RISC-V SoC Hardware Vulnerability Detection Toolset

Final Report

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: 04/24/2022 v1

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, where participants compete 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 bugs found were released. These lists give a strong foundation for testing our tools to verify their effectiveness.

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 the vulnerabilities
- 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 executing 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 Revised Design Plan

1.1 Changes to Design Since Completing 491

See our 491 Design Document for our initial design.

Since completing our design document in the first semester of our project, our design plan was modified to better fit in with our constantly changing understanding of our project. Mainly, our team decided to decrease the number of tasks we would aim to complete in order to produce a smaller number of quality deliverables that can satisfy a majority of the project requirements (see 1.2 Requirements). Listed below are some of the major items which our team decided to remove from the design:

- Missing Default Case Bug Facets
  - In our initial plan, our team decided to try and detect/exploit three main classes of vulnerabilities: Signals that Never Change, Address Overlaps, and Missing Default Case Statements. Midway through the semester, our team determined that we would not have sufficient time to complete the required tasks for all three of these classes. Therefore, we ended our efforts to develop tools for Missing Default Case Statements in order to focus on the other two classes.

- 2018 HACK@DAC SoC Simulation / Exploitation
  - We had anticipated using the HACK@DAC SoC from 2018 because a list of its vulnerabilities are already available. Because of this, our initial goals revolved around detecting and exploiting bugs in the 2018 SoC. However, getting the SoC simulation up and running proved to be very difficult due to broken dependencies and missing documentation. After the simulation team spent several weeks on that SoC design without successful results, we decided to pivot our attention to detecting bugs on the already available and functioning SoC from the 2021 competition.

- Exploitation Framework
  - During our initial planning, the 2021 HACK@DAC competition was underway. This competition focused more on automated/programmatic exploitation of vulnerabilities in the SoC as opposed to previous competitions which concentrated on merely detection of bugs. For that reason, our team was initially planning on creating an exploitation framework which would streamline the exploitation of detected vulnerabilities. However, we later determined that the exploitation framework was out of the scope of our given project description. As such we chose to focus our efforts on other areas of the project.

Overall, our project's scope had to be reduced as the semester progressed. We underestimated the complexity and difficulty of the tasks needed to complete our initial plan. Although we did not accomplish everything we set out to achieve, we are pleased with the level of progress we have made during the semester.

1.2 Requirements

1.2.1 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 user gives an invalid signal name then the product shall display and error message
S3. 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. When the product accepts an SoC file, the product shall display line numbers of and descriptions of potential address overlap locations.
A3. If the product finds no overlaps, then the product shall exit with a success display message.

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. When the product accepts the root file and file or signal name, then the product shall display:
   a. The signal path
   b. Boolean of whether the signal has changed
US5. If abnormal behavior happens during execution within the product, the product shall display an error message

1.2.2 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

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)

1.2.3 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.

1.3 Engineering Standards

4 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.

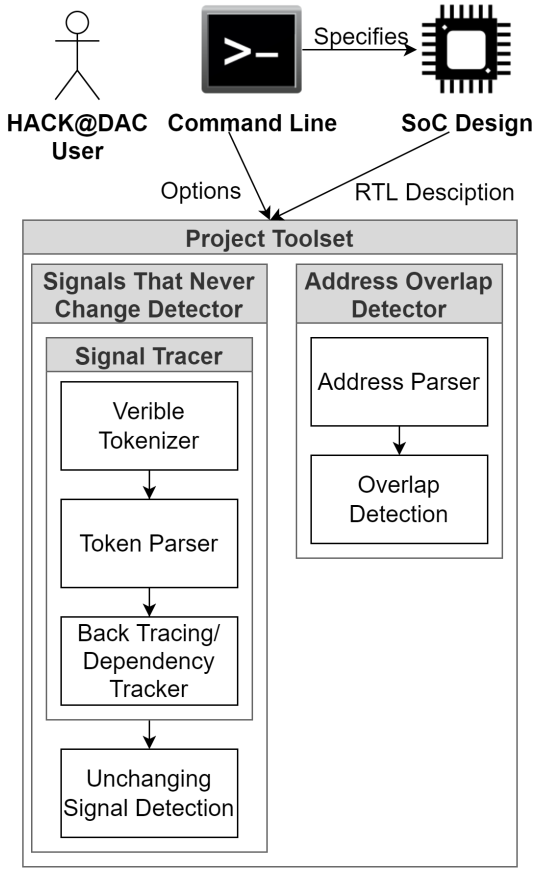
- 2.3.6.1. Google Shell Style Guide

1.4 Security Concerns/Countermeasures

One concern is that our tools could be used by a malicious actor. That is, someone could run our tools on an open source SoC and discover vulnerabilities that they could craft an exploit for. They could then use this exploit to steal sensitive information, or cause harm in other ways. This concern is mostly mitigated by the fact that for an SoC to be exploited in this way, it would have to be open source. Generally, with such products there is a lot of constructive development. So, there would already be a lot of effort by benevolent forces to try and find such vulnerabilities.

2 Implementation Details

2.1 Signal Tracer Tool



The backbone of our project is known as the Signal Tracer Tool. This tool operation is described below.

- The user specifies a directory of an SoC design, the top level module in that design, a signal they wish to trace and its associated module.
- An external library, Verible, tokenizes the Verilog/System Verilog files in the directory.
- The program builds a dependency tree structure of modules in the SoC design.
- The program then searches the tree structure for the module and signal specified by the user.
- The tool then traces the signal down the module tree finding any dependencies.
- Finally the tool outputs the signal's specific dependency tree.

The Signal Tracer Tool is an incredibly useful and versatile tool for programmatically searching through a hardware design. Using the Signal Tracer Tool allows a user to quickly and easily identify signal dependencies without having to manually search through files on their system

2.2 Signals That Never Change Detector

The first bug detection program in our project is the Signals That Never Change bug detection tool. This tool can detect non-changing signals within an SoC design and report their approximate location to the user. To do this, the tool makes heavy use of the Signal Tracer Tool.

The Signals That Never Change Detector tool functions as follows:

- The user supplies the tool with the directory of the SoC they wish to debug and the top level module name in the SoC.
- The Signal Tracer Tool is then used to identify the dependencies of each signal in the design.
- If a signal has no dependencies, has circular dependencies, or is always constant, it is flagged.
    - If a signal is flagged in this manner, we add it to a list of signals that should be manually observed by the user. These signals are not necessarily vulnerabilities however they are usually signals of interest while debugging an SoC.

2.2.1 Signals That Never Change Injector

In order to test the Signals that Never Change detection tool, our team designed a basic bug injector. This program assisted in automating the testing process to ensure the quality of the detection tool. The injection program functions as follows:

- The user runs the tool with a command line argument specifying the root directory of the SoC design.
- The tool selects a hardware description file from that directory, and selects a signal assignment within that file.
- The selected signal assignment is then modified so that the signal is unchanging. This is the injection of a Signals that Never Change bug. The location of this bug is then stored in a log file.
- To return the SoC to its original state (before any bugs were injected) the program just needs to be run again with a command line flag '-r'. The program will then remove each bug specified by the log file.

2.3 Memory Address Overlap Materials

Our group divided the Memory Address Overlap bug into multiple objectives.

These goals were:

- Locate the address files that determine the memory location of the cores
- Write a script to evaluate the address files such that it can correctly identify a misconfiguration in the code
- Find an exploit through simulation using the given kernel code and API

After reviewing the SoC source code for about two weeks, this team concluded that two files governed the addresses of all memory-mapped registers. The remaining work was divided between two teams.

The first team designed a script that can detect memory address overlaps. . The script was originally written in bash completely. However, given that no one was very fast at writing in bash and some things are easier done in Java. The script did some evaluation but the majority of the evaluation is done in Java. The tool can correctly identify core overlaps and units within the core having conflicting addresses.

The second team attempted to validate the script by exploiting a detectable memory address overlap to violate a security feature of the SoC. Since no memory address overlap bugs were reported in the 2021 SoC, and finding bugs in the SoC were out of scope for this project, we attempted to inject and detect our own vulnerability. Work in this area primarily focused on leaking a secret key from a trusted encryption engine by overlapping its memory addresses with an input to another encryption engine. Currently, there is no recorded indication that the files we found affect the simulation we're running, so we could not successfully validate our script. Future work in this area might include trying other exploits or synthesizing the SoC onto an FPGA.

3 Testing Process / Results

3.1 Signals that never change & Signal Tracer Testing

In order to test the Signals That Never Change bug detector, we developed a bug injecting program that can inject unchanging signal bugs into a Verilog/System Verilog design. For the purposes of testing the Signals That Never Change Detector, a minimum viable hardware design was used to decrease the amount of time needed for each test.

The testing process is described below:

- The Signals That Never Change Injector is used to inject an unchanging signal bug into a random position of the hardware design. The log file generated by the injector specifies the location of the injected bug.
- The Signals That Never Change Detector program is run. It will then output where it detects unchanging signal bugs (if it finds any).
    - For a successful test, the Signals That Never Change Detector will be able to determine which signal is associated with the injected bug. The output of the detector will correspond with the injection sight specified by the log file created by the injector.
    - A failed test occurs when the detection tool does not accurately detect the bug injected by the injection tool.
- The results (success/failure) are recorded.
- The injection removal feature of the Signals That Never Change Injector is used to return the hardware design to its original state before the injection.

The above procedure was used on a limited-scope hardware design. The Signals That Never Change Detector was able to correctly detect over 60% of the injected bugs. In the planning phase of our project, our goal for this tool was to achieve 50% accuracy, so we are satisfied with the results. However there is certainly room for improvement.

A few test trials resulted in failure, not due to the fault of the detection tool, rather to the limited nature of the bug injector. The output of the detector contains several false positives, however these would be largely mitigated by running the tool on a full SoC design. However, running the tool on a full SoC design requires a large amount of computation time, so for testing purposes a smaller hardware design was used.

3.2 Memory Address overlap testing

To test and validate the code within the memory address overlap tool we used: Junit testing, test files, and full system validation.

- The bash script that evaluates the address file was tested through test files. The individual components of the program would be run on test files to validate correct operations.The full script was then tested on these files in sequence to validate the functionality.
- The Overlap Detector was tested through the use of Junit and more test files. The individual functions are tested through mock data and asserted that expected output matched the test output. This Detector was then also run one 4 separate mock data files to validate the functionality of the code.
- Finally, the full system was tested on the SoC design after mock overlap bugs had been manually injected. The Overlap Detector could find 90% of the bugs consistently. The bugs that the tool tended to miss were the ones that involved extreme offset values but the correct base.

4 Placing your Work in the Context of Related Products

The problem of finding problems in code is by no means a new one, though one thing our project does differently is doing so in RTL hardware description code specifically. For the problem of software, several solutions exist. One such solution is Amazon CodeGuru, which uses machine learning to identify potential security vulnerabilities in software and patch them. Amazon CodeGuru also provides profiling tools for runtime analysis, but its main relation to our project is in the code analysis. While its design may be similar, in that many software vulnerabilities such as buffer overflows fall into general categories which can be identified by looking for patterns, the application of machine learning into that problem means that false positives can be reduced and more complex problems can be identified. However, one potential issue with taking this approach with regards to our problem is that a dataset of buggy software is much easier to come by than a large dataset of buggy RTL, due to much more software being produced than hardware. However, our Signals that Never Change Injection tool may be useful for generating such a dataset.

Another program which is similar to our work is GitGuardian, which is intended to scan repositories for secrets such as encryption keys or passwords. While searching for such vulnerabilities isn't a problem which requires the same level of knowledge about a codebase as finding RTL bugs does, it is more similar to our project than CodeGuru is in that it uses static tools which use properties about the vulnerabilities it catches to have hardcoded detection and solutions (rather than using datasets and machine learning).

Another approach which looks at security questions regarding hardware is Verismith, a research program which generates large, valid RTL code and tries to find bugs. However, it looks for bugs in the behavior of these programs when run on FPGAs, as opposed to bugs in the verilog code itself (which, for automatically-generated programs, have no real intended behavior to check other than their behavior in simulators). However, it is addressing the same fundamental problem of trust when it comes to hardware security and how difficult and important it is to verify that trust is not misplaced.

The application which is probably most similar to our problem, however, is that of a modern IDE which has features for finding where variables are defined/set (analogous to the signal tracer component) as well as features for finding compiler errors and compiler warnings and displaying these to the user. In order to do this, it needs to be able to build a map of the codebase and be able to understand properties of different variables and how they interact with each other. The biggest difference, then, is that IDEs only give suggestions for compiler errors or general program errors, while our tools are specifically geared towards identifying more subtle bugs which can cause security problems.

Appendices

Appendix I - Operation Manual

Signal Tracer & Signals That Never Change Bug Detector

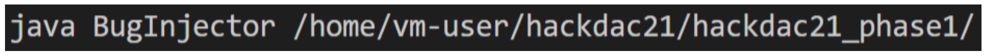
The Signal Tracer and Signals That Never Change Bug Detector require Java, Linux, and Verible (An open source verilog tokenizer) in order to run.

1. The detector requires multiple command line arguments to be specified in order to run.
    a. Run the program with the -help tag to see all command line arguments.
    b. Needed arguments are -top and -dir
2. The program currently runs on our VM by running the ./build command script. The necessary files can be found in our git repository.
    a. This command will pull the latest version from git, compile and run the program with the specified arguments
    b. If the -wmap tag is used, a file map will be generated as a .txt document that acts as a temporary database to improve parsing time.
    c. If the -map tag is used, the database map will be read from and initial parsing will not occur.
3. After running the command, the detector will generate a list of possible problematic signals

Signals That Never Change Bug Injector

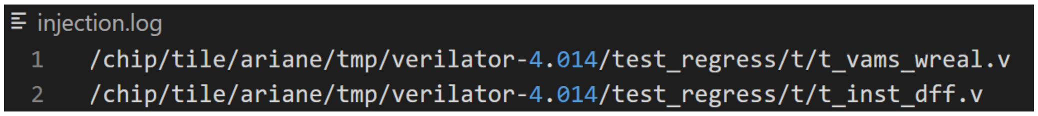
The Signals That Never Change Bug Injector requires Java to run. Be sure to have Java installed on your computer.

1. Check out the necessary files from: https://git.ece.iastate.edu/sd/sdmay22-41/-/tree/master/Signals_That_Never_Change_Injector
2. Build and run the program with a command line argument specifying the root directory of the SoC you wish to inject a bug into. Note, the directory should contain either Verilog (.v) files or System Verilog (.sv) files.



```
java BugInjector /home/vm-user/hackdac21/hackdac21_phase1/
```

3. A file called `injection.log` will be updated with all of the files that have had a bug injected into them.



injection.log

```
/chip/tile/ariane/tmp/verilator-4.014/test_regress/t/t_vams_wreal.v
/chip/tile/ariane/tmp/verilator-4.014/test_regress/t/t_inst_dff.v
```

4. Run the program with the command line flag `-r` to remove injected bugs from the files listed in `injection.log`. Each entry in the log file will be cleared after the bug has been removed.

Address Overlap Tools

1. Clone the necessary files from https://git.ece.iastate.edu/sd/sdmay22-41/-/tree/master/Memory_Address_Overlap onto your VM with the SoC design
2. Go through the bash script and compete the TODOs to ensure correct configuration for your SoC design
3. Download a java JDK to your VM (make sure it is no older than version 11)
4. (Optional) Make the bash script an environment variable
5. ./evaluateAddresses.bash or bash evaluateAddresses.bash
6. Interpreting the output:
    a. The output will be empty if there are no bugs detected. However, if there are issues within the SoC the output will display what core base addresses have conflicting address spaces and addtionally it will display what address locations cannot be resolved because of the issue.
    b. There is a file created by the program called addressesStartAndBase that will give you a filtered view of all of the addresses, their starting locations and the associated base.

Appendix II - Alternative Design Versions

Versions before clients specs changed

The original design for the project was a tool set to identify bugs in the SoCs. Midway through the first semester we received the documentation for the 2021 HACK@DAC competition. The point system had changed in that it also awarded points for exploiting a bug as well as finding it. Our advisor encouraged us to consider this development in planning our design. This changed the direction of our project in that we wanted to find exploits and an exploitation framework for each of the bugs. This expectation changed again during the second semester when we realized the work required behind each requirement. This drove us to evaluate our goals and it was decided that the exploitation framework was out of scope for our initial goals.

Versions before learning more about the project

Before learning about the project, we planned to explore every single bug, identify it, replicate it, and exploit it. In the initial design conversations, not many members of the team were very familiar with SoC development, and this seemed like the most logical course of action. The plan was that after we had accomplished these goals we would put together multiple static command line tools and potentially fuzzers, to identify the bugs on the SoC. This was great brainstorming but incredibly idealistic. We underestimated the amount of learning was needed for each team member before any work could be started. We underestimated the complexity of SoCs and the sheer volume of code we'd be sifting through to find the well crafted bugs. The simulation of the older SoC design turned out to be infeasible due to deprecated libraries. Exploiting the found bugs in the older design proved to be a pointless and impossible task. As the end of the first semester completed, our team had narrowed the tool set to a few select static command line tools to identify a select number of bugs. This scope changed again throughout the second semester as we more fully grasped the problems laid before us. We realized that for feasibility, our efforts were best spent on bug identification and proof of concept. In addition to writing the static toolset we designed a bug injector for testing. We also spent many hours on simulation to prove the exploitability of the bugs we can identify. This is valuable for knowledge transfer and the testing of the quality of our work.

Appendix III - Other Considerations

For our team, this project included extra challenges beyond the creation and implementation of our design. Our project involved simplifying the incredible complexity of hardware design and security, a task that occupies the time of teams of full time professionals . Prior to this semester a majority of our team members had little to no experience with SoCs or other hardware designs with similar levels of intricacy. In addition to this, our project's problem statement started as open-ended and less defined than other projects.

A large part of our time spent on the project was dedicated to learning about hardware design/security and scoping out our final vision for the project. From the beginning of the semester we heavily underestimated the difficulty of the project and how long particular tasks would take. If we were to start over with our current level of knowledge, we would have a much firmer foundation for our project to be built on.