulating the outputs of the FSMs.

V. Conclusion

In this work we presented RTSec as an automated way to perform analysis and add security features to designs at HDL level. RTSec uses the AST representation of the HDL to perform inference of logic primitives. We showed how we are able to use this inference to detect FSMs. We demonstrated that our RTSec's FSM inference scheme has on-par or better performance than that of existing tools. Using these capabilities and the ability to edit the AST we implemented two security mechanisms, watermarking and logic locking, which to our knowledge no other tool is capable of doing. For future work, we plan on extending RTSec to cover more extraneous HDL cases while including more security-related schemes.

Acknowledgements

This work was partially supported by National Science Foundation (CNS-1801599, CCF-2019283, CCF-2019310), and partially supported by KBR Inc.

References

[1] Cadence, *JasperGold Platform and Formal Property Verification App User Guide*, December 2020, version 2020.12.

[2] Synopsys, *VC Formal Verification User Guide*, December 2019, Version P-2019.06-SP2.

[3] ——, *Design Compiler Optimization Reference Manual*, December 2011, Version F-2011.09-SP2.

[4] ——, *Design Compiler User Guide*, December 2011, Version F-2011.09-SP2.

[5] Intel, *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*, June 2021, UG-20131.

[6] Xilinx, *Vivado Design Suite User Guide: Synthesis*, January 2021, UG901 v2020.2.

[7] A. Cui, C.-H. Chang, S. Tahar, and A. T. Abdel-Hamid, "A robust fsm watermarking scheme for ip protection of sequential circuit design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 678–690, 2011.

[8] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1601–1618.

[9] Various Contributors, "Opencores.org," 2021.