RISC-V SoC Hardware Vulnerability Detection Toolset

Team 41:
Mason Korkowski, Micah Mundy, Gerald Edeh, Kolton Keller, Eva Kohl, Savva Zeglin, & Magnus Anderson

Client/Advisor: Henry Duwe

Project Vision: Simplify Hardware Debugging

- Project: Hardware Vulnerability Analysis Toolkit

- More advanced processors → higher complexity → more vulnerabilities

- Use cases
  - Hardware CTF participants and Test engineers: first line of attack
  - Computer hardware developers: first line of defense

What is HACK@DAC and Why is it Important?

- Hardware Capture the Flag (CTF) event
  - Teams compete to detect bugs in a provided, flawed SoC design
  - Encourage creation of automated bug finding tools
  - Extra incentive to exploit the vulnerabilities that are detected
- Our Project
  - Specific focus on HACK@DAC competition
  - Following all competition rules

Source: https://hackatevent.org/

Goal: Develop a tool-set for hardware debugging.

- User groups: HACK@DAC participants and SoC developers
- Reduce the time and complexity of debugging RISC-V SoC Hardware.
- Approach: Create a toolset that decreases the time and effort needed to detect and exploit bugs by automatically finding issues in given RTL code

HACK@DAC provided SoCs

2018 SoC

- Past competition
- Provided:
  - Buggy and clean versions of SoC
  - Comprehensive list of bugs

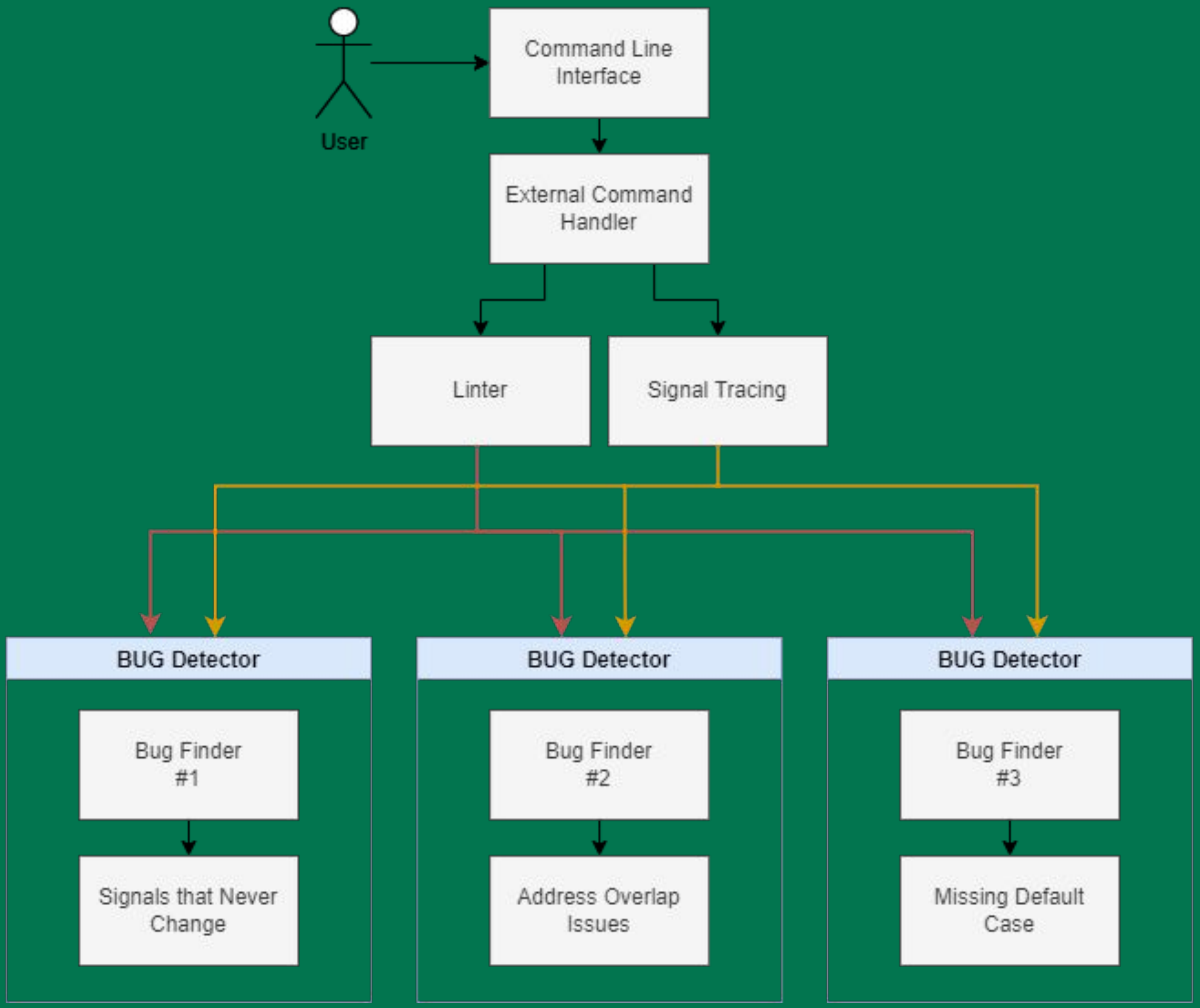
2021 SoC

- Ongoing competition
- Provided:
  - Buggy SoC

14	Incomplete case statement in ALU can cause unpredictable behavior.	riscv_alu.sv
15	Faulty logic in the RTC causing inaccurate time calculation for security-critical flows, e.g., DRM.	rtc_clock.sv
16	Reset for the advanced debug unit not operational.	adbg_tap_top.v
17	Memory-mapped register file allows code injection.	riscv_register_file.sv

System Overview

- **Who?**
  - **HACK@DAC competitors**
  - **RTL Verification Engineers**
  - **RTL Designers**
- **What?**
  - **Signal Tracing**
  - **Linting**
  - **Bug Detecting**
- **Why?**
  - **Open Source**
  - **Not reliant on Simulators**

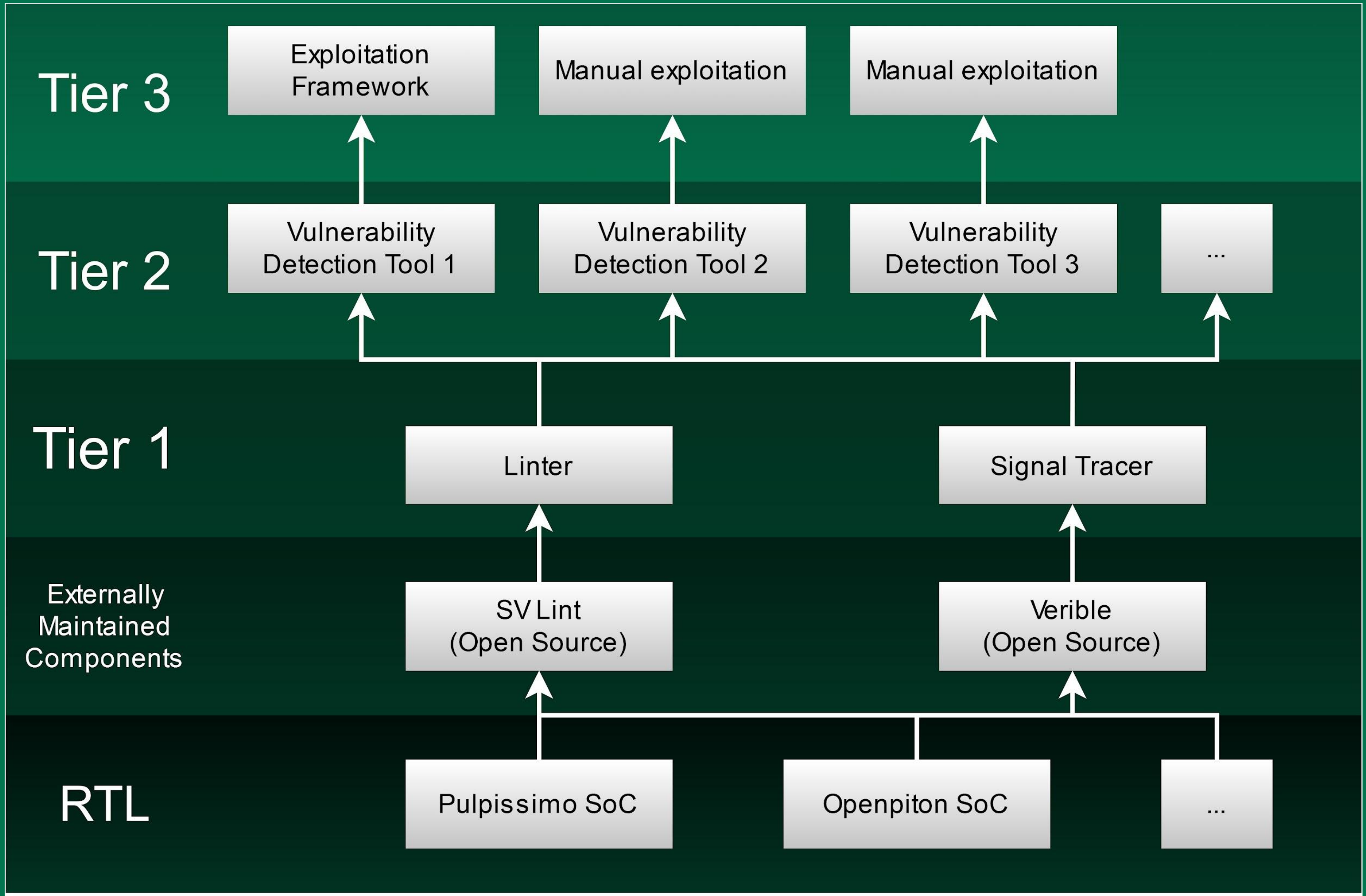


Summary of Requirements

End Deliverable: Suite of tools intended to find hardware-induced vulnerabilities in System-on-Chips (SoCs) from their RTLs

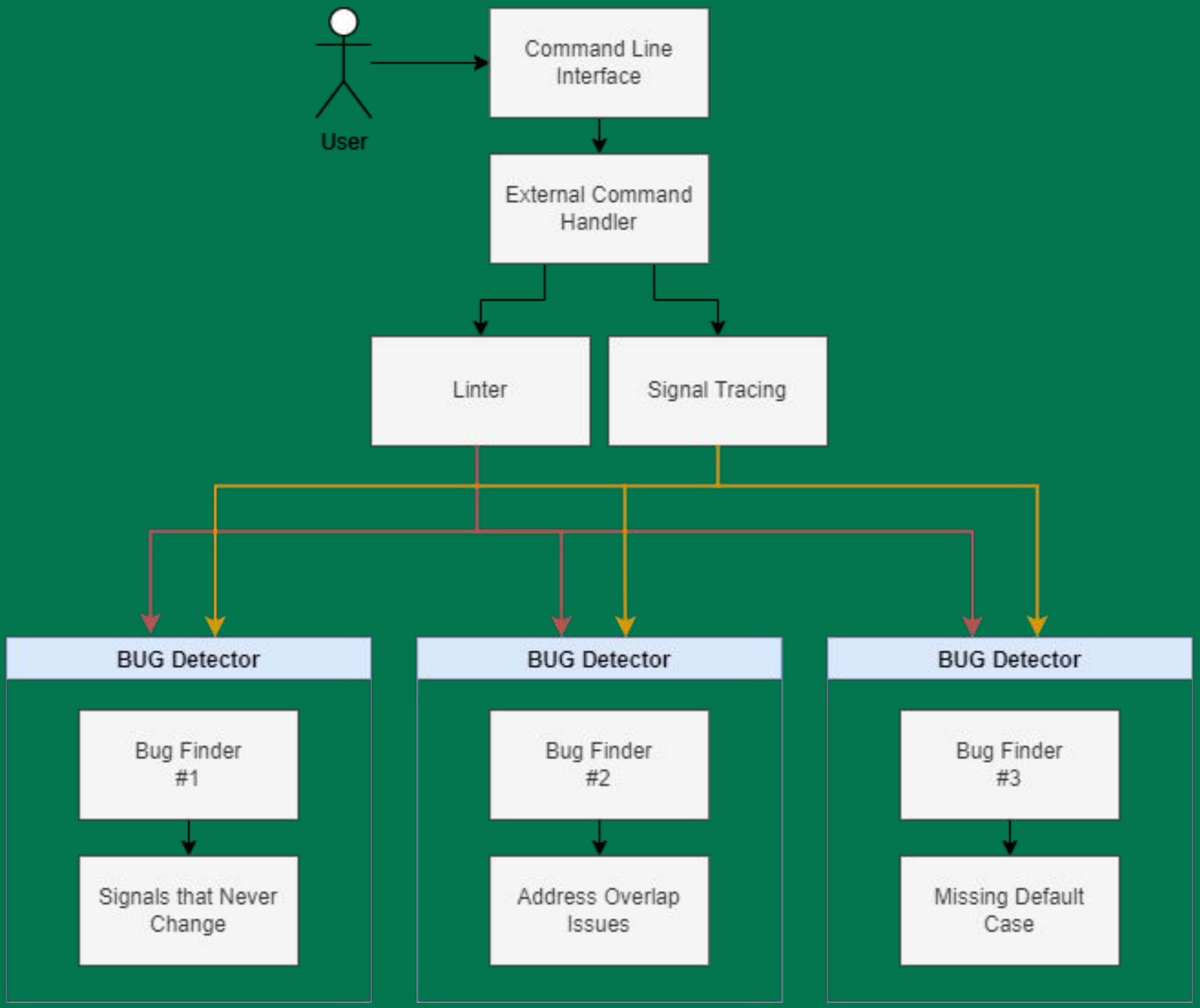
- Functional:
  - Tool Specifications:
    - Automatically find vulnerabilities, or give the user necessary information to manually find them
    - Accessible via a command line interface
    - Responsive: continuous, instantaneous feedback
- Non-Functional:
  - Hardware requirements: one laptop (4 cores, 16GB RAM)
  - Usable by someone with intermediate knowledge of hardware design
- Constraint: Operate according to HACK@DAC standards and rules

Conceptual Design Diagram



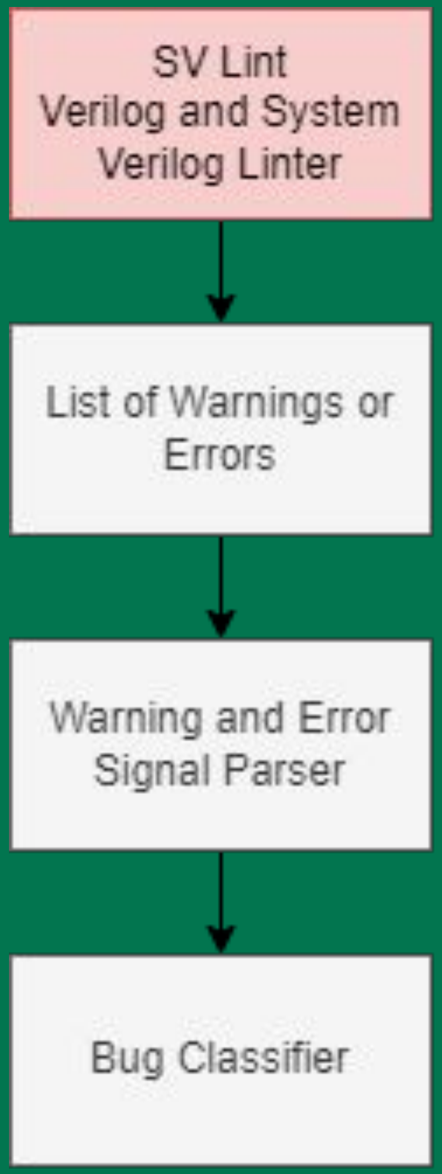
Program Flow

- User Input
  - Signal Names
  - File paths
- External Tools
  - SVLint
  - Verible Lexer
- Signal Tracing
  - Dependency tree building
  - Directory discovery
- Bug Classification
- Individual Bug Detectors
  - Signals that do not change
  - Address Overlaps
  - Missing Default Cases



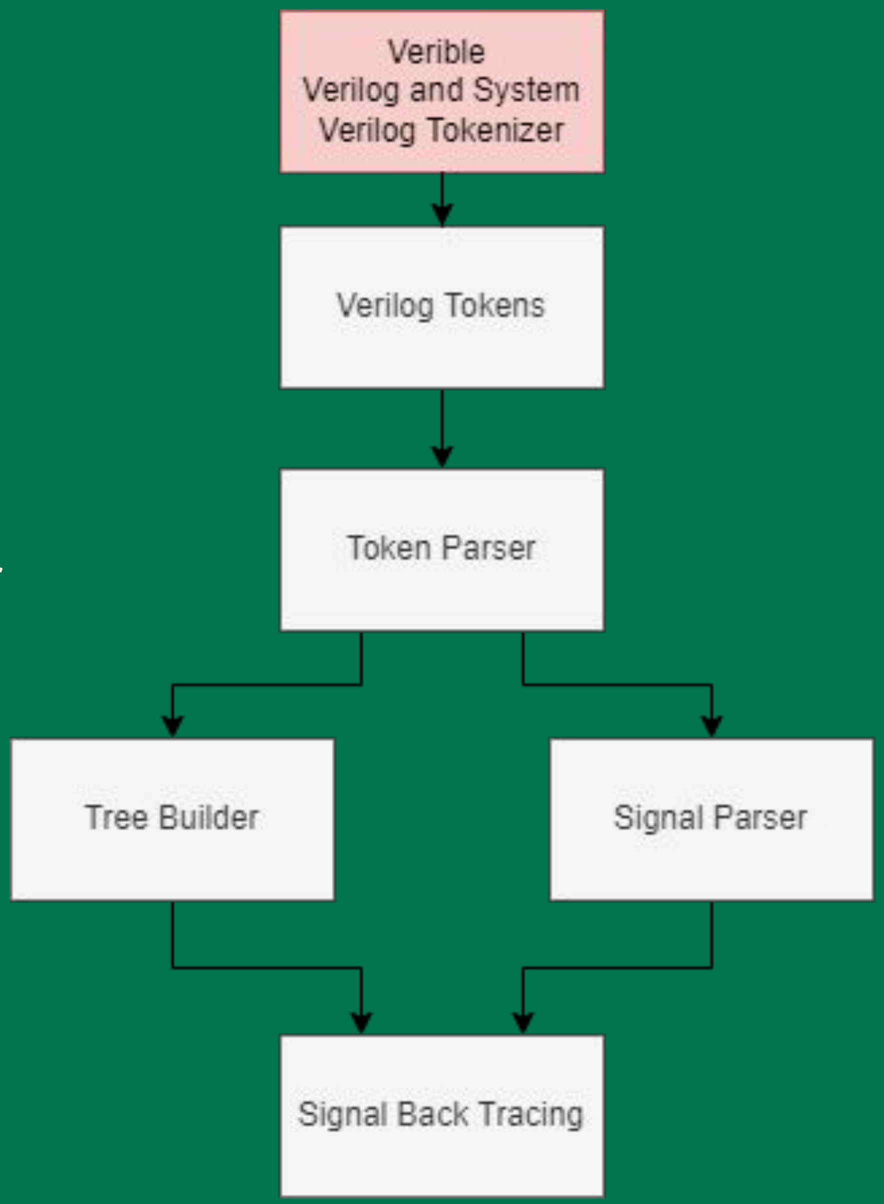
Linting Component

- What it does
  - Finds potential warnings/errors in RTL
  - Determines which are of concern
- Why we need it
  - Use errors from SVLint to find potential bug locations
  - Narrow down errors provided by SVLint
- How it works
  - Input RTL into SVLint
  - Additional logic to trim output of SVLint
    - Determine which issues have a high likelihood of being bugs that we can detect



Signal Tracer Component

- What it does
  - Builds hierarchy of modules in SoC design
  - Given a signal builds a dependency tree
- Why we need it
  - Bugs often rely/exist because of signals across multiple modules
  - Parsing System Verilog manually is difficult, however it is less difficult and quicker than a compilation tool
- How it works
  - Use Verible to generate tokens which we then parse
  - Build a module dependency tree
  - Hardware equivalent of a software call stack can then be created for any signal in the SoC



Bug Detection Tools

- Collection of tools that are based on the Signal Tracer Component
- Will be used to find bugs of specific categories
  - Initial bug categories/types we have chosen to attempt to detect
    - Signals That Never Change
    - Memory Address Overlap
    - Missing Default Case Statements
- Output:
  - Depending on type of bug:
    - Template exploit program
    - Where/how to set related signal
    - Other useful information for exploiting the bug

Prototype Implementation (Bug Detection)

1. Input 2018 HACK@DAC SoC into SVLint

```
$ svlint --include ./include/riscv_defines.sv riscv_alu.sv
```

2. SVLint indicates specific line and signal have issue:

```
Fail: case_default
   --> riscv_alu.sv:139:7
    |
139 |       case (vector_mode_i)
    |       ^^^^ hint  : `case` must have `default` in `always_comb` or `function`
```

```
case (vector_mode_i)
  VEC_MODE16: begin
    adder_in_b[18] = 1'b1;
  end

  VEC_MODE8: begin
    adder_in_b[ 9] = 1'b1;
    adder_in_b[18] = 1'b1;
    adder_in_b[27] = 1'b1;
  end
endcase
```

This corresponds to bug 14 from the 2018 bug list

14	Incomplete case statement in ALU can cause unpredictable behavior.	riscv_alu.sv	// take care of partitioning the adder for the addition case case (vector_mode_i) VEC_MODE16: begin

Prototype Implementation (Bug Exploitation)

3. Build a dependency tree of the 2018 SoC verilog modules provided a root file

4. Trace the signal found by the linter until it is eventually given a value.

```
riscv_core
|       cluster_clock_gating
|       riscv_if_stage
|       |       riscv_hwloop_controller
|       |       riscv_compressed_decoder
|       riscv_id_stage
|       |       riscv_register_file
|       |       |       cluster_clock_gating
|       |       riscv_decoder
|       |       riscv_controller
|       |       riscv_int_controller
|       |       riscv_hwloop_regs
|       riscv_ex_stage
|       |       riscv_alu
|       |       |       alu_popcnt
|       |       |       alu_ff
|       |       riscv_mult
|       riscv_load_store_unit
|       riscv_cs_registers
|       riscv_debug_unit
```

```
PRINTING SIGNAL TREE

      riscv_alu:               Origin =>        vector_mode_i
      riscv_alu:        vector_mode_i =>        vector_mode_i
 riscv_ex_stage:        vector_mode_i =>       alu_vec_mode_i
     riscv_core:       alu_vec_mode_i =>      alu_vec_mode_ex   Terminated
     riscv_core:       alu_vec_mode_i =>       alu_vec_mode_i   Terminated
 riscv_ex_stage:        vector_mode_i =>        vector_mode_i   Terminated
```

Prototype Implementation (Bug Exploitation)

5. After tracing the signal alu_vec_mode is set to VEC_MODE32 when the processor is reset.

6. The Initial case statement did not have a case for VEC_MODE32, picked out by the linter

7. By resetting, then immediately using an instruction, such as ABS (an instruction that does not set VEC_MODE) the ALU enters an undefined state.

```
always_comb
begin
  jump_in_id                 = BRANCH_NONE;
  jump_target_mux_sel_o      = JT_JAL;

  alu_en_o                   = 1'b1;
  alu_operator_o             = ALU_SLTU;
  alu_op_a_mux_sel_o         = OP_A_REGA_OR_FWD;
  alu_op_b_mux_sel_o         = OP_B_REGB_OR_FWD;
  alu_op_c_mux_sel_o         = OP_C_REGC_OR_FWD;
  alu_vec_mode_i             = VEC_MODE32;
```

```
case (vector_mode_i)
  VEC_MODE16: begin
    adder_in_b[18] = 1'b1;
  end

  VEC_MODE8: begin
    adder_in_b[ 9] = 1'b1;
    adder_in_b[18] = 1'b1;
    adder_in_b[27] = 1'b1;
  end
endcase
```

Design Complexity

- Challenges in development
  - Dependences of the 2018 SoC made it difficult to simulate
  - Lack of experience in complex hardware among team members made for a steeper learning curve.
  - Open-ended Scope
- Challenges in the design
  - One of the challenges in the design is exploiting a bug in the design.
  - To address this challenge, the agile project development model is used.

Project Plan

Major Milestones:	**Associated Risks / Mitigation Strategies:**
Categorize bugs from 2018 HACK@DAC SoC	Initial categorization may be inaccurate. Bugs may need to be re-categorized later based on new knowledge gained through Agile process.
Run programs on simulated 2018/2021 SoCs	2018 SoC has been unmaintained for years. Could inject similar bugs into more recent designs, or use 2021 SoC bugs for verification.
Create static analysis toolset to find bugs	High complexity could lead to low accuracy for tools. Attempt to keep tools as generic and frequently reassess our goals through Agile development.
Use tools to detect/exploit bugs SoCs	Highly dependent on all milestones above. Adhering to above risk mitigation strategies will be crucial for mitigating the risk of this task.
Develop new approaches to find bugs	We don’t yet know which methods will be reasonable. Rapid research and prototyping will help us quickly reach a sensible set of solutions.

Project Plan – Schedule/Milestones

Goal	Fit Criteria	Timeline
Categorize known bugs	Fit all bugs into at least one category	This Semester Weeks 4-8
Run programs on the simulated 2018 HACK@DAC SoC	At least one by the end of the Fall 2021 semester	This Semester Week 8-14
Run programs on the simulated 2021 HACK@DAC SoC	At least one by the end of the Fall 2021 semester	This Semester Week 4-6
Create static analysis toolset to find bugs (see next milestone for evaluation metric)	At least 3 different categories of bug detectable with at least 50% accuracy	Next Semester Weeks 2-12 (individual tools done in sprints of two weeks)
Generate concepts for new approaches to finding bugs in SoC designs	At least one, well documented, new approach	Next Semester Weeks 2-12
Using assistance of static analysis tools, detect and exploit bugs in each of the 2018 and 2021 SoCs	At least one bug from each SoC	Next Semester Weeks 4-12

Test Plan

Unit Testing: Each tool will be individually tested by injecting vulnerabilities into processor designs.

Interface Testing:

- The interface between the parsing core and the RTL
- The interface between the tools and user.

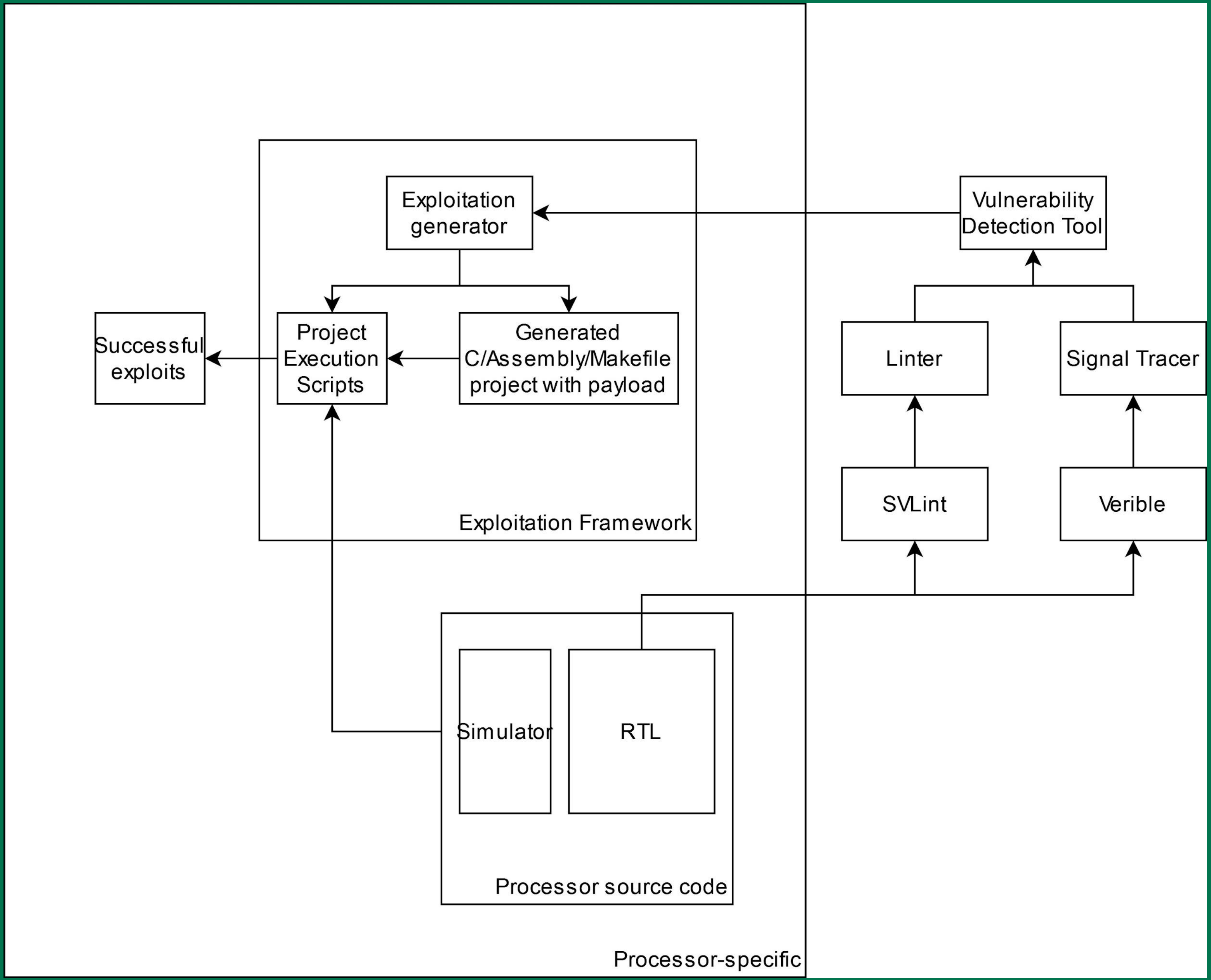
Integration Testing: Between the parsing core and the tools.

System Testing: Testing each module individually

Regression Testing: New bug detection tools will not break others

Acceptance Testing: Final Design will be evaluated based on the requirements.

Measure Of Success: The proportion of HACK@DAC bugs our tools can locate. Extra successful if we can use tool output to exploit bugs.


Exploitation generator
Vulnerability Detection Tool
Successful exploits
Project Execution Scripts
Generated C/Assembly/Makefile project with payload
Linter
Signal Tracer
SVLint
Verible
Exploitation Framework
Simulator
RTL
Processor source code
Processor-specific

Current Progress and Future Plans

Done this Semester

- Identified/classified bugs
- Explored and identified helpful open source software
- Proof of concept prototypes for the backbone of the toolset (Signal tracing, linting)

Plan for Next Semester

- Perfect Signal Tracing/Linting Components
- Develop and test tools for:
  - Missing Case Statements
  - Signals That Never Change
  - Memory Address Overlaps
- Use above tools to detect/exploit bugs

Questions?