SAND24XX-XXXXR
LDRD PROJECT NUMBER: 235369
LDRD PROJECT TITLE: Hardware Fuzzing with An Emulator
PROJECT TEAM MEMBERS: Michael Haselman, Brett Weyer, Nancy Lau.

ABSTRACT:

Bugs in digital logic have led to some significant security vulnerabilities. Hardware bugs are particularly troublesome since they cannot be easily patched. Additionally, if the bug is in the root of trust, all trust built upon it can be vulnerable. Traditional testing either require a deep knowledge of the system, creative attack vectors and lots of human interaction. This is not scalable as there are very few engineers that can wear the hat of a designer, a verification engineer, and a cybersecurity expert. Hardware fuzzing is a relatively new research area in dynamic hardware testing. It has proven to be an effective method for discovering bugs, unexpected behaviors, and security vulnerabilities in software. While hardware fuzzing is new to the hardware domain, it has a strong track record in software testing. Fuzzing is a testing technique that randomly mutates the input data to uncover bugs or vulnerabilities in the design. It is especially good at finding corner cases that test engineers can not envision. Another advantage over other dynamic testing techniques is that, if done well, deep knowledge of the design is not required. Additionally, fuzzing scales well. If the system is set up correctly, it can run unsupervised for weeks if necessary. In this work, we propose using hardware fuzzing to improve the input vector generation for an information flow tracking tool. To get reasonable throughput of test vectors, an emulator is targeted as the execution platform. Efficient emulator execution has some specific requirements.

INTRODUCTION AND EXECUTIVE SUMMARY OF RESULTS:

This seedling exploration was funded under the Digital Assurance for High Consequence Systems Mission Campaign (DAHCS MC) as a late start LDRD, running for approximately five months prior to the first fully funded year of the DAHCS MC. The DAHCS MC is a 7.5-year research portfolio created to develop the scientific foundation needed for rigorous, rapid, cost-effective, generalizable digital assurance[1] across high consequence systems[2]' (HCS') lifecycles. To create this foundation, the DAHCS MC aims to develop capabilities needed to understand systems-level implications of trade-offs against digital risk in high-consequence cyber-physical systems – across lifecycles and across digital abstraction levels – delivering a process and ecosystem to obtain empirical, ideally measurable, digital assurance of HCS, and ultimately enabling informed acceptance of digital risk. The DAHCS FY24 Late Start Call document [8]

[1] *Digital assurance* includes processes, measures, and/or controls applied to digital technologies to make sure that a system fulfills its intended purpose, even in the active adversarial cyber environment of today.

[2] *High consequence systems (HCS)* serve a very specific purpose, or *mission*, where failure to meet the mission can result in unacceptable consequences.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

describes these definitions, goals, research roadmap, and other technical research context for this seedling LDRD in significantly more detail.

This late start LDRD addresses the Scalable Analysis research need identified in the DAHCS MC 2024 Late Start Call. This work attempts to develop a scalable solution to test vector generation for verification of digital circuits. The research and development (R&D) effort focused on improving the tools for controlling the digital risk in digital components of high consequence cyber-physical system.

Bugs in digital logic have led to some significant security vulnerabilities. Hardware bugs are particularly troublesome since they cannot be easily patched. Additionally, if the bug is in the root of trust, all trust built upon it can be vulnerable. Traditional testing either require a deep knowledge of the system, creative attack vectors and lots of human interaction. This is not scalable as there are very few engineers that can wear the hat of a designer, a verification engineer, and a cybersecurity expert. Additionally, each solution will be bespoke. Hardware fuzzing is a relatively new research area in dynamic hardware testing. It has proven to be an effective method for discovering bugs, unexpected behaviors, and security vulnerabilities. While hardware fuzzing is new to the hardware domain, it has a strong track record in software testing. It was even used to discover the Heartbleed bugs. Fuzzing is a testing technique that randomly mutates the input data to uncover bugs or vulnerabilities in the design. It is especially good at finding corner cases that test engineers can not envision. Another advantage over other dynamic testing techniques is that, if done well, deep knowledge of the design is not required. Additionally, fuzzing scales well. If the system is set up correctly, it can run unsupervised for weeks if necessary.

In this work we explore the use of fuzzing to improve the input test vector generation for information flow tracking. We modify an open source fuzzer (AFL++) to generate inputs for an open-source AES core. While we did not fully get the pipeline implemented, we developed enough to see the benefit and develop a path forward.

DETAILED DESCRIPTION OF RESEARCH AND DEVELOPMENT AND METHODOLOGY:

Fuzzing

Fuzzing is a testing technique that randomly mutates the input data to uncover bugs or vulnerabilities in the design. It is especially good at finding corner cases that test engineers can not envision. Another advantage over other dynamic testing techniques is that, if done well, deep knowledge of the design is not required. Additionally, fuzzing scales well. If the system is set up correctly, it can run unsupervised for weeks if necessary. Fuzzing was originally developed to test software. As shown in Figure 1, fuzzing has three main components, initial

input, the fuzzing software, and an instrumented executable. The initial input is generally a known good set of input test vectors such as images or network packets. These are fed into the executable for the initial test. The executable is compiled with the fuzzer to add coverage hooks. If the operating system detects a failure of the software, the fuzzer ceases operation. Otherwise, the initial input is randomly mutated. The mutated input is sent into the executable for testing. Assuming no failure, the fuzzer checks to see if the mutation increased the coverage of the code. Coverage in software is most commonly branch coverage. If it did, the mutation is kept and fed into the next mutation cycle. Otherwise, it is dropped, and a new mutation is attempted. This continues until a failure occurs, the coverage is deemed adequate, or a set testing time has expired. Software fuzzing has uncovered several software vulnerabilities including Heartbleed and Shellshock.

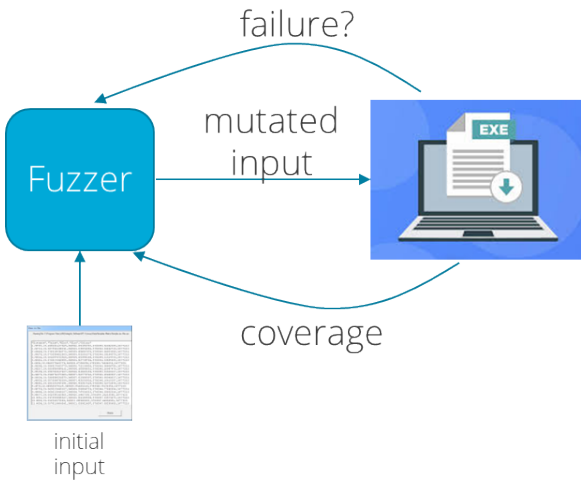

Figure 1. General structure of fuzzing.

AFL

AFLplusplus (AFL++) [9], the successor of American Fuzzy Lop (AFL), is a software fuzzer that is traditionally used to fuzz software binaries. AFL++ utilizes coverage-based fuzzing to monitor which areas a fuzzed binary is executing. This information allows the tracking of how inputs lead to specific binary behavior and facilitates the creation of a corpus—a database of inputs that cover the entire codebase. Although AFL++ can be used to in both black-box and white-box fuzzing applications, our discussion of AFL++ will focus on its ability achieve comprehensive codebase coverage.

In AFL, iteration starts with initial manually generated seed inputs that are subsequently mutated. If those mutated inputs result in unusual changes in coverage, they are deemed interesting and stored into the corpus for further mutation. This process allows the AFL++ fuzzer to continually explore new code paths with feedback, thereby enhancing the likelihood of discovering unknown bugs and vulnerabilities.

Hardware Fuzzing

Hardware fuzzing is a relatively new research effort to apply software fuzzing methods to the verification of register transfer logic (RTL) such as Verilog or VHDL. There are some challenges unique to hardware fuzzing though.

Like software fuzzing, there are three main components of hardware fuzzing.

1. A way to generate meaningful and valid inputs. For a complex system that requires a series of inputs to initiate system or operate a bus for example, the input generator needs to adhere to a certain set of inputs for operation. Truly random inputs will mostly lead to illegal inputs that don't increase the overage. Inversely, if there is no randomness and no deviation from the "protocol" then you might not find a certain set of bugs/vulnerabilities.
2. A way to monitor coverage. Coverage must be defined. Research has used bits toggle [6], mux coverage [3] and control registers [3] as the entities to monitor. The coverage should feed back into the next round of input mutations to make it targeted. If a mutation to the input increases the coverage, it is kept. If not, the mutation is reversed from the input set.
3. A way to indicate failure. For software this is generally a crash, but in hardware this can be difficult to detect. Hardware doesn't crash, but it "fails" in many ways. It can lock up, output incorrect results, leak secret data or error in many other undesirable ways. There have been a couple of ways to monitor for "failures." The first method is to use assertions. These are used in many methods of digital verification. The other, and more preferred method is to compare the test results to a golden reference model. This is called differential fuzzing. Often this is an ISA emulator that was built to verify the processor design. Another method to get a golden reference is to use a formally verified state machine. Both methods can be used. If the design engineer put assertions in the design during the implementation, these can be used to detect a certain set of know failures while the golden model can catch the corner cases. A final method is to use an information flow tracking tool such as Cycuity's Radix [2] detect information leakage.

One challenge in hardware fuzzing that doesn't exist in software fuzzing is interfacing software to hardware. Fuzzers are currently written in software and fuzzing techniques have become incredibly advanced in the software verification domain. They support many coverage metrics, and they are highly efficient at testing software implementations. With a lack of fuzzer implementations in the hardware space, we needed to rely on works previously developed for the software world. This allowed us to take advantage of the long history of fuzzing research and optimizations, but it increased the complexity of the testing process—testing no longer remained solely in the hardware simulation domain. Using an existing software based fuzzer requires a way to move date between the software and RTL simulator or emulation. This is generally accomplished using the direct programming interface (DPI) available in SystemVerilog. The SystemVerilog DPI is basically an interface between SystemVerilog and a foreign programming language, in particular the C language. It allows the designer to easily call C functions from SystemVerilog and to export SystemVerilog functions, so that they can be called from C.

There are three ways researchers have proposed to fuzz the RTL.

1. Convert the RTL to software using something like Verilator [7]. This has limitations on the type of RTL that can be converted (i.e., only synthesizable). This means that assertions traditional RTL testbenches cannot be used. The upside of converting the RTL to software is that it can then be driven with software. This is helpful as it is much easier to code the mutation, and coverage logic in software. Additionally, it runs faster than traditional RTL simulation.
2. Run in a simulator [3]. The benefit of using a simulator such as Modelsim is that this toolchain is very familiar to RTL designers. It also supports assertions and complex test infrastructure, even if it isn't synthesizable. The downside is that it runs many orders of magnitudes slower than hardware.
3. Run in prototyping hardware such as a FPGA [10]. This has the benefit of faster execution. Unfortunately, the limitations make it not feasible for fuzzing. The lack of visibility makes calculating coverage impossible and it is difficult to detect failures at a fine grain.

We propose a solution that takes advantage of both the software and the simulation solution. We have access to a Siemens Strato Veloce Emulator. This computer can accelerate RTL simulations up to 1000x. It also supports all the advanced verification techniques such as assertions. To support software input we will use some previous work our group did with the direct programming interface (DPI). DPI allows C programs to control SystemVerilog testbenches.

Information Flow Tracking

Information flow tracking (IFT) [1] is a dynamic RTL verification technique that is especially well suited at detecting confidentiality and integrity failures in RTL digital designs. IFT tools are designed to track data as it flows around a design. It achieves this by tagging instances of data and tracks where it flows throughout execution. For security use cases, the data is generally a secure asset such as an encryption key. During execution, if the "tainted" data ever reaches a location that has been deemed insecure, a failure results. The taint is preserved through any transformation of the data, so even if the secure data is combined with other data, the result is also tainted. This guards against simple reverse engineering of the secret data. The tool also allows for exceptions to the rules. For example, during encryption operations, the key will taint the resulting cyphertext. Since it is known that the encryption operation is secure, this path can be excluded. The leading commercial IFT is a tool called Radix from Cycuity.

Figure 2 shows and example of a scenario where IFT can be used to secure a design. In this design, there is a key that should be kept secret. For confidentiality, an external attacker should not be able to read the key in debug mode. Additionally, for integrity, the attacker should also not be able to overwrite or modify the key.

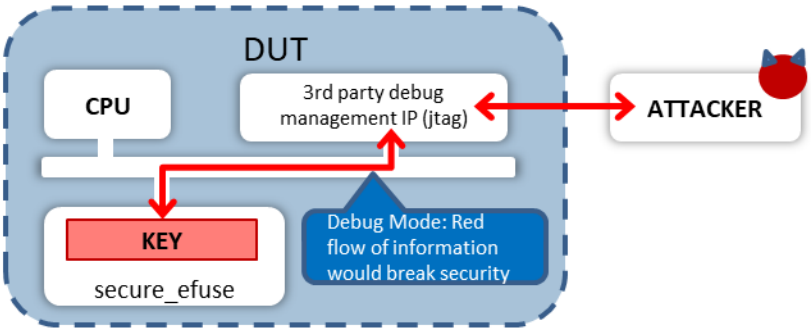

Figure 2. Example of IFT tracking a security key to verify it can not be accesses by an external attacker.

The steps to use the Radix tool is outlined below.

1. Define the security requirements.
   - Secure Asset – efuse key
   - Objective – no access allowed
   - Attack Surface/Boundary – ports on device (JTAG, UART, SPI, etc)
   - Conditions where security policy would be relevant – debug mode
2. Write out the security requirements.
   - "The efuse key cannot be accessed via JTAG when the DUT (device under test) is in debug mode"
3. Write down Radix rules in the rule syntax.
   - Confidentiality: dut.secure_efuse.key =/=> dut.jtag.$all_outputs unless(dut.debug_mode == 0)
   - Integrity: dut.jtag.$all_inputs when (dut.debug_mode == 1) =/=> dut.secure_efuse.key

Note that the only required step in this flow is to write the Radix rules. The preceding steps are done to help define the security rules and ideally are a part of the device security plans or requirements. The confidentiality rule in this test checks that the key that is in the secure_efuse module shall not flow to any of the outputs in the jtag module unless debug_mode equals "0". The integrity rule can be interpreted in a similar manner.

After the security rules are written, they are compiled into the design as a security monitor. The security monitor tracks the simulation or emulation and signifies a pass/fail of each rule. It also shows how the violation occurred via a graphical trace of the taint.

Like all dynamic RTL verification tools, IFT performs only as well as the input test vectors it is supplied with. This makes it a good candidate to apply fuzzing to. Additionally, the IFT tool defines the failure modes in the security rules and coverage is easier to define then in generic

RTL verification. For these reasons, we chose to target IFT for our initial hardware fuzzing effort.

Coverage in IFT is a much smaller problem to solve then in general RTL verification. The only signals that are of concern in the design are signal that the secure asset can possibly connect to. Of course, signals that can be read by the processor can in theory connect to most of the rest of the design, but in practice secure assets, such as encryption keys, are restricted to isolated areas for security reasons. Additionally, there is generally a security boundary that enforces the security of the asset. This can be the output of a module or a multiplexor. The security boundary is the coverage that matters in IFT.

The graphic in Figure 3 illustrates how coverage is calculated in IFT. The RTL can be statically analyzed to identify all the signals the source can "taint" (shown in purple) on the way to the destination. The vertical line indicates the security boundary that is designed to be the boundary of where the secret is controlled. The nodules on the boundary line indicate the possible connections at the boundary. Therefore, good coverage would be when all the nodes are tainted. It is also possible that some of the nodes cannot be tainted (yellow area) because flow control does not allow the source to travel there. In this illustration, it would be a failure for the "tainted" data to reach the purple area.

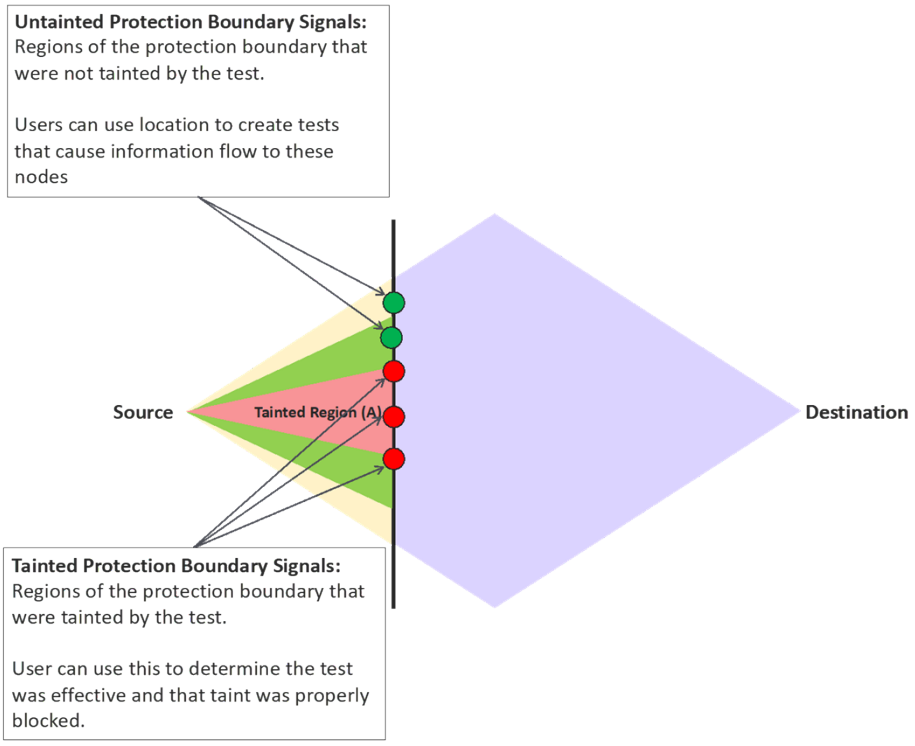

Figure 3. Coverage diagram for IFT.

Figure 4 shows an example of the RTL and schematic of a security boundary. In this example, the case statement is turned into a multiplexor. The signal data_in_f1 is the secure asset in this case. The multiplexor mux_2 is where the secure mode is controlled but all the data is controlled at the wide_select_3_6 multiplexor so this is where secure boundary is set, and the coverage is monitored. The secure mode signal should also added to the coverage metrics since that is what controls the secure mode.

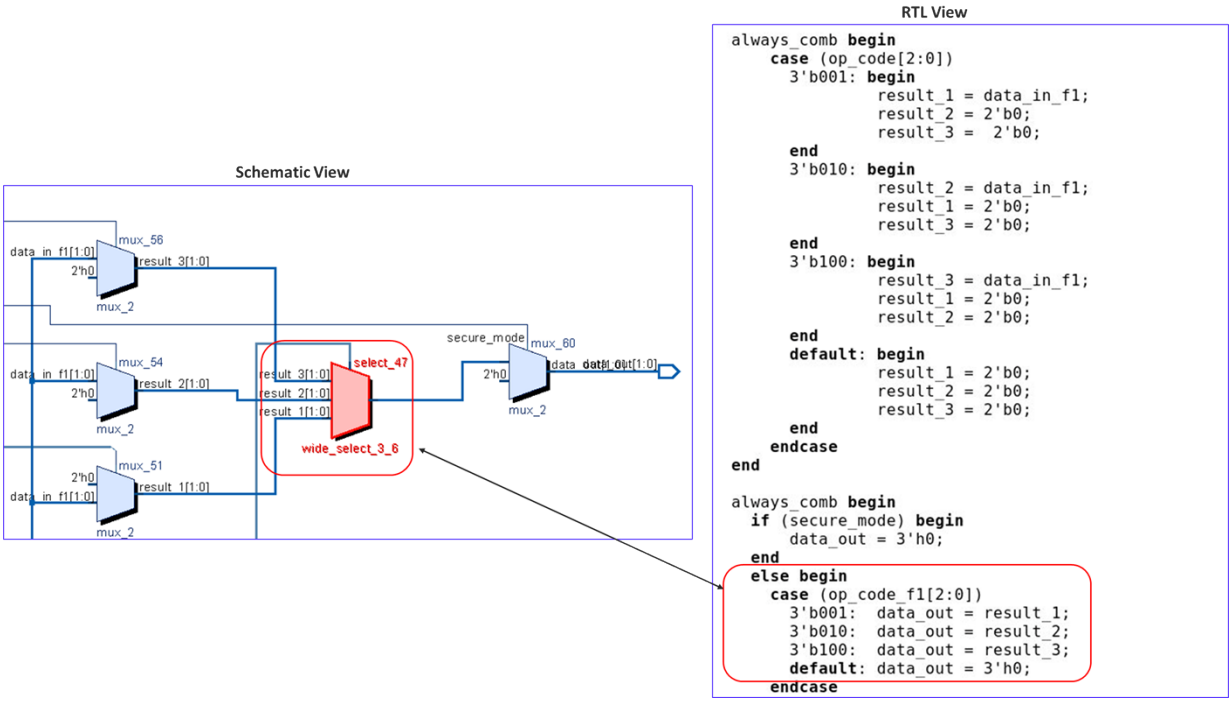

Figure 4. Schematic and RTL view of a boundary. In this circuit, the wide_select_3_6 multiplexor is the security boundary. Note that mux_2 controls secure mode, but the data arrives via wide_select_3_6 multiplexor.

Our approach

There are four main components to the developed fuzzing process:

1. The AFL++ fuzzer process mutates inputs and executes the target program.
2. The AFL++ target program takes inputs from fuzzer process, drives HW simulation, and passes results back to fuzzer process.
3. The hardware simulation takes inputs from target program and executes the DUT.
4. The simulation tracks coverage and violations in the DUT and sends them back to the target program.

The AFL++ fuzzer process will exist for the lifetime of the testing procedure. This is what uses the coverage and violation feedback to make an informed decision on which inputs made

meaningful progress in verifying the design under test. Additionally, this fuzzer process is also responsible for mutating the inputs to be passed to the hardware simulation.

In the (simplified) process of normal AFL++ software verification:

1. The fuzzer process mutates the inputs.
2. The fuzzer process passes the inputs to the target program.
3. The target program is executed with the inputs and tracks coverage.
4. The target program terminates and reports the results back to the fuzzer process.
5. Jump to step 1.

This cycle repeats for the duration of the testing procedure, and each time the target program is executed, a new process is created. This means the target program has a limited lifetime, unlike the AFL++ fuzzer process and hardware simulation. To facilitate cooperation between the hardware simulation and software fuzzer, we needed to develop a new fuzzing protocol which synchronized the interactions and data transfer between the fuzzer process and the hardware simulation.

To integrate a software fuzzer with the hardware simulation process, we needed to integrate a method to measure coverage and security violations and develop an interface where the simulation and fuzzer could repeatably (in a loop) communicate inputs, coverage and violation feedback, and functional results. Additionally, this all needed to be done in a synchronized manner to avoid mangling data due to race conditions between the independent hardware simulation process and the fuzzer process.

It is important to note that in our implementation of this hardware fuzzer, the persistent AFL++ fuzzer process does *not* communicate directly with our hardware simulation. That is the responsibility of the AFL++ target program. AFL++ requires the fuzzer target to be a binary which has been instrumented for tracking coverage. This is something that cannot be done in the hardware simulation, so we needed to create a small C program—our AFL++ target—to act as a bridge between the AFL++ fuzzer process and the hardware simulation.

The solution we developed is depicted in Figure 5.

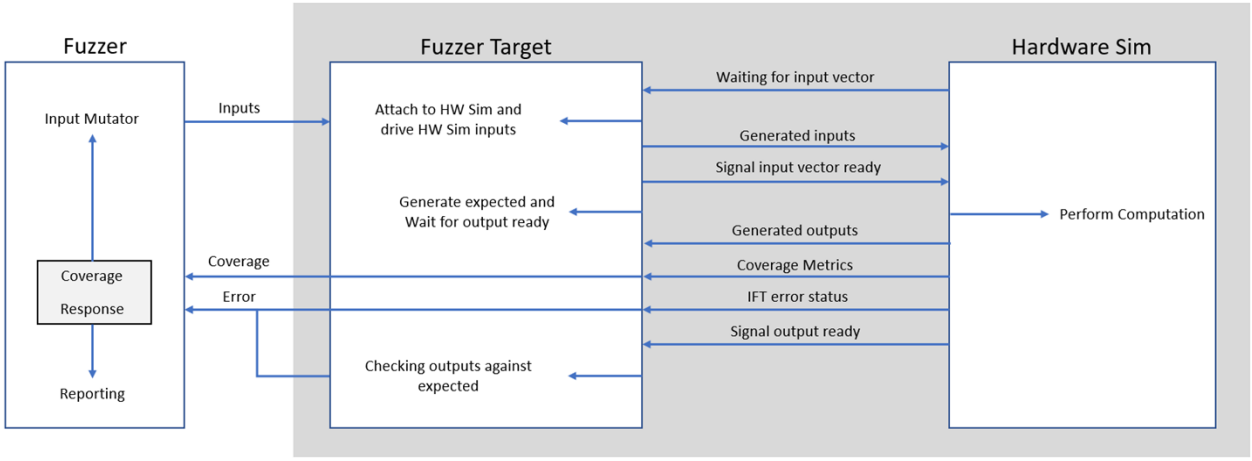

Figure 5. Architecture of our hardware fuzzing pipeline.

In the (simplified) process for using AFL++ to verify hardware:

1. Hardware simulation begins with the IFT software attached.
2. Hardware simulation initializes the C environment.
3. Hardware simulation waits for input.
4. AFL++ fuzzer process begins.
5. Fuzzer process mutates the inputs.
6. Fuzzer process forks the target program and passes in generated inputs.
7. Target program attaches to the hardware simulation through DPI interface.
8. Target program drives the inputs to the hardware simulation.
9. Target program computes expected functional output while waiting for response from hardware simulation.
10. Hardware simulation passes generated outputs, coverage metrics, and violation status to the target program.
11. Target program passes coverage metrics to the fuzzer process.
12. Target program compares the generated outputs to the expected.
13. Target program reports error/violation status to the fuzzer process.
14. If fuzzing not complete, jump to step 5, else continue to 15.
15. Hardware simulation de-initializes C environment.
16. Hardware simulation terminates.

Steps 5 to 13 repeat in a cycle until the testing procedure is complete.

The entire time this is running, the Radix tooling (our IFT software) is diligently tracking the hardware simulation and generating the coverage metrics and security property violation status. The hardware simulation then reports this information to the fuzzer target and it is passed through back to the fuzzer process. The IFT tooling only interacts with the hardware simulation,

then the hardware simulation is responsible for reporting the coverage results back to the fuzzer. The Radix tool does not have any direct interaction with the fuzzer process or fuzzer target.

To start the fuzzing process, the hardware simulation, through DPI calls, initializes the C runtime environment. It creates a shared memory region, which the fuzzer target can attach to in order to transfer data between the simulation and the fuzzer target in a synchronized manner. The shared memory region holds mutexes, condition variables, status codes, and data structures that accommodate the transfer of data to and from the device under test. It is important the hardware simulation be the maintainer of this C runtime environment, because the fuzzer target has a limited lifetime.

After the C runtime is initialized, the SystemVerilog testbench, in a loop, performs the following actions:

1. Retrieve test inputs from shared memory.
2. Perform the test.
3. Write results into shared memory.
4. Signal to the target program that computation is complete, and results are available.

The target program is responsible for passing generated inputs from the fuzzer process to the simulation. In the current state of the project, the target program is responsible for generating the test inputs which it then passes to the simulation. This will be the case until the fuzzer is fully integrated. When executed, the target program obtains the mutex for the shared memory, randomly generates test data and maps that data into the shared memory region, then signals that data has been committed. To retrieve the test inputs, the SystemVerilog testbench obtains the mutex for the shared memory, then conditionally waits (releasing the mutex) until the target program signals that data has been committed to the shared memory region. It is important that the simulation is started before the fuzzer to mitigate an initial race condition here.

At this point, the simulation is executing and tracking information flows with the provided inputs. At the same time, the target program is responsible for generating expected results. In the example scenario we have developed, these expected results are obtained from a C implementation of the AES algorithm. These expected results should only pertain to the functional performance of the design.

Once the simulation is complete, the simulation obtains the mutex for the shared memory region, writes the computed results into the shared memory, and signals the target program that processing is complete. The target program conditionally waits for the processing complete signal and reads the results of the simulation process (functional, IFT violation status, and coverage results). The target program then compares the expected and actual functional results and notifies the fuzzer of the status of this comparison. The target program also passes along the IFT violation status and coverage results to the fuzzer.

The fuzzer; having access to the functional, coverage, and IFT violation status results; takes the information it receives and 1) updates its global coverage metrics, 2) reports any IFT violations or functional errors to the designer, and 3) computes the next mutation for the fuzzer to pass to the target program so that the simulation process may continue.

RESULTS AND DISCUSSION:

The current implementation of our hardware fuzzing approach has achieved a significant portion of the project targets. We have demonstrated that the hardware simulation process, using commercially available tools, can be designed an instrumented in such a way that 1) we can have a software-based program driving the inputs to the simulation, 2) the simulation can be implemented to have repeating, synchronized interaction with the software program (a requirement for fuzzing), 3) information flow tracking tools can serve as an alternative for an Operating System when it comes to capturing "violations" in a design, and 4) using coverage-based fuzzers is a viable alternative to existing constrained random verification techniques. This last point is assessable from a cumulation of points 1-3.

The code base developed now provides all the necessary interfaces for integration with a software fuzzing tool. The target program successfully drives the simulation, and the simulation successfully runs and reports back to the target program the functional results, violation status, and coverage metrics. These are the only pieces of information which are needed to integrate our system with a software fuzzing platform. That integration, however, is still a work in progress. The AFL++ platform has been developed over a period of nearly 10 years, first as an open-source project backed by Google (then called AFL), and now by a dedicated group of researchers who continue to add and update features as their own research projects require. It is a highly complex set of code which will require more effort to fully integrate with our hardware-focused platform.

Most of the difficulty surrounding the integration of AFL++ is due to its software-focused design when operating on the existing coverage metrics reporting system. AFL++ has three major components: 1) the fuzzer platform, 2) the forkserver, and 3) the instrumented compiler. It is points 2 and 3 which complicate our integration. When utilizing AFL++ to aide in the verification of a software program, the first step is to compile the target software program with AFL++'s modified gcc compiler—afl-gcc or afl-g++ for the simple usage of AFL++. This compiler is responsible for integrating the coverage metric system into the compilation process. Each basic block in the program (a set of sequential instructions no jump/branch/call instruction goes into or out of) is injected with additional instructions that update the coverage map which is used to track the execution path (trace) of a single execution of the target program. Each basic block is instrumented with the equivalent of the code in Figure 6.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Figure 6. https://aflplus.plus/docs/technical_details/

As the program executes, specific indexes in the coverage map (shared_mem in Figure 6) are incremented to demonstrate that a specific chain of basic blocks have been reached. The index that is incremented is computed by: the current basic block's random ID XORed with the previously executed basic block's random ID right shifted by 1. This process allows the fuzzer to track how many times the provided set of inputs hits a specific sequence of basic blocks (with some collisions occurring), which allows the fuzzer to determine whether the current inputs are adequately advancing coverage. This process is depicted in Figure 6.

The approach that AFL++ takes to coverage integration is not feasible in a hardware system, because this technique requires a sequential process, where hardware is inherently highly parallelized. So, AFL++'s coverage system must be modified so that the fuzzer and forkserver processes accept alternate implementations of coverage metrics. And this is the work that is ongoing.

One of the stated objectives of developing a hardware fuzzing platform is to minimize the degree of device-specific expertise required to begin the verification process. In an ideal fuzzing system, one would only need to modify 1) the beginning input corpus for the fuzzer (a few initial test cases so the fuzzer does not have to guess the structure of the input data), 2) the simulation-to-target program shared memory interface to accommodate different input and result data specifications, and 3) the Radix information flow tracking rules which specify what a "violation" is. With this setup, a verifier would only need to have: an understanding of reliability and security principles, the port map, and brief interactions with a member of the design team to begin the fuzzing process.

With the currently available tooling, however, the changes required to begin the verification process are more complicated. The information flow tracking tool we have utilized is responsible for tracking both the violation status and the coverage achieved in the design. Defining violation properties requires an understanding of reliability and security principles for a specific architecture, and with minimal interaction with a designer, can be tailored to a specific implementation of an architecture relatively quickly and with no modification to the testbench or design code. Integrating the coverage tracking mechanism, however, requires a more significant understanding of the Radix-S tooling and requires potentially significant modifications to the testbench, depending on how many properties are being tracked at the same time. Ideally, we could make this a boilerplate process, but this does not currently exist.

ANTICIPATED OUTCOMES AND IMPACTS:

In this work, a software to hardware pipeline was implemented with and IFT tool. Changes to an existing open source fuzzer were architected and some of the changes were implemented. The final integration was not completed though.
The next steps complete the whole pipeline are:

1. Completing integration of fuzzer.
    a. The easiest path forward is to integrate with AFL++. This may not be the best long-term solution as AFL++ is designed for software fuzzing and has a lot of features that are not necessary for this application.
2. Automating coverage metric extraction.
    a. Currently, finding the security boundary is a manual process. It may be possible to automate this process through static analysis of the design and security rules.
3. There are some optimizations that could be performed on the DPI interface to improve the performance.
    a. There is some work required to make the DPI more generic. The current implementation has mid-level integrations for the current design, so a new design would need to strip components out before useable for other designs.
    b. To track coverage for a 32-bit register requires a significant amount of memory (2^32 bits: 512MB). Current integration tracks coverage in 8-bit register chunks because of significantly lower memory requirements (4 x 2^8 bits: 128B for a 32-bit value). Are there techniques that could improve memory performance?

Finally, to get the performance required for a reasonable throughput of test vectors, the simulation of RTL needs to be migrated to an emulator. This has its own challenges. An emulator is essentially a custom accelerator for RTL simulation. Like all other compute accelerators, the best performance gains are achieved when the traffic between the software and the accelerator is minimized, and the accelerator can compute for long periods of time without software intervention. The current fuzzing pipeline is not architected in this manner. The time taken to move failure, coverage and mutated inputs back and forth would require the emulator to stall. This would likely reduce most of the possible gains of fuzzing on the emulator. There are a few solutions that we will be investigating. The first possible solution is to make sure each "batch" job of test vectors is large enough to dominate the overall compute time. Another option is to speculatively execute mutations before the coverage results are received. This way, the next set of inputs would be ready for execution immediately after the completion of another. Mutations that did not result in increased coverage would need to be rolled back on future iterations of the fuzzer. A more radical approach would be to implement the fuzzer in the RTL testbench. In the emulator, the testbench can run on the emulator. This would eliminate the need to send data back and forth from software.

One shortcut that this work took to get a pipeline working was using a simple DUT that had simple inputs that required little synchronization. One challenge in fuzzing large digital systems is the need for synchronization on many different inputs. For example, a large SOC will require inputs for software as well as inputs from the many peripherals to properly verify the system. There has been a lot of debate if this is possible in a fuzzing solution. While this may be solved by future research in the community, it is unlikely to be a scalable solution. For critical components of DAHCs, fuzzing may a good solution though. DAHCs requires another level of vigor for verification and components such as encryption cores would require extra scrutiny. Tools like fuzzing and IFT can be an integral part of the verification process for systems as such.

CONCLUSION:

Hardware fuzzing has a lot of promise but also a lot of hurdles to become a viable and scalable verification technique as it has been in the software community. In this work we demonstrated that a software to hardware pipeline to facilitate fuzzing is possible and we outline the steps to integrate an open source fuzzing into this pipeline. Finally, we proposed methods to achieve the necessary speedups required by using an emulator. This work has laid the foundation for future work in using hardware fuzzing to rigorously validate RTL for digital components in DAHCS.

REFERENCES:

[1] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, "Theoretical analysis of gate level information flow tracking," *Design Automation Conference*, Anaheim, CA, 2010, pp. 244-247.

[2] https://cycuity.com/solutions/.

[3] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim and B. Lee, "DifuzzRTL, "Differential Fuzz Testing to Find CPU Bugs", *2021 IEEE Symposium on Security and Privacy (SP)",* San Francisco, CA, 2021 pp. 1286-1303.

[4] Timothy Trippel and Kang G. Shin and Alex Chernyakhovsky and Garret Kelly and Dominic Rizzo and Matthew Hicks, "Fuzzing Hardware Like Software", *31st USENIX Security Symposium*, Boston, MA, 2022, pp. 3237-3254.

[5] W. Fu, O. Arias, Y. Jin and X. Guo, "Fuzzing Hardware: Faith or Reality? Invited Paper, "*2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH),* AB, Canada, 2021, pp. 1-6.

[6] H. Siemen, J. Lienke and G. Gläser, "Hot Fuzz: Assisting verification by fuzz testing microelectronic hardware," *2023 19th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* Funchal, Portugal, 2023, pp. 1-4.

[7] T. Tripple, K. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo and M. Hicks, "Fuzzing Hardware Like Software," *Proceedings of the 31st USENIX Security Symposium*, 2022, Boston, MA, pp. 3237-3254.

[8] https://sandialabs.sharepoint.com/sites/DAHCS/SitePages/FY24-Late-Start-LDRD-Call.aspx.

[9] https://aflplus.plus/

[10] K. Laeufer, J. Koenig, et al. "RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, CA, 2018, pp. 1–8, 2018.