RISC-V SoC Hardware Vulnerability Detection Toolset

DESIGN DOCUMENT

Team 41

Mason Korkowski
Micah Mundy
Gerald Edeh
Kolton Keller
Eva Kohl
Savva Zeglin
Magnus Anderson

Client/Advisor: Henry Duwe

sdmay22-41@iastate.edu
https://sdmay22-41.sd.ece.iastate.edu/docs.html

Revised: 12/08/2021 v3

Executive Summary

Systems on Chip's (SoC's) are largely increasing in popularity as pervasive technology. As the popularity increases, so does the complexity, size, and scales. With larger designs comes the challenge of testing and verification to ensure there are not any system critical bugs that can lead to security vulnerabilities. The goal of our project is to develop a suite of tools to aid a developer in the detection and exploitation of bugs in Register Transfer Level (RTL) designs.

Similar to software Capture the Flag (CTF) competitions to find vulnerabilities in software systems, hardware CTF events also exist. One of these competitions is the HACK@DAC competition. The primary objective of these CTF events is to encourage the creation of automated tools that can detect and exploit security vulnerabilities. These competitions ideally promote SoC designs to become more robust as these competitions lead to the development of tools--often open-sourced--that can help cybersecurity and RTL designers test their designs. This early testing and detection is important in the hardware industry as it could save billions of dollars system critical bugs are found before the design is fabricated as an Application Specific Integrated Circuit (ASIC).

Due to the scale and complexity of finding all bugs in all hardware SoC designs, we decided to limit our scope. For our design, we are specifically focusing on aiding participants of the HACK@DAC hardware CTF competitions. In doing so we opted to follow all the rules set forth by the competition. We have chosen to base our project on this competition since it provides a testing environment for our design. After the completion of previous competitions, detailed lists of any past bug found were released. These lists give a strong foundation for testing our tools to verify their effectiveness. The HACK@DAC competition heavily rewards teams who can exploit the bugs as well as detect them, so our project also includes a focus on providing assistance with exploiting our detected bugs.

Table of Contents

Summary of Requirements

- The product will be a suite of tools intended to find hardware-induced vulnerabilities in System-on-Chips (SoCs) from their RTL implementations
- Tools will automatically find vulnerabilities, or give the user necessary information to manually find them
- These tools will be accessed via command line interface
- These tools will respond to user input, provide status updates, and complete execution within specified time constraints
- The product will be capable of running on a computer running a VM with 4 cores and 16GB RAM
- The product will operate according to HACK@DAC standards and rules
- The user will be someone with intermediate knowledge of hardware design

1 Team

1.1 Team Members

Mason Korkowski

Micah Mundy

Gerald Edeh

Kolton Keller

Eva Kohl

Savva Zeglin

Magnus Anderson

1.2 Required Skill Sets for Your Project

(if feasible – tie them to the requirements)

- Understanding and proficient in Cybersecurity Principles
- Understanding and proficient ability to program in hardware description languages.
- Understanding of SoC design files project structure practices.
- Experience in hardware and software programming best practices.
- Experience with and proficient in git and vsc practices.
- Experience in hardware and software debugging techniques.
- Strong problem solving skills.
- Proficient communication skills.
- Experience with and proficient in basic programing data structures.
- Experienced understanding of basic processor architecture.
- Basic understanding of software architectures.
- Basic understanding of bash and scripting languages.

1.3 Skill Sets covered by the Team

(for each skill, state which team member(s) cover it)

Understanding and proficient in Cybersecurity Principles

- Kolton

Understanding and proficient ability to program in hardware description languages

- Savva, Mason

Understanding of SoC design files project structure practices

- Mason

Experience in hardware and software programming best practices

- All team members

Experience and proficiency in GitLab

- All team members

Experience in hardware hardware and software debugging techniques

- All team members

Strong problem solving skills

- All team members

Good communication skills

- All team members

Experience with and proficient in basic programing data structures

- All team members

Basic understanding of basic processor architecture.

- Savva, Mason, Micah, Kolton

Basic understanding of software architectures.

- Magnus, and Eva

Basic understanding of bash and scripting languages.

- All team members

1.4 Project Management Style Adopted by the team

Our team will manage our project with an Agile/SCRUM methodology. This will help mitigate risks during the development process as well as ensure that our end-product is as useful as possible.

2 Introduction

2.1 Problem Statement

Increasingly our pervasive technology is being dominated by systems on chips (SoCs). Unfortunately, the increasing complexity of such designs makes removing hardware bugs and security vulnerabilities especially challenging. The goal of this project is to construct a tool set framework that can be used for hardware capture-the-flag competitions such as https://hackathard.com/hacksec21/. Students will be encouraged to use an agile approach to the development of the tool set based on publicly-available buggy SoC designs (e.g., https://github.com/gdessouky/hackdac_2018_beta) as well as participation in a hardware capture-the-flag competition.

2.2 Requirements & Constraints

2.2.1 Functional Requirements

General Toolset

G1. Each tool in the product shall have suggested exploitation options.

Signal Tracing

S1. When the user runs the Signal Tracer tool, the product shall accept the root file of SoC and name of signal to trace
S2. If the the user gives the Signal Tracer a non root file, the product shall display error
S3. If the user gives an invalid signal name then the product shall display and error message
S4. When the product accepts root file and signal name, then the product shall display the signal file path to user

Address Overlap

A1. When the user runs the Address Overlap tool, the product shall except a SoC root file
A2. If the user gives a invalid root file then the product shall display an error message
A3. When the product accepts an SoC file, the product shall display line numbers of and descriptions of potential address overlap locations.
A4. If the product finds no overlaps, then the product shall exit with a success display message.

Missing Default

M1. When the user runs the Missing Default tool, then the product shall except a SoC file
M2. If the user gives an invalid file, then the product shall display an error message
M3. When the product accepts an SoC file, then the product shall display lines with line numbers where there is a missing case statement.
M4. If the product finds no missing case statements, then the product shall exit cleanly.

Unchanging Singal

US1. When the user runs the Unchanging Signal tool, the product shall accept a SoC root file and a signal name or file from SoC design

US2. If the user inputs an invalid signal name then the product shall display an error message

US3. If the user inputs an invalid file name then the product shall display an error message

US4. If the user inputs an invalid root file then the product shall display an error message

US5. When the product accepts the root file and file or signal name, then the product shall display:

   a. the signal path
   b. values at each point on path
   c. Boolean of whether the signal has changed

US6. If off nominal behavior happens during execution within the product, the product shall display an error message

2.2.2 Nonfunctional Requirements

Look and Feel

LF1. The product shall have some type of command-line user interface, tailored to hardware/security engineers.

Usability

U1. The product shall be usable within an hour by someone with intermediate knowledge of hardware design to identify bugs

Performance

P1. The product respond to user input within 1 second on a laptop with 4 cores and 16GB RAM

P2. The product shall provide periodic, detailed status updates and progress reports within 5 seconds

Operational

O1. Tools in the product shall be able to be run on a laptop running a VM with 4 cores and 16GB RAM (constraint)

Maintainability

M1. The project shall have a well documented README or each tool for knowledge transfer purposes.

Legal

L1. The product will operate according to HACK@DAC standards and rules. (constraint)

2.2.3 Constraints

1 The product shall be completed by May 2022.

2 The product shall be implemented without purchasing anything, such as software licenses.

3 The product shall conform to the rules and regulations of HACK@DAC competitions.

2.3 Engineering Standards

2.3.1. IEEE Policies, Section 7: Shall be honest and realistic in stating claims or estimates based on data.

- 2.3.1.1. Standard ensures that the team will be honest with the data recorded.
- 2.3.1.2. Creates a trustworthy environment with the client, as data is sincere.

2.3.2. IEEE Policies, Section 7: Shall treat all persons fairly and not engage in acts of discrimination

- 2.3.2.1. Standard ensures that students treat one another with respect and honesty.
- 2.3.2.2. Creates a safe environment for students to work in, and discuss ideas.

2.3.3. IEEE 1800-2017 : Standard for SystemVerilog

- 2.3.3.1. Standard for the SystemVerilog Language, This specification lays out the standards for syntactically correct SystemVerilog code that will need to be parsed for this project.

2.3.4. IEEE 1212.1-1993 : Standard for Communicating Among Processors and Peripherals Using Shared Memory (Direct Memory Access - DMA)

- 2.3.4.1. Standard for DMA use in microarchitectures for communicating between memory sub modules. May be necessary for detection of bugs in DMA modules.

2.3.5. RISC-V Instruction Set Manual

- 2.3.5.1. The RISC-V Instruction Set Manual lists all the instructions that are able to be executed by the RISC-V architecture. This will be very useful when writing test code for the buggy SoC and it may be used for detection of bugs in the software portion of the design, such as with input fuzzing tools.

2.3.6. Style guides: enable us to write software that visually appears unified to reduce the time required for understanding and modifying the code.

- 2.3.6.1. Google Shell Style Guide

2.4 Intended Users and Uses

Future participants of the HACK@DAC competition benefit from the results of the project. Finding bugs in a program can be very time consuming and stressful. By using the tool they will be able to find apparent bugs in the program. Also someone that is using HACK@DAC as practice to improve their debugging skills will benefit from this tool. In both scenarios, a use case can be using the tools to assist them when finding bugs. This will give more time to find other bugs in the program.

3 Project Plan

3.1 Project Management/Tracking Procedures

Our group is adopting Agile as our project management style. Agile fits our project's goals in that it allows for flexibility in requirements and scheduling. Because the scope of our project is still being realized and cemented, we anticipate much change within the implementation of the toolset. Given the turbulent nature of our project, the agile methodology is essential for success.

We will use GitLab to track our team's progress. GitLab includes a repository for the software and tools we develop and an issue tracker to monitor progress. Our primary communication will take place via Discord.

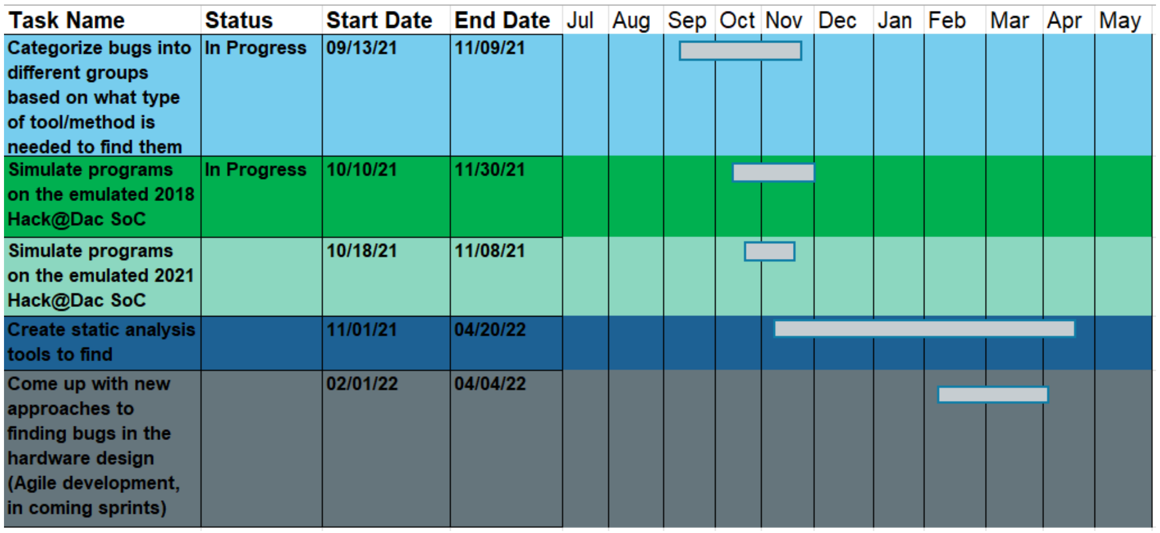
3.2 Task Decomposition

- Categorize bugs into different groups based on what type of tool/method is needed to find them
    - Determine the security properties that are violated for bug categories
    - Determine the information needed by tools to detect the bugs in each category
- Run programs on the simulated 2018 HACK@DAC SoC
    - Get simulation environment set up
    - Exploit a vulnerability on the SoC
- Run programs on the simulated 2021 HACK@DAC SoC
    - Get simulation environment set up
    - Find a vulnerability in the SoC
    - Exploit a vulnerability on the SoC
- Create static analysis toolset to find bugs (for applicable bug categories)
    - Research open-source static analysis tools for RTL (verilog linters/parsers)
    - Develop or find a tool that can trace signals through the SoC design to aid other tools
    - Extend the signal tracing tool in order to detect/exploit more specific classes of bugs (address overlap, missing case statements)
- Using assistance of static analysis tools, detect and exploit a bug in the 2018 and 2021 SoCs
- Generate concepts for new approaches to finding bugs in SoC designs (Agile development, in coming sprints)
    - Explore alternative design options

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

- Categorize known bugs from the HACK@DAC 2018 into different groups based on what type of tool/method is needed to find them
    - Fit all bugs into at least one category
- Run programs on the simulated 2018 HACK@DAC SoC
    - At least one by the end of the Fall 2021 semester
- Run programs on the simulated 2021 HACK@DAC SoC
    - At least one by the end of the Fall 2021 semester
- 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
- 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
- Generate concepts for new approaches to finding bugs in SoC designs

3.4 Project Timeline/Schedule



Task Name	Status	Start Date	End Date	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May
Categorize bugs into different groups based on what type of tool/method is needed to find them	In Progress	09/13/21	11/09/21			■	■	■						
Simulate programs on the emulated 2018 Hack@Dac SoC	In Progress	10/10/21	11/30/21				■	■						
Simulate programs on the emulated 2021 Hack@Dac SoC		10/18/21	11/08/21				■	■						
Create static analysis tools to find		11/01/21	04/20/22					■	■	■	■	■	■	
Come up with new approaches to finding bugs in the hardware design (Agile development, in coming sprints)		02/01/22	04/04/22								■	■	■	

3.5 Risks And Risk Management/Mitigation

Task	Risk	Probability	Mitigation Plan
Bug categorization	Later in the project, bugs need to be re-categorized to fit with new knowledge or developments	0.7	When beginning development of a new tool to identify bugs, revise categories of bugs as needed. Discuss which bugs the tool is likely to find and if the bug categories need to be refined.
Run programs on the simulated 2018 HACK@DAC SoC, a variant of the popular Pulpissimo SoC with injected vulnerabilities.	This project has been abandoned for three years, and it uses several IP cores that have been updated since then.	0.4	The most recent Pulpissimo design can already be simulated. Inject similar vulnerabilities into the most up-to-date Pulpissimo platform. The list of vulnerabilities in the HACK@DAC 2021 SoC will be released sometime in the spring, so that SoC can eventually be used for verification instead.
Run programs on the simulated 2021 HACK@DAC SoC	No risk. The competition is ongoing and there are instructions available. We have successfully completed this task at this point.	0	
Create static analysis toolset to find bugs (for applicable bug categories)	The tools we create may be too specialized for the 2018 and 2021 SoCs. The complexity needed to create these tools may be high, resulting in low accuracy in bug detection.	0.6	When creating tools, we plan to keep our designs as generic as possible, so that the tools will work on any verilog processor design. Our Agile project plan will assist in mitigating the risk of a tool becoming too complex by frequently re-assessing our goals and our progress towards those goals.
Using assistance of static analysis tools, detect and exploit bugs in the 2018 and 2021 SoCs	Highly dependent on success of all previous tasks. Also dependent on the tools we create being accurate and providing us with insight on how to exploit the bugs.	0.8	During the tool development process, thorough testing will be critical in order to ensure that the tools are accurate. Adhering to above risk mitigation strategies will mitigate most of the risk for this task.
Generate concepts for new approaches to finding bugs in SoC designs	With Agile development, we don't know yet which categories/types of bugs will be attainable to detect.	0.5	Research and rapid prototype development will be key. Quickly determining whether detecting/exploiting a bug type is feasible will be critical to ensuring that we don't commit too much time to an idea that will eventually fail.

3.6 Personnel Effort Requirements

Task	Expected Time	Explanation/ Textual Reference
Bug categorization	1-3 weeks (may need reassessed)	Spend some time locating each bug in the 2018 SoC and determining categories that the bugs fit into. Later in the development of the project, these categories may need to be revised.
Run programs on the simulated 2018 HACK@DAC SoC	4-6 weeks	Have to figure out and learn how to run the emulated programs. There have been issues with running the program that have delayed progress. The programs required have changed since 2018, so the
Run programs on the simulated 2021 HACK@DAC SoC	1 week	HACK@DAC has provided a very useful instruction manual on how to set up the 2021 SoC, so this task is simple to complete.
Create static analysis toolset to find bugs (for applicable bug categories)	13-14 weeks	This is one of the main tasks for the project. Through Agile development, we hope to constantly update our expectations and quickly produce prototypes for static analysis tools for bug detecting.
Using assistance of static analysis tools, detect and exploit bugs in the 2018 and 2021 SoCs	3-4 weeks	Upon completion of each static analysis tool, we will use the tool's output to attempt to exploit a detected bug. The difficulty of this task depends on the quality of the tool and the nature of the bug that was detected.
Generate concepts for new approaches to finding bugs in SoC designs	5-8 weeks	Throughout our Agile development process, if a tool is completed or determined to be a less valuable use of our time, the design and implementation process can start over again for a new tool. Depending on how quickly tools are completed or discarded, the concept generation part of the process may be repeated many times.

3.7 Other Resource Requirements

- Google Drive Tools Suite
- Gitlab
- Software IDE(s)
- Questasim / other hardware simulation software
- Virtual machine for simulation, other tasks
- Personal computers for use of above tools and software

4 Design

4.1 Design Context

4.1.1 Broader Context

Since our project consists of a set of software tools, we have not identified any major public safety, global, or environmental impacts. However, since a secondary use for our toolset is to find bugs during the designing/testing process of commercial SoCs, our tool could improve performance and decrease the amount of development time needed in these areas. This could increase cost effectiveness for groups designing SoCs.

4.1.2 User Needs

HACK@DAC teams and hardware designers/testers need a toolset that assists them in detecting bugs in buggy SoC chips because finding bugs manually can be challenging and time consuming for them.

4.1.3 Prior Work/Solutions

RTL developers frequently need to assess the security of their hardware, but few solutions exist to assist this process. We currently know of four solutions that are available to us:

1. Verilator
2. Questa Sim
3. Quartus Prime
4. Vivado

Despite their effectiveness, these systems require (1) a significant amount of human labor, (2) a license, (3) extensive training, and/or (4) several dependencies. In addition, no known tool is capable of finding every vulnerability in hardware.

To mitigate these issues, we propose a lightweight, open-source toolset that includes an intuitive interface and installation procedure without the need to compile, synthesize, or simulate RTL. Such a toolset may identify less vulnerabilities than the industry standard tools, but makes common RTL analysis procedures faster, more convenient, and more accessible.

4.1.4 Technical Complexity

1. Since our project is to find bugs in an SoC, we are using many software engineering and security principles. Three principles that we are especially interested in: testability, maintainability, and integrity. The heart of the HACK@DAC competition is that SoC chips must be testable to find the hardware bugs. There must be methods and systems to identify bugs in the testing process. The second principle that we are focusing on is maintainability. Our project must be maintainable in that as SoC chips change and the competition changes from year to year, our software needs to accommodate those changes. Finally, we want to have integrity in the code that we produce so that we create a quality solution.

2. Our problem scope contains multiple challenging requirements. As mentioned previously, detecting and exploiting bugs is a complex and challenging task. In section 4.1.3, we mention other similar, existing solutions. We intend to produce a design that exceeds these solutions in a few key ways. First, our project will be simple to install and use. Second, our design will be open source. And third, our project won't rely on compiling, synthesizing, or simulating RTL in order to detect bugs. By adhering to these three properties, our design will likely be useful in a way that existing tools are not to an audience of HACK@DAC teams or hardware designers/testers.

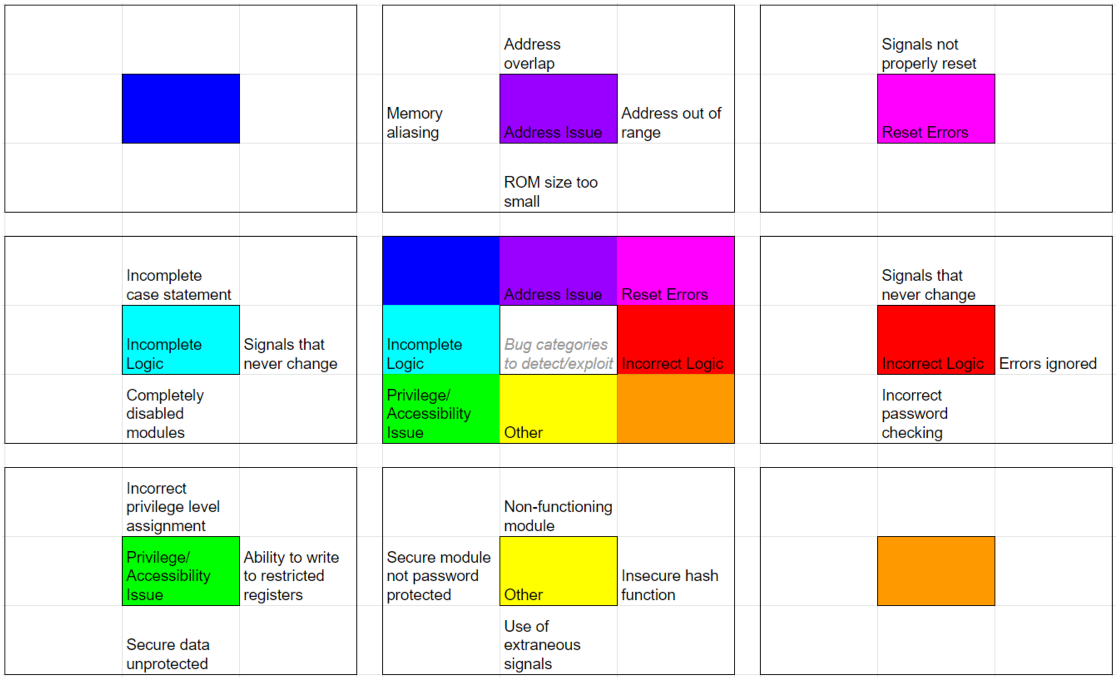
4.2 Design Exploration

4.2.1 Key Design Decisions

- How to set up toolset environment (What will the user have to do in order to download and use the tools)
- Programming Language(s) we will be using to create the tools
- Which open-source software tools that already exist to use as a part of our solution
- Priority order for which methods to use to find bugs in a simulated SoC
- Whether or not we should exploit the bugs we find through simulation

4.2.2 Ideation

In order to start developing tools to detect bugs, we must first determine which categories or types of bugs we think are most worthwhile to detect. We have created a lotus blossom with the categories of bugs that we have identified from the HACK@DAC 2018 SoC bug list. The categories are: Address Issue, Reset Errors, Incomplete Logic, Incorrect Logic, Privilege/Accessibility Issue, and Other.


Address overlap
Memory aliasing
Address Issue
Address out of range
ROM size too small
Signals not properly reset
Reset Errors
Incomplete case statement
Incomplete Logic
Signals that never change
Completely disabled modules
Address Issue
Reset Errors
Incomplete Logic
Bug categories to detect/exploit
Incorrect Logic
Privilege/ Accessibility Issue
Other
Signals that never change
Incorrect Logic
Errors ignored
Incorrect password checking
Incorrect privilege level assignment
Privilege/ Accessibility Issue
Ability to write to restricted registers
Secure data unprotected
Non-functioning module
Secure module not password protected
Other
Insecure hash function
Use of extraneous signals

4.2.3 Decision-Making and Trade-Off

Our team determined four factors that we believe make it worthwhile to attempt to detect bugs from different categories. Those factors are our estimated ease of detecting the bug types, our estimated ease of exploiting the bug types, our estimate of the speed at which a program could detect the bug type, and the relative frequency of similar bugs in the 2018 SoC design.

In the decision matrix below, higher numbers indicate that a bug category is considered by our team to be more worthwhile to try to detect.

We determined that Address Issue and Incomplete Logic were the most worthwhile categories of bugs to attempt to detect first.

Additionally, given that the HACK@DAC competition highly rewards the exploitation of found bugs, we also decided that we would work towards exploiting bugs that our tools can find.

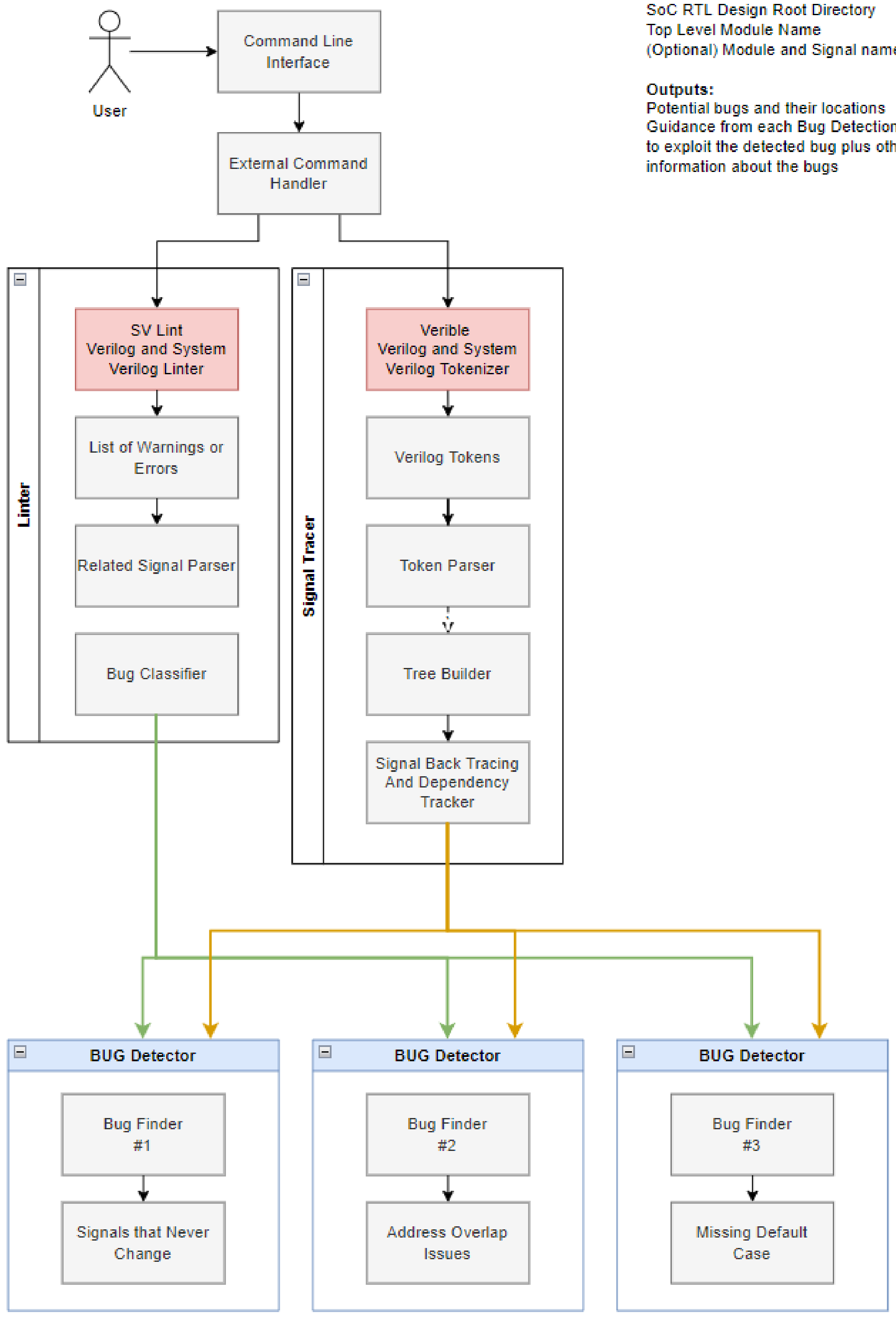
Bug Category	Estimated Ease to Detect	Estimated Ease of Exploitation	Estimated Detection Speed	Relative Frequency of Similar Bugs in 2018 SoC	**Total Score**
Address Issue	4	4	4	4	**16**
Reset Errors	3	3	4	4	**14**
Incomplete Logic	5	4	4	3	**16**
Incorrect Logic	2	3	4	5	**14**
Privilege / Accessibility Issue	2	4	3	3	**12**
Other	1	2	3	1	**7**

4.3 Proposed Design

Tasks done so far / in progress:

- Classifying Bugs
- Research and use Verilog Parsers/Linters
- In progress: Prototyping Verilog hierarchy builder and signal tracer
- In progress: Simulate HACK@DAC 2018 SoC
  - This SoC is based on the Pulpissimo platform. The current version of this SoC can be simulated.
- Simulate HACK@DAC 2021 SoC
- In progress: Exploiting a bug from 2018 SoC

4.3.1 Design Visual and Description



Inputs:
SoC RTL Design Root Directory
Top Level Module Name
(Optional) Module and Signal name to track

Outputs:
Potential bugs and their locations
Guidance from each Bug Detection tool on how to exploit the detected bug plus other information about the bugs

Our design plan features one large-scale signal tracing unit, and several small bug detection tools that depend on the signal tracing module.

Starting from the top of the diagram, the user will interact with a command line interface. It is assumed the user understands how to navigate using a command line and has an understanding of the different commands. Next the user will provide the command line with a few key items, first is the directory that the design is in, second is the name of the top level verilog or system verilog file, and third is any signal that needs to be traced.

Once the command has been sent, the external command handler will set up and run the external tools that we are using for this project. After receiving the output from the external tools, the Verible lexer and tokenizer is used to create a dependency tree. It provides the tokens from each file. Using these tokens we are able to find key information about each file such as the module name and what modules it depends on. Since we can find its dependencies we can build a tree that lists dependencies. Using this tree and given a signal name, we can then trace the signal throughout the design. Another external tool that will be used is SVLint, which looks at a Verilog file and provides any warnings or errors. Since running this linter often provides several hundred errors or warnings, we need to check if the error is something our tools can detect.

These warnings are passed to a bug classifier that determines if this is a type of bug that we understand. If it is, we send the relevant signal tracing information as well as the bug error information to a bug detection tool that is specifically designed to deal with that type of bug. This tool then will determine the likelihood that the error/warning is an actual problem within the SoC. If this likelihood is high, the tool will give the user some information to assist in exploiting the bug. Not all bugs are created equal, so there is no one-size-fits-all solution. Depending on the bug type, the tools can output a code snippet/template to run on the SoC, information about how to set a specific signal, or other useful information about the bug.

4.3.2 Functionality

This completed project will be applicable to the scenarios of everyday RTL design evaluation and HACK@DAC competition participation.

The current design satisfies the functional requirements in that there is a plan and design to complete each of the four tools listed in the requirements to the necessary performance standard listed in the nonfunctional requirements. The current plan gives space and encouragement towards further improving the requirements to meet goals without gold plating or excessive requirements. The nonfunctional requirements line up with the goal and context of the project and there is time within the plan for their implementation.

When the product is finished, the developer who chooses to use it will do the following. He will clone the git repository and read the READMEs to understand the tools given. He will cd into the repository and find the tools we have specified in the requirements. To run the tools he will simply follow the README instructions and the output will give insight into potential bug locations and exploitation techniques.

4.3.3 Areas of Concern and Development

One of our largest concerns for this project at this point is exploitation of a bug. This is a very difficult and complex task that has teams of professional hardware cybersecurity specialists working on it. We are worried that our skill set does not match that and that our knowledge past a basic check is lacking while compared to other teams or tools that already exist.

In order to mitigate this concern, we are using the Agile project development model. This allows us to constantly update our expectations and to ensure that we are focusing our efforts in the correct areas of the project. We are able to modify our tools and the types of bugs we can detect in order to maximize our ability to discover and exploit them.

4.4 Technology Considerations

The strengths of our project lie in the realm of system compatibility, availability, and ease of use. Our project is dependent on Verible and SVLint, but not simulation software. This makes the tools lightweight and easily compatible with the developers system. Being that it will be open-source, this will make it easily accessible to any developer who desires to use it. Finally, with good documentation and intuitive design, these tools will be user friendly and give insightful debugging information.

The weakness of this project is that it is limited in the categories of bugs it seeks to find and identify. As displayed earlier in the ideation section of this document, it is displayed that there are more bugs found in SoCs then our project will be programmed to identify. This means that there is blind spots and the developer that seeks to use this completed project should be aware that this project is only one set of tools to find bugs but is not a catch-all of all system issues.

One trade off we made initially was in our primary language selection. We decided to start working in Java for several reasons. Although other languages, like C based languages, may be faster concerning runtime and often easier to develop on Linux environments, we chose Java because all our team members have a large amount of experience with it due to ISU courses, string manipulation is easier in Java than C based languages, and the automatic garbage collector in Java allows us to avoid memory leaks and other issues that could cause problems in a large code base.

One alternative design was to use hardware fuzzing to detect bugs. This would've used SoC simulation, and would've been able to detect different classes of bugs that our current design currently does not detect. There are two main reasons we decided not to include fuzzing as part of our design. The first is that by only using static analysis, our tools will be lightweight and easily installable by any user. Simulation is more heavyweight and platform dependent. The second main reason is that after much consideration, we concluded that fuzzing was just not realistic for the scope of our project. This is due to the amount of time already spent on deciding the scope, as well as the difficulty of simulating one of the buggy HACK@DAC SoCs.

4.5 Design Analysis

To the extent of the understanding of the team, the current design plan is completely operational and compatible for the goal of the project. Strengths of the plan lie within well thought out architecture and clearly defined requirements. Up to this point, we have undergone several iterations for our design, and we anticipate having to iterate more in the future. Throughout the lifetime of our Agile project plan, we modify our design choices as the team and client see fit.

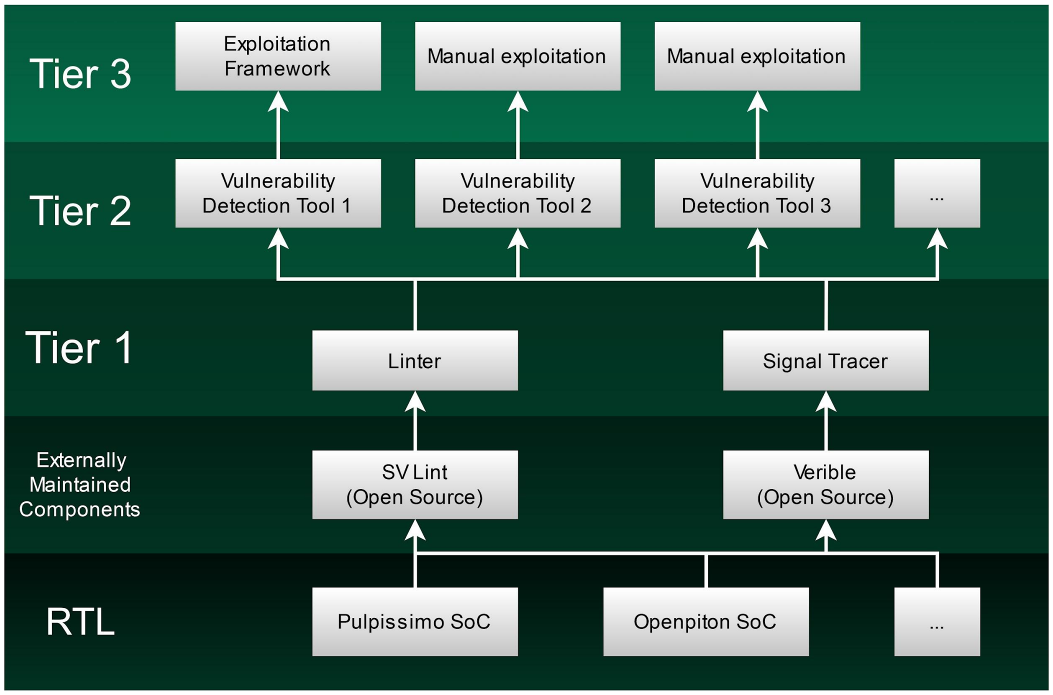
4.6 Design Plan

One of the most important requirements of this project is a complete, accurate toolset, so the initial design plan is to develop and test a single tool before beginning work on the others.

This tool will depend on the signal tracer and SystemVerilog linter (collectively called "cores"), so we plan to create these first. Integration between the tracer, linter, and tools will begin while verifying the accuracy of the tracer and linter to maximize concurrent work.

After one tool fulfills its requirements, work may begin concurrently on the other vulnerability detection tools.

Below is a dependency diagram of the project's internal components. In this diagram, each tier represents a stage in the development process. We aim to complete both tiers by the end of the spring semester.



A more thorough schedule of the project implementation can be found in section 6.

5 Testing

5.1 Unit Testing

The proposed project comprises independently operated tests that each aim to semi automatically detect different vulnerabilities in SoCs by analyzing their designs in the Register-Transfer Level (RTL) written in SystemVerilog. Therefore, each tool can be individually tested by intentionally injecting these vulnerabilities into multiple common, open-source processor designs. If a test can detect the injected bugs without misidentifying a substantial amount of untouched RTL, the test meets the expectations listed in section 2.

Despite each tool's uniqueness, all will share two core units responsible for linting extracting the structure of the RTL (a core called the signal tracer). This module will be independently, manually tested on existing projects written in SystemVerilog throughout the development process to ensure its accuracy.

5.2 Interface Testing

There are two interfaces:

- between the cores and the RTL. This interface is tested by aggregating a repository of projects written in SystemVerilog and manually asserting that the structure produced by the cores match the RTL.
- between the tools and a user. There are minimal requirements for this interface, but we continuously work with the client to assess the efficiency of controlling these tests.

5.3 Integration Testing

The only required integration is between the cores and the tools. The units are so tightly coupled with the parser that any testing method on the modules will validate the integration of the cores.

5.4 System Testing

Each bug detection tool is independent of the others, so testing each module individually validates the whole system.

5.5 Regression Testing

New bug detection tools will not break others due to their independence. Updates to existing tools will be unit-tested per section 5.1.

5.6 Acceptance Testing

Our final design will be compared against the requirements document that we developed early in the semester. Weekly meetings with the client assert that the project continues to satisfy the non-functional requirements.

5.7 Results

The end goal of our project is to detect bugs in an SoC design. The bug detection tools we intend to create are based on the list of given bugs from the HACK@DAC 2018 competition, so a good measurement of our project's success would be what proportion of these bugs our tools can locate. Another strong metric of success is how effectively the tools assist in the exploitation of these bugs. If our tools give useful information that allows for exploiting many bugs, the quality of our project results will be very high.

6 Implementation

Our plan moving forward next semester:

- Stage 1 (~2 weeks): Finish implementing and testing the linter and signal tracer. Simulate the HACK@DAC 2018 competition RTL to test the system
- Stage 2 (~2 weeks): Test the signal tracer with mocking and/or against the 2018 SoC RTL. Implement a tool that detects missing default case statements
- Stage 3 (~4 weeks): Implement tools that detect:
    - Signals that don't change
    - Memory address overlaps
- Stage 4: (~2 weeks): Test the tools developed in stage three
- Stage 5: (~6 weeks): Continue developing and testing new bug detection tools to aid in detection/exploitation of bugs through Agile process

7 Professionalism

7.1 Project Specific Professional Responsibility Areas

The standard of professionalism that we have decided to use for our project follows the IEEE standard. The following is the explanation of how each area applies to our project.

Work Competence:

> The area of work competence is incredibly important to our project, in fact, it is arguably most important. The reason for this is because our project deals with the verification of system standards. If a system standard is violated, that is a bug that we need to find. We are tasked with designing software that detects these discrepancies within SoCs and enables the programmer to be more effective in his debugging practices. Consequently, the programmer needs to be able to trust our work. The code of our project must consistently perform to standard and enable the user to be more effective, without having to question our work.

Financial Responsibility:

> The area of financial responsibility is of little to no concern to our group. The project itself is nonprofit and there are currently no plans to market or sell the product. All members of the team are working for the sake of a job well done and not a payment and therefore there is little to no thought of monetary gain or transaction.

Communication Honesty:

> The area of communication honesty is fundamental to the progress of development. Within our project we are all learning about the HACK@DAC and the process of how to find bugs effectively. It is important that we honestly bring our skills to the table as well as our shortcomings. Our stakeholder is our faculty advisor and ultimately, one of the greatest assets we have for success. The more honest we are in communicating the progress we have made in the project, the more he can help us develop better solutions. Finally, it is also crucial that we as teammates are honest with each other. We each bring a unique set of skills and experiences to the table which, in collaboration, can develop more holistic solutions.

Health, Safety, Well-being:

> Health, Safety, and Well-being are important to the project as we develop solutions. As far as the project is concerned, there isn't much direct impact for users or developers in the realm of safety and health. However, it is of significance as we consider off-nominal scenarios in our requirements development and as we proceed in current development. We endeavor to consider the health of a team member and/or stakeholder to be a direct factor in the health of the team.

Property Ownership:

Ethics to do with Property Ownership is fundamental to the success of any software development project and therefore it is incredibly relevant to this project. As we uncover and develop our own solutions to current issues we need to be aware of where our ideas and solutions are coming from. We need to respect the rights individuals have towards their own ideas and rightly give them credit. Within our project, we rightly are sorting through many open source solutions but it is still very crucial to the success of the project that we maintain a proper respect and understanding of the use of these projects.

Sustainability:

N/A

Social Responsibility:

Social Responsibility is important to the project as it is designed to benefit society and improve the process of finding bugs within the SoC designs. Upon success, it will improve the workflow of SoC chip designers and HACK@DAC participants.

7.2 Most Applicable Professional Responsibility Area

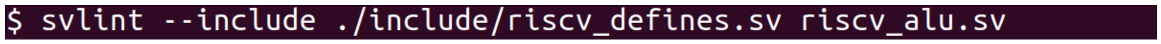
Within the project, the proficiency and the impact are in the realm of Work Competence. In the IEEE standards of ethics, section 5 says "To improve the understanding of technology; its appropriate application, and potential consequences" This pertains the work competence in that it exemplifies that working well is continuously learning about your project and knowing how it impacts other systems and individuals. This statement highlights our project, in that our goal is to know SoC designs, apply them appropriately, and display potential consequences of bugs.The goal of the project is to ultimately improve the work competence of the average HACK@DAC participant. Not only have ethics in work competence been a leading driver in the vision of the project but they also have been invaluable to the team. The team has displayed work competence in demonstrating understanding and aptitude in individual areas of expertise and this has contributed to quality mutual understanding and shared goals. These shared goals have been foundational in ensuring that work contributed has been timely and efficient.

8 Closing Material

8.1 Discussion

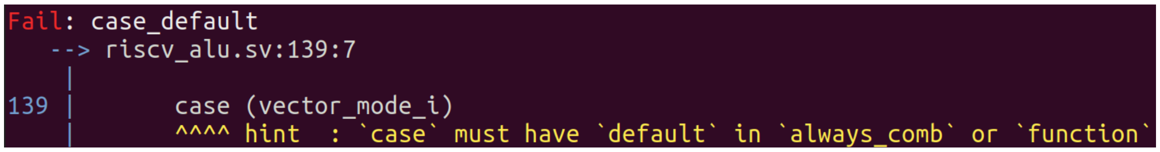
Below is a prototype of the process we plan on using to detect and exploit bugs:

1. Since we know there is an issue inside the file `riscv_alu.sv` in the 2018 HACK@DAC SoC, we can input that file to SVLint.



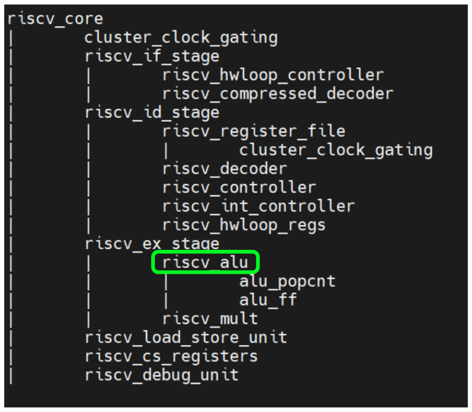
```
$ svlint --include ./include/riscv_defines.sv riscv_alu.sv
```

2. The output given by SVLint indicates that it can detect the bug we are looking for:



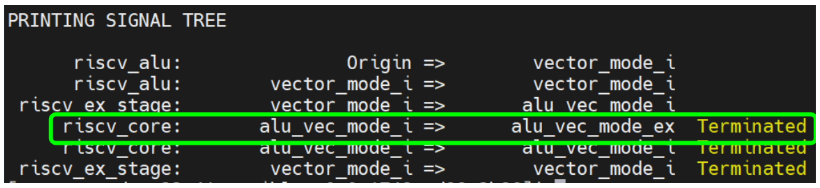
```
Fail: case_default
   --> riscv_alu.sv:139:7
    |
139 |       case (vector_mode_i)
    |       ^^^^ hint  : `case` must have `default` in `always_comb` or `function`
```

3. Using a prototype program we have written, we can build a dependency tree of the Verilog modules in the 2018 SoC design.



```
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
```

4. Then we can trace the signal `vector_mode_i` through the SoC design using the same program prototype:



```
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
```

5. Bug 14 in the Bug Classification list (see Appendix) can be activated by resetting the SoC, then immediately using an ABS instruction. The reset causes the value of `vector_mode_i` to be set to a predefined value of `VEC_MODE32`. The ABS instruction does not alter the value of `vector_mode_i`, so when the Verilog code for the ALU is executed, the ALU enters an undefined state.

8.2 Conclusion / Steps Taken So Far

Our work this semester started with attempting to properly scope our project. To begin, we classified the list of bugs provided by the 2018 HACK@DAC competition to assist with narrowing down which types of bugs our team could feasibly detect/exploit.

Next, we began researching and experimenting with different open source RTL tools that we could use as a part of our design. A Verilog linter we found called SVLint was particularly useful in finding a specific bug in the 2018 SoC, an undefined default case which causes unpredictable behavior in the SoC.

Our research of similar tools that currently exist led us to determine that we wanted our tools to be easy to install and use, open-source, and not require any compilation or simulation of the RTL. As discussed in section 4.1.3, the other tools we found lacked many or all of these properties. Our goal is to differentiate our toolset by adhering to these three properties.

During the process of exploring the exploitation of the bug that can be detected by SVLint, we determined that a program which can build a hierarchy of SoC designs and trace signals through them would be very useful. Therefore, we began prototyping a tool that could display signals and modules given a top level file of the SoC. Currently, this prototype gives a 90% accurate representation of the signal module tree.

We then identified that the signal tracing program would be an incredibly useful asset in detecting other bugs. Our current design plan involves utilizing the signal tracing program to detect bugs of a few particular categories.

Our plan of action moving forward is laid out in section 6. Throughout the project, inexperience among the team members with complex RTL implementations made it an uphill task to make progress early on in the design process. Our skills and knowledge will become more solid with each Agile sprint and with each iteration of our design.

8.3 References

"IEEE Code of Ethics." *IEEE*, Institute for Electrical and Electronics Engineers , https://www.ieee.org/about/corporate/governance/p7-8.html.

Authored, revised and maintained by many Googlers. *Styleguide*, Github, https://google.github.io/styleguide/shellguide.html.

"IEEE POLICIES." THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC, 30 Sept. 2021.

Seth-Lab-Tamu. "Seth-Lab-Tamu/HACKDAC-2018-Soc: Buggy SOC Used for the Second Phase of the HACK@DAC 2018 Hardware Security Competition." *GitHub*, HACK@DAC, https://github.com/seth-lab-tamu/hackdac-2018-soc.

"HACK@DAC 2021." *HacKEVENT*, HACK@DAC , https://hackatevent.org/hackdac21/.

8.4 Appendix

Bug Classification:

https://docs.google.com/spreadsheets/d/1fcpJHAmYbfWzrVyBpebDnIUiYFohVU-G/edit?usp=sharing&ouid=102151251018947909307&rtpof=true&sd=true

HACK@DAC 2021 rules:

https://drive.google.com/file/d/1CiYporz7Tc2UxBV5S-gwZ7GSXZSxm1AS/view?usp=sharing

SVLint:

https://github.com/dalance/svlint

Verible:

https://github.com/chipsalliance/verible