1. ENEL 592 - Final Report

1.1. Table of Contents

- 1. ENEL 592 - Final Report
  - 1.1. Table of Contents
  - 1.2. Introduction
  - Hardware Security Verification
  - 1.3. System-on-Chip Platform
  - 1.4. CWE Selection
    - 1.4.1. CWE-1274
    - 1.4.2. CWE-1189
    - 1.4.3. CWE-1231
    - 1.4.4. CWE-1233
    - 1.4.5. CWE-1240
    - 1.4.6. CWE-1256
    - 1.4.7. CWE-1191
    - 1.4.8. CWE-1244
    - 1.4.9. CWE-1260
    - 1.4.10. CWE-1272
    - 1.4.11. CWE-1276
    - 1.4.12. CWE-1277
    - 1.4.13. CWE-1300
    - 1.4.14. Selected CWEs
  - 1.5. RTL Bug Insertion
    - 1.5.1. Bug 1: Incorrect Lock Bit Behaviour
    - 1.5.2. Bug 2: Persistent SRAM Data
    - 1.5.3. Bug 3: Unwritable Flash Memory
    - 1.5.4. Bug 4: Software-Readable Key Register
    - 1.5.5. Bug 5: Memory Range Overlap Reversed Priority
    - 1.5.6. Discussion
  - 1.6. HLS-Induced Security Weaknesses
    - 1.6.1. Security Weakness: Intermediate Result Leakage
    - 1.6.2. Automatic Detection: Static Analysis
    - 1.6.3. Automatic Detection: Formal Verification
      - 1.6.3.1. Property Formulation
      - 1.6.3.2. Verification Environment Generation
    - 1.6.4. Automatic Correction: Directive Generation
    - 1.6.5. Experimental Results and Discussion
  - 1.7. Conclusion
  - 1.8. Appendix A: OpenTitan
    - 1.8.1. Architecture
    - 1.8.2. Security Features
    - 1.8.3. Collateral

1.2. Introduction

The aim of my ENEL 592 final project is to insert a set of security bugs into an System-on-Chip (SoC) design, and create associated testbenchs and firmware that demonstrate their implications. This is the culmination of my two previous assignments, where I surveyed hardware security verification and open-source SoC designs. The bugs should be as "realistic" as possible; they should resemble bugs found in-the-wild and be impactful.

Next semester, I will build on this project and approach the problem from the other side of the coin -- bug detection and/or correction. The resulting SoC will also serve as a good benchmark for this future work.

Hardware Security Verification

My survey of the cutting edge techniques under active research is given in the Appendix. I focused on three techniques, Static Analysis, Fuzzing, and Property Generation.

1.3. System-on-Chip Platform

The SoC I used for bug injection is the OpenTitan SoC, which I detailed in assignment 2. An excerpt of assignment 2 describing the OpenTitan SoC can be found in the appendix A.

1.4. CWE Selection

The inserted bugs should be impactful and representative of those found in the wild. They should also be "distributed" and affect different parts of the SoC while still being security-critical. I relied on the Hardware CWEs to gain inspiration for candidate bugs. The hardware CWEs is a list of common weaknesses found in hardware designs. They are not bugs themselves, but are often found in designs as a result of bugs.

The 2021 CWE Most Important Hardware Weaknesses contains the most important hardware CWEs of 2021, evaluated using the following criteria:

1. How frequently is this weakness detected after it has been fielded?
2. Does the weakness require hardware modifications to mitigate it?
3. How frequently is this weakness detected during design?
4. How frequently is this weakness detected during test?
5. Can the weakness be mitigated once the device has been fielded?
6. Is physical access required to exploit this weakness?
7. Can an attack exploiting this weakness be conducted entirely via software?
8. Is a single exploit against this weakness applicable to a wide range (or family) of devices?
9. What methodologies do you practice for identifying and preventing both known weaknesses and new weaknesses?

This list is as a valuable starting point because it provides insight into industry and the challenges currently faced. My intuition was that analyzing and implementing bugs that fall within these CWEs should fulfill the desired criteria (realism and impact) and provide the most value for future benchmark uses.

The list contains 12 CWEs:

1. CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC)
2. CWE-1191: On-Chip Debug and Test Interface With Improper Access Control
3. CWE-1231: Improper Prevention of Lock Bit Modification
4. CWE-1233: Security-Sensitive Hardware Controls with Missing Lock Bit Protection
5. CWE-1240: Use of a Cryptographic Primitive with a Risky Implementation
6. CWE-1244: Internal Asset Exposed to Unsafe Debug Access Level or State
7. CWE-1256: Improper Restriction of Software Interfaces to Hardware Features
8. CWE-1260: Improper Handling of Overlap Between Protected Memory Ranges
9. CWE-1272: Sensitive Information Uncleared Before Debug/Power State Transition
10. CWE-1274: Improper Access Control for Volatile Memory Containing Boot Code
11. CWE-1277: Firmware Not Updateable
12. CWE-1300: Improper Protection of Physical Side Channels

These 12 CWEs are all applicable to bug insertion at the RTL to varying levels. They can all get introduced during the implementation phase, as noted on their CWE pages, which is the development phase I am focusing on. Some do not appear applicable at first glance, but are fairly open to intepretation because they are so generic. For example, CWE-1240: Use of a Cryptographic Primitive with a Risky Implementation mainly mentions the use of "weak" cryptographic primitives (e.g., weak algorithms like MD5), but this can also be understood as the incorrect implementation of a strong algorithm. The latter may be suitable for this project depending on how much modifification to the original design is required.

To narrow down the list of CWEs to insert, I further classified them by CWE Category, the highest level of the CWE hierarchy. Again, the goal was to develop a distributed set of bugs and classifying them by category will allow me to gain the most functional variety. The CWE categories and their summaries were obtained from the CWE list.

CWE-1196 - Security Flow Issues: weaknesses in this category are related to improper design of full-system security flows, including but not limited to secure boot, secure update, and hardware-device attestation.

- CWE-1274: Improper Access Control for Volatile Memory Containing Boot Code

CWE-1198 - Privilege Separation and Access Control Issues: weaknesses in this category are related to features and mechanisms providing hardware-based isolation and access control (e.g., identity, policy, locking control) of sensitive shared hardware resources such as registers and fuses.

- CWE-1189: Improper Isolation of Shared Resources on System-on-a-Chip (SoC)
- CWE-1260: Improper Handling of Overlap Between Protected Memory Ranges

CWE-1199 - General Circuit and Logic Design Concerns: weaknesses in this category are related to hardware-circuit design and logic (e.g., CMOS transistors, finite state machines, and registers) as well as issues related to hardware description languages such as System Verilog and VHDL.

- CWE-1231: Improper Prevention of Lock Bit Modification
- CWE-1233: Security-Sensitive Hardware Controls with Missing Lock Bit Protection

CWE-1205 - Security Primitives and Cryptography Issues: weaknesses in this category are related to hardware implementations of cryptographic protocols and other hardware-security primitives such as physical unclonable functions (PUFs) and random number generators (RNGs).

- CWE-1240: Use of a Cryptographic Primitive with a Risky Implementation

CWE-1206 - Power, Clock, Thermal, and Reset Concerns: weaknesses in this category are related to system power, voltage, current, temperature, clocks, system state saving/restoring, and resets at the platform and SoC level.

- CWE-1256: Improper Restriction of Software Interfaces to Hardware Features

CWE-1207 - Debug and Test Problems: weaknesses in this category are related to hardware debug and test interfaces such as JTAG and scan chain.

- CWE-1191: On-Chip Debug and Test Interface With Improper Access Control
- CWE-1244: Internal Asset Exposed to Unsafe Debug Access Level or State
- CWE-1272: Sensitive Information Uncleared Before Debug/Power State Transition

CWE-1208 - Cross-Cutting Problems: weaknesses in this category can arise in multiple areas of hardware design or can apply to a wide cross-section of components.

- CWE-1277: Firmware Not Updateable

CWE-1388 - Physical Access Issues and Concerns: weaknesses in this category are related to concerns of physical access.

- CWE-1300: Improper Protection of Physical Side Channels

I continued by analyzing these CWEs in detail. I will discuss how we can generally characterize these CWEs such as where they can occur and how bugs *may* manifest in hardware designs to introduce these weaknesses. I will also introduce and discuss another CWE, CWE-1276: Hardware Child Block Incorrectly Connected to Parent System, because it is one that I personally encountered in the Hack@DAC 2022 competition. It is important to mention that I am not trying to develop a definitive set of bugs for any CWE, rather I am attempting to demonstrate how a bug can introduce a CWE.

Since I am operating at the RTL implementation stage, the characteristics under consideration are the functional locations (both inter-and-intra-modular) where they can get introduced, the sequence of logical operations involved, and errors in these logical operations that result in CWEs. These characteristics were chosen because they give meaningful insight into the bug insertion process and provide a formalized way to introduce bugs. The characteristics of possible bugs such as the # of lines modified will be discussed in a later section.

1.4.1. CWE-1274

CWE-1274 is the lack of access control for volatile memory containing boot code. The secure boot process typically consists of first executing a small program residing in ROM which loads trusted firmware from flash memory into volatile memory to be executed. This trusted firmware is responsible for the bulk of the secure boot of the system, such as going to user mode after system configuration, in the case of bare-metal systems. Attackers may thus try to modify the firmware when it is in the volatile memory to cause insecure behaviour, such as not going to user-mode at the end. It is crucial to have proper access control in place to ensure that this trusted firmware cannot be written to once it has been loaded in from flash memory. This CWE is really a subset of a larger concern which is the proper implementation of access control in memory controllers. A warranted specificity, given the importance of boot firmware.

This concern can be localized to the two different concerns: (i) the bootloader must configure the access control policy such that the loaded firmware cannot be tampered, and (ii) memory controllers responsible for enforcing access control policies must be functional. The

first concern is a software resposibility and outside the scope of this project. The second concern, however, is very relevant to RTL implementation. For this CWE, we are specifically concerned with controllers of volatile memory, such as DRAM and SRAM. DRAM controllers are typically off-chip components and the OpenTitan SoC itself does not offer a DRAM controller for analysis. Within the memory controllers, the specific parts of concern are the configuration of the memory regions and access control to those regions, and the enforcing of those regions. The concern with the former is the ability to read and write to the configuration registers, and the concern with the latter, is the ability to properly enforce the configurations at all times. If we assume "secure" ROM code that properly attempts to configure the access control, the general sequence of logical operations are:

1. Write boot code from Flash to DRAM/SRAM
2. Write access control policy to SRAM/DRAM controller
3. lock access control policy so it cannot be changed

Any errors in these steps may cause confidentiality, integrity, or availability concerns for the systems as a whole. Also some of the potential errors that could occur may fall under other CWEs (e.g., lock bit protection as discussed later).

1.4.2. CWE-1189

CWE-1189 is the improper isolation of shared resources in an SoC. This is fairly generic and depends a lot factors. "Shared resources" may consist of anything from memory to pins. There are two ways that this weakness is addressed. First, resources may not be shared between entities with different trust levels. Second, shared resources between different trust levels should have protection mechanisms in place, such as access control to regulate the access between the different trust levels. The first is ideal but may not always be practical, for example, having dedicated "secure RAM" is not something that is done (as far as I know). Because this CWE is so broad, it hard to identify specific design/implementation aspects to focus on. The proper isolation of resources depends heavily on the use-case of the resources themselves and the way that software is expected to interact the shared resources. We can generally state that any resources that are expected to be shared between different trust levels must have isolation features integrated. For memory, this means memory range configurations, for memory-mapped pheripherals, this means lock bits for configuration registers, etc. Because there are so many aspects to this CWE, I will refrain from attempting to localize areas of concerns.

1.4.3. CWE-1231

System configuration registers are often protected by lock bits. This is required for configurations that are critical to the security of the device. For example, systems typically only operate normally inside of a well defined temperature range. Outside of that temperature range, system behaviour can become unpredictable. Security-critical devices should detect such extreme temperature ranges and deal with them appropriately (e.g., clearing assets from memory, shutdown, etc.). The configuration registers are typically written by trusted software during boot and locked afterwards to ensure that they are not modified. The ability to lock these registers is crucial to ensuring that security features that rely on them cannot be circumvented. CWE-1231 is the improper prevention of the modification of these lock bits.

The example listed in the CWE website, in my opinion does not deliver an accurate representation of the actual weakness. However, it does demonstrate the conceptual challenge involved with using lock bits effectively. Consider the registers listed in Table 1. The goal of these registers is to detect when operation temperature has gone above the allowable max (125 Centigrade by default). As shown in the table, the temperature limit, `CRITICAL_TEMP_LIMIT`, and sensor calibration, `TEMP_SENSOR_CALIB` are lockable using `TEMP_SENSOR_LOCK`. This ensures that the limit cannot be changed and that the sensor readings are accurate. However, notice that the register that enables hardware shutdown, `TEMP_HW_SHUTDOWN`, is not lockable and that the enable bit is read/write. This means that unprivileged software may have the ability to write to this register and disable the critical temp response. This clearly does not satisfy the intent of this security feature and undermines its functionality. A more secure solution would be to also lock this register using `TEMP_SENSOR_LOCK`, for practically no cost. This demonstrates the bigger challenge in my mind -- to determine what is a "security asset" that must be locked.

Table 1: CWE-1231 Example Registers

Once again, this specific CWE is related the improper modification of lock bits. We assume that every asset has been correctly identified and protected with a lock bit, and that the lock bit effectively protects the asset. Our concern are unathorized writes to the lock bits that disable them incorrectly. For example, CVE-2017-6283 was from a vulnerability in an NVIDIA product resulting in the incorrect clearing of read/write lock bits of the keyslot of an RSA function. This is one such scenario where clearing a register is not the desired behaviour (a somewhat counterintuitive behaviour). This real-life vulnerability gives valuable insight into both the localization and errors in logical operations required to introduce this CWE. Obviously, this CWE can only manifest where there are lock bits. Assuming that lock bits are local to only their modules (i.e., a lock bit for a register inside of a module will not be an output of that module), then we can restrict our search for modules that contain lock bits. Next, within that module, every assignment to that lock bit could be a potential location for that CWE. Some conceptual questions to ask are: (i) when can this assignment happen? (ii) does the value being assigned make sense? (iii) do the proper steps occur prior to the assignment if applicable (e.g., authetication)? These questions help evaluate the security of the lock bit.

Any logical errors, such as the bit being cleared on reset, should be detected by using these questions (and more). On the flip side, we can take advantage of this to add incorrect assignments to the lock bit and introduce the CWE.

1.4.4. CWE-1233

CWE-1233 is somewhat of a "step back" to CWE-1231 (counterintuitive given the numbering). CWE-1233 is the lack of lock bit protection for security-sensitive hardware controls. In CWE-1231, we assumed that all necessary configuration registers had the necessary lock bits. This is not always the case and the determination of configuration registers that must be protected often comes down to human expertise, known past vulnerabilities, and processes based on those such as threat modelling. A scenario where a security-sensitive configuration register is not protected is possible due to either the heavy reliance on human knowledge and/or that a vulnerability related to that configuration is not known or considered. The potential impact of this CWE are consistent with CWE-1231's but the root cause is completely different. The lack of lock bit protection generally comes from more upstream sources such as the architecture and design stages of the development lifecycle but they can nonetheless still be introduced in the implementation stage. For example, errors like protecting the wrong register due to a typo, or the system not actually using the lock bit even if it is present. These are more liberal intepretations of the CWE but I believe they are fundamentally the same. Ultimately, if the lock bit does not protect what it is meant to protect, then it may as well be non-existent. This is still different from CWE-1231 in that CWE-1231 does protect its asset, it's value can just be incorrectly changed.

My liberal intepretation of the CWE make localization efforts challenging. There are two separate concerns. First, there the "original" meaning, the lack of lock bit protection where it is required. This can only be remedied by identifying all possible assets and verifying that they do lock bits. So localizing this intepretation means taking a list of all assets and asserting that they have a protection mechanism around them. Obviously, the localization concerns stem from the ability to identify these assets. Second, the "liberal" meaning, which is to ensure that the lock bits that are present do in fact protect the asset. Again, here we are focused on assets but also have additional context in that we know they are protected. Then, a simple approach to localizing potential issues is to consider every configuration register which has a lock bit. The logical operations for the first concern are none, as they are not present in the implementation at all. For the second concern, the general sequence of operations are:

1. write configuration register
2. write lock bit
3. check lock bit before writing to configuration register again
4. deny write if lock bit is set

The key steps are 2-4. We must ensure that the lock bit is properly writable, we must ensure that the lock bit is checked before accessing the configuration register, and we must deny the access if the lock bit is set. Any errors in these crucial steps, such as checking the lock bit concurrently/after the access (another CWE in itself), can introduce this CWE into the system.

1.4.5. CWE-1240

CWE-1240 is the use of a cryptographic primitive with a risky implementation. Cryptopgraphic algorithms are a popular use case for hardware acceleration. In many cases, these any hardware cryptopgraphic accelerators are used for both software requests and hardware features like device attestation. As stated in the CWE page, "for hardware implementations of cryptographic primitives, it is absolutely essential that only strong, proven cryptographic primitives are used." Algorithms like DES should not be used because they are proven to be too easily cracked. Furthermore, all aspects of the cryptographic algorithms should be considered, including the Random Number Generator (RNG) used for generating the Initialization Vector (IV) or Number Used Once (nonce). A more subtle concern that I believe falls under this CWE and is not discussed in the CWE page is the incorrect implementation of a strong algorithm. If we consider a scenario where a system designer is including an AES accelerator into their design, then it is imperative that they adhere to the specifications of the algorithm and protect against other known attack vectors such as side-channel attacks. This is why in the software domain at least, it is recommended to use cryptographic libraries such as OpenSSL, developed and scrutinized by experts. Unfortunately, hardware designs are privy to more secrecy which incentivizes the development of in-house cryptographic hardware. Any oversight in this custom implementation can have disastrous effect and lead to product recalls.

This CWE can be introduced by any aspect of the design related to cryptography. Before even inspecting the implementation, the algorithms being implemented themselves could introduce this CWE. This includes encryption algorithms, hashing functions, Random Number Generators, block cipher modes of operation. Once all algorithms have been validated as "secure", then each algorithm's conformance to its specification must be validated, if applicable. This is not limited to the functional input/outputs, it also includes the internal state and funtion as well. Every action in these algorithms are meaningful and modifying even one can negatively affect its security. There is no specific sequence of operations to be aware of for this CWE as it can occur in a broad range of scenarios.

1.4.6. CWE-1256

CWE-1256: Improper Restriction of Software Interfaces to Hardware Features is another very broad CWE. It is generally concerned with software having access to hardware which can lead to insecure behaviour in the hands of a malicious actor. In my mind, this is the "root cause" of all software-exploitable hardware vulnerabilities. Ultimately, if software had no control or access over the hardware it is running on, then hardware would hardly be a concern. Unfortunately, software must have some level of control and access to hardware features for performance (who needs that, right?) and these vulnerabilities arise from the software-hardware interactions. Conceptually there are two aspects to this CWE: (i) does software have access to something it shouldn't? (ii) when should software be able to access this? The first aspect is questioning whether software, privileged or not, should have access to a hardware feature at all. The second aspect is questioning the conditions which must be met to consider the software access to be "secure". This can range from "obvious", such as Memory Mapped IO configuration (whose protection is discussed above with lock bits) to not-so-obvious such as memory accesses. An infamous example of this is the RowHammer which causes bitflips in memory (hardware) due to specific memory access patterns (software).

The potential locations for this CWE are locations that are directly controllable by software, so the entire design. The localization can be slightly restricted to what is deemed to be "security-critical" but can quickly lead to a false sense of security. Because hardware is a finite resource, there is bound to be sharing across security domains and such sharing must also be identified and labelled appropriately. Similarly to the previous CWE, this CWE is too broad to pin down to one sequence of operations. It inevitably depends on the context and the way in which software interacts with the hardware.

1.4.7. CWE-1191

Hardware designs contain debug infrastructure meant to assist in post-silicon validation and quality-control. This debug infrastructure typically consists of an access port (e.g., JTAG) and a scan chain that allows for easy shift in and out of registers. This is the closest to "white-box" access possible post-silicon and can expose secure assets if not designed properly. There is a rich body of literature available exploring this topic. CWE-1191: On-Chip Debug and Test Interface With Improper Access Control is the improper protection of assets from debug interfaces. Assets like cryptographic keys must be kept secret at all times and any exposure to unauthorized actors may lead to information disclosure. This can involve multiple mechanisms such excluding certain registers from the scan chain and adding access control mechanisms.

Similarly to many of the other CWEs, this CWE's localization is very dependent on what the assets of the design are, but in this case also what the debug infrastructure is. If these two pieces of contextual information are available, then we can isolate areas of interest as the intersection between the two (where assets meet debug infrastructure). The bigger challenge is to actually identify what the assets are. Nowadays, it can be considered "industry practice" to protect security assets from the scan chain. In my mind, any issues would originate from the inability to identify assets requiring protection correctly in the architectural and design stages.

1.4.8. CWE-1244

CWE-1244 is related to this debug port protection and can be considered the "next step" of CWE-1191. Where CWE-1191 is the lack of debug access control mechanisms to protect assets, CWE-1244 is the improper use of available access control. The example present on its page delivers the "intent" behind this CWE quite well. Consider a scenario where an attacker has physical access to a device and JTAG port. There is an access control bit that enables and disables the JTAG debugger, `JTAG_SHIELD`. However this bit is not set on boot-up, instead, it is set when control is transferred over to user-level software. This leaves the system vulnerable during the boot-up period, when `JTAG_SHIELD` is in some unknown state, and may allow the attacker to read or write secure assets. For example, they could modify the instructions in memory to modify the boot flow. From this, we can intuitively understand not only this specific scenario, but the CWE in general.

This CWE can manifest in any IP block which stores or uses control and status registers related to debug access control. While this is likely to be in debug-related modules it is not necessarily the case. For example, consider the code snippet from the Hack@DAC 2021 OpenPiton SoC shown in Figure 1. This snippet was taken from the top-level of an AES accelerator. We can deduce that `debug_mode_i` is a debug-related access control signal that denies read access to the keys when in debug mode. However, one of the keys in not protected -- a security bug that can lead to the leakage of that key. I consider this to be part of this CWE instead of CWE-1191 because there is an access control mechanism in place, it was just used incorrectly. The point is that debug-related bugs do not always appear in debug-related modules (although it does a great job of illustrating that one CWE can manifest in many different ways when considered with the previous scenario). This specific CWE, however, **can only appear in modules where there is a debug access control signal**. This is a key intuition that can guide both its insertion for this project and any future detection/correction work.

```
assign key_big0    = debug_mode_i ? 192'b0 : {key0[0], key0[1], key0[2], key0[3], key0[4], key0[5]};
assign key_big1    = debug_mode_i ? 192'b0 : {key1[0], key1[1], key1[2], key1[3], key1[4], key1[5]};
assign key_big2    = {key2[0], key2[1], key2[2], key2[3], key2[4], key2[5]};
```

Figure 1: Hack@DAC 2021 Debug AES Keys Access Control Bug

The sequence of logical operations involved for this CWE are relatively simple, as it must all be related to reads/writes to the aforementioned debug access control signal. The challenging part is determining all appropriate time where these operations (read/write) must happen. In cases where there are multiple debug access levels, the value being read/written is also important. The first scenario I discussed presented a situation where access control was written too late, the second scenario presented a situation where it was not read when it should have been. It follows that any modification to these reads or writes could introduce this CWE. Considering the two scenarios again, this could mean removing the reset value of the register storing the bit and removing an access control check (as is shown in the snippet), respectively.

1.4.9. CWE-1260

Memory in computer systems is organized into ranges that are controlled by software and enforced by a Memory Management Unit (MMU) or a Memory Protection Unit (MPU). There are also physical memory regions enforced by the Physical Memory Management (PMP) unit, meant to separate physical memory space for each hardware thread (or *hart*). For example, the RISC-V privileged specification contains a PMP implementation. The software-controlled address ranges are typically software-configurable to allow for dynamic change during operation. CWE-1260 is related to the overlapping of these memory ranges. While overlapping memory regions is typically allowed, it can introduce risks if memory ranges with different privilege levels are overlapping and the MMU/MPU is not designed to handle these overlaps well. Consider a scenario where there are two memory regions, `region1` and `region2`. `region1` is dedicated to privileged software and its configuration (location and size) can only be modified by privileged software. `region2` is usable and configurable by both privileged and unprivileged software. A potential attacker can configure `region2` such that it overlaps with `region1`, and give itself the ability to read/write/execute the privileged memory. To address these issues, overlap between different access levels should not be allowed or a priority hierarchy should be established. Using the same scenario, the priority required to access the overlapped region should be the highest level of priority required of either regions.

This CWE is challenging to mitigate because address spaces are dynamically configured at runtime. There is no way of pre-verifying address ranges during design/implementation/validation. From a hardware standpoint, the only course of action is to design/implement/verify the MMU/MPU to ensure it implements security features that address these issues. It follows that at the hardware level, this CWE can manifest in memory control units that are responsible for configuring and enforcing memory ranges. Specifically within those designs, the functionality that performs the access control checks is of interest. Figure 2 illustrates the priotization of PMP regions in the Ibex core used in the OpenTitan SoC. This is one of the functional regions where this CWE can get introduced. As the comment notes, the PMP entries are prioritized from 0 up to N-1. This conforms with the RISC-V privileged specification. A simple albeit potent bug here would be to reverse this ordering to N-1 down to 0.

```
// PMP entries are statically prioritized, from 0 to N-1
// The lowest-numbered PMP entry which matches an address determines accessibility
for (int r = 0; r < PMPNumRegions; r++) begin
  if (match_all[r]) begin
    access_fail = ~final_perm_check[r];
    break;
  end
end
```

Figure 2: Ibex Core PMP Memory Region Priotization

1.4.10. CWE-1272

CWE-1272 is related to operational state transitions such as going from debug mode or boot-up to operation. There is often secure information which is required in that state but should not be accessed in any other states. This CWE is introduced when this secure information is not cleared during state transitions. For example, a key used for device attestation during boot should not be lingering in memory after the boot is complete. It is imperative to clear any memory or registers that store such sensitive information when transitioning states. Even in cases where it is deemed "safe" such as in internal registers, the goal is to implement and adhere to the principle of least privilege. Allowing secure assets to linger when they are not required introduces unnecessary security risk. For a more concrete example, consider the following scenario. A secure system implements a One-Time Programmable (OTP) memory like fuse memory to store a unique key used to derive all other keys. This root key must be loaded in from the OTP during boot-up for said key derivation. Assume that the key derivation process is sequential and that each key created only depends on the one before it. The root key should thus only be persistent until the first key is created, and should subsequently be cleared. Failure to clear this key and any other sensitive information may allow attackers to access it and introduce vulnerabililties to the design.

This CWE is fairly broad with respect to potential location. State transitions affect the system as a whole and narrowing down potential locations must be approached by considering every state, the assets required within that state, and the actions done leaving that state. Generally speaking, actions upon entering a state are important to consider as well to check the integrity of assets but this is outside the

scope of this CWE. I believe there are two ways this localization can be approached. First, we can leverage the control and status registers/signals and "track" them through the SoC to determine where they interact with sensitive information. Second, we can begin with enumerating sensitive assets and "track" those through the different state transitions. Ultimately, the end result is that noteworthy locations are **where sensitive assets and control/status signals intersect**. Figure 1 provides an example of this intersection in a different context, between the debug status signal and the AES keys. This CWE cannot be localized to any specific IP type or functionality as it depends heavily on contextual information such as what asset is required for what state.

The logical operations related to this CWE appear relatively simple but are again very context-dependent. For example, in a boot-up setting the sequence of states is well defined and required assets are also typically well defined. In other power state transitions, such as *normal power, additional power, low power, hibernate, deep sleep, etc.*, as listed in the CWE page, the transition is dynamic and the required assets vary heavily. However we can generalize this and say that transitioning from State A to State B requires that sensitive information used in State A is cleared. The key generalization here is that we are not concerned with what state B requires and simply clear everything during transition, and we assume that if it requires it, it will have ability to access it. The specific operation we are focused on for bug insertion is thus the clearing of this data.

1.4.11. CWE-1276

CWE-1276: Hardware Child Block Incorrectly Connected to Parent System is not part of the 2021 Most Important Hardware CWE list. I decided to include it in this project because I personally encountered it in the Hack@DAC 2022 Competition finals and found it interesting. Hardware designs are typically broken up into "modules" when written in HDLs. The modules are meant to "black-box" functionality and enable reuse, much like classes used in software development. The integration of the different modules that make up a complete design is crucial to ensure that it behaves as expected. Even if each sub-module of a design are fully verified, any errors in their integration may result in security concerns. For example, consider the following module port definition from the 2021 Hack@DAC SoC:

```
module aes1_wrapper #(
    parameter int unsigned AXI_ADDR_WIDTH = 64,
    parameter int unsigned AXI_DATA_WIDTH = 64,
    parameter int unsigned AXI_ID_WIDTH   = 10
)(
  input logic               clk_i,
  input logic               rst_ni,
  input logic [7 :0]        reglk_ctrl_i,
  input logic               acct_ctrl_i,
  input logic               debug_mode_i,
  input  ariane_axi::req_t  axi_req_i,
  output ariane_axi::resp_t axi_resp_o,
  input logic               rst_2
  );
```

All of these signals are critical to the proper behaviour of the module, but the `debug_mode_i` signal is of particular interest. This signal is used to protect the AES keys in debug mode, as discussed above. If the module is instantiated as shown in listing 1, then the keys would be accessible during debug. This may seem "too simple" but I consider it to be a very realistic in scenarios where for example, the debug insfrastructure is still a work-in-progress and the port is never updated once the implementation is complete. This CWE can occur at any and every module instantiation. Localization of possible areas of concern can then done by considering all module instantiations, then filtering for "security-critical" modules, then "security-critical" signals. Here the bigger challenge is finding those "security-critical" aspects of the design, which varies design by design.

```
aes1_wrapper aes1_u0(
  .clk_i(clk),
  .rst_ni(rst_n),
  .reglk_ctrl_i(reglk_ctrl),
  .acct_ctrl_i(acct_ctrl_i),
  .debug_mode_i(1'b0),
  .axi_req_i(axi_req),
  .axi_resp_o(axi_resp),
  .rst_2(rst_2)
  );
```

Listing 1: Buggy aes1_wrapper Module Instantiation

1.4.12. CWE-1277

This CWE, the inability to patch firmware does not appear to be the most relevant to the purpose of this project at first glance. A bug cannot remove the entirety of the patching infrastructure in place. It can, however, make it so that the patching infrastructure is not accessible. It is realistic to imagine that the ability to patch firmware must be protected, as unauthorized modifications should never occur. Any implementation errors in the authentication/patch integrity process could result in a denial of service to the patching infrastructure. The example used from the CWE website is fairly generic but one of the key points is that they also mention that oversight during implementation can lead to this CWE, validating my interpretation of the CWE.

For the context of this project, I assume that the ability to patch firmware is intergrated into the design. The potential locations that this CWE can get introduced is then path from where patches can be installed to the memory storing the firmware. The challenge is establishing the start of the path -- where the patch originates from as it starts from software. An alternative is to start with the asset (e.g., the flash memory storing the firmware) and work backwards to understand the data path and control path intended for patching. Generally, it will involved authenticating the patch (Is it coming from an authorized source?), checking its integrity (Was it tampered with?), and writing to the flash. These steps are not necessarily controlled through hardware and it is thus challenging to generalize potential locations across designs. The closest we can get is to inspect the ability to write to the Flash memory. Since "patching" simply refers to modify the stored instructions, it depends on the ability to write to the memory. The inability to write implicitly results in the inability to patch.

As mentioned above, the logical operations, at a high-level, for a software-initiated firmware patch are: (i) software initiates patch, (ii) the source of the patch is autheticated, (iii) the integrity of the patch is checked, (iv) the existing firmware is either overwritten or the new software is written at another memory location. Any implementation errors in these steps can result in the introduction of CWE-1277. Since many of these steps are handled by dedicated software typically built into the OS, there is no "standard" flow of hardware operations. We can assume that errors in any cryptographic accelerators will result in failure in the authentication/integrity steps. More "subtle" issues are likely centered around the writability to Flash memory storing the firmware. For example, in the OpenTitan SoC, data in the flash is scrambled as shown in Fig. 3. This mechanism uses a local PRINCE cipher. Errors in this PRINCE cipher result in bad data being written to the flash while not propagating outwards to affect other IP.

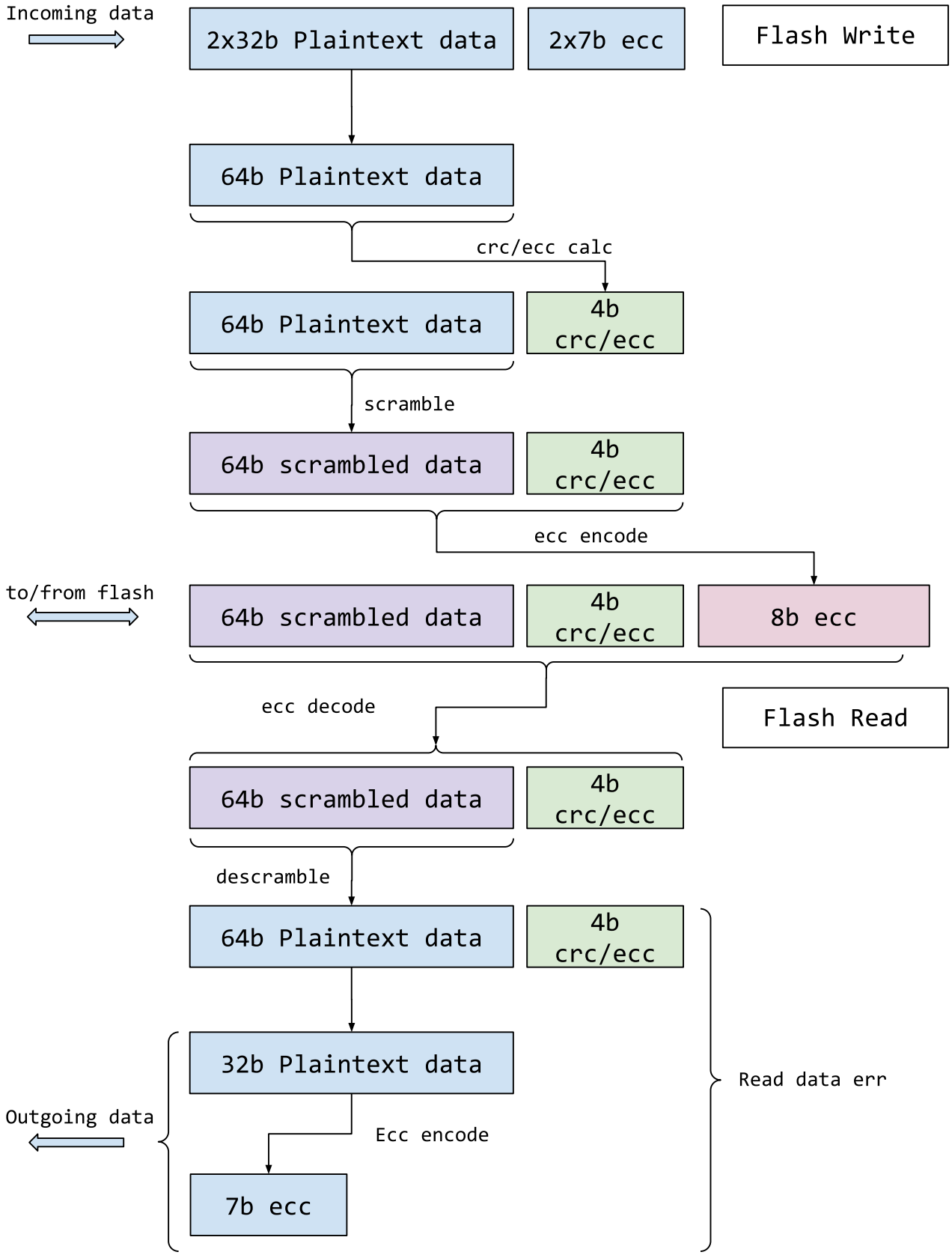

Figure 3: OpenTitan Flash Data Scrambling Flow

1.4.13. CWE-1300

CWE-1300 is the improper protection of physical side-channels. Side-channels, such as power consumption, electromagnetic emissions, and timing have been studied for decades and are a proven way to circumvent security measures. They typically require physical access but that is not always the case. There are established best practices to reduce the presence of side-channels as much as possible in security-critical applications, but these are not always implemented properly, or the assets to protect are not all identified. As the CWE page, mentions this weakness is generally addressed at the architectural stage with countermeasures like constant-time operations and

masking, and in the implementation stage by restricting access to locations that may leak side-channel information (e.g., power pins). The first countermeasure introduces often significant performance overhead, and the second is not always possible. For the context of this project, this CWE may be introduced by errors in the implementation of countermeasures at the RTL. For example, errors in the masking scheme of a cryptographic accelerator may not be as effective as intended.

We can localize potential locations of this CWE to aspects of the design whose behaviour depend on the data they are operating on. Side-channels is all about being able to deduce secret information from the side-effects of hardware execution. For example, in a non-masked AES accelerator the power consumption of the block will depend on the plaintext and key. By observing this difference over multiple inputs (e.g., supply known plaintexts to recover unknown key), it is possible to recover sensitive information. Unfortunately, this is still fairly broad and identifying the parts that act on "sensitive" information or not is a challenge that will vary by context. In the OpenTitan SoC, side-channel protection is mainly implemented in the AES core and the KMAC core. This aligns with my observation that side-channels are generally mostly considered for cryptopgraphic situations, for better or for worse.

1.4.14. Selected CWEs

For each category, I chose a representative CWE that I believe will require the most minimal amount of modification to the design to demonstrate how easily they can introduced and to make them as "stealthy" as possible, theoretically making them more challenging to detect. Then, I filtered it down to a final set of 5 CWEs to implement. The criteria for this filter was simply personal interest.

The final set of CWEs I chose consists of:

1. CWE-1231: Improper Prevention of Lock Bit Modification
2. CWE-1272: Sensitive Information Uncleared Before Debug/Power State Transition
3. CWE-1277: Firmware Not Updateable
4. CWE-1276: Hardware Child Block Incorrectly Connected to Parent System
5. CWE-1260: Improper Handling of Overlap Between Protected Memory Ranges

Even though I am interested in side-channel and cryptographic weaknesses, I ultimately chose to forgo them because developing exploits for these weaknesses are involved tasks. They both typically require many inputs to statistically piece together secure information but this would be cumbersome to demonstrate in a testbench setting. They are also generally harder to introduce through the small implementation bugs that I will be doing here.

1.5. RTL Bug Insertion

1.5.1. Bug 1: Incorrect Lock Bit Behaviour

As discussed above, correct lock bit behaviour is critical to secure behaviour. One application of lock bits in the OpenTitan SoC is for the write-enable of cryptographic accelerator configuration registers. It is crucial to ensure that these configuration registers cannot be modified during operation because an attacker could manipulate them to recover secret information like the key or cause denial of service. For example, the key could be updated during operation to cause errors in the encryption or hash.

The KMAC IP in the OpenTitan SoC contains such a lock bit, called `cfg_regwen`. This bit controls the write-enable of pracitically all sensitive registers in the KMAC IP, such as the CSR registers, key registers, the hast count register, etc. By default, this bit is set high (the registers are writeable) when it is in idle. This funtionality is implemented in the OpenTitan SoC using two lines, as shown in Fig. 1. It is fairly straightforward -- `cfg_regwen` is set to high if and only if the KMAC module is in `IDLE`. To insert a bug into this behavior a simple but effective alteration is to modify it so that `cfg_regwen` is always high. It is also reasonable to assume this would be a "real-life" mistake during development/debug if for example, there was a bug in the FSM and the designer temporarily wanted the ability to always write to CSR registers but forgot to change it back afterwards. The new, buggy behaviour might then be described as shown in Fig. 2.

```
// Configuration Register
logic engine_stable;
assign engine_stable = sha3_fsm == sha3_pkg::StIdle;

// SEC_CM: CFG_SHADOWED.CONFIG.REGWEN
assign hw2reg.cfg_regwen.d = engine_stable;
```

Figure 1: Original KMAC Lock Bit Behavior

```
// Configuration Register
logic engine_stable;
assign engine_stable = sha3_fsm == sha3_pkg::StIdle;

// SEC_CM: CFG_SHADOWED.CONFIG.REGWEN
// assign hw2reg.cfg_regwen.d = engine_stable;
assign hw2reg.cfg_regwen.d = 1'b1;
//CWE-1233: regwen is always 1 so kmac can be configured during
operation!!
```

Figure 2: Buggy KMAC Lock Bit Behavior

1.5.2. Bug 2: Persistent SRAM Data

This bug is meant to represent CWE-1272 but is a fairly loose interpretation. Depending on the boot and power-up/down instructions, the program flow and what must be cleared will change so it is difficult to identify sensitive assets without knowledege of the programs. Instead, a generic concern might be the clearing of memory that will at some point hold sensitive data, such as the SRAM. In the OpenTitan SoC, the data in the SRAM is scrambled and the data effectively becomes "invalid" when the scrambling key is renewed (the data cannot be unscrambled). Once the key is renewed, an initialization can also be requested to write the SRAM with pseudo-random data. This functionality is crucial to confidentiality of the SRAM data, as these operations are often done in and out of state transitions. A section of the implementation in OpenTitan is shown in Fig 3. This code snippet was taken from `sram_ctrl`, the SRAM Controller. This controller is the interface between the rest of the system and any SRAM memory. The `always_ff @ (posedge clk) or negedge rst_ni` block signify that the signals inside are registers. The `key_req_pending_q` gets asserted when a request for a new key is issued and `key_ack` gets asserted when the new key has been provisioned, which subsequently updates the key and nonce. On reset, `key_req_pending_q` is cleared and the key and nonce are set to constants defined by parameters. If a new key is never requested, the constant key will persist. A simple but meaningful modification is to modify the assignment to `key_req_pending`. For example, completely removing the assignment, as shown in Fig. 4, will make it stay at 0. We can also extend this by adding a bug to the initialization function so that the data cannot be cleared. This is shown in Fig. 5. Together, these ensure that previous data always be unscrambled, and that sensitive information cannot be wiped.

```
always_ff @(posedge clk_i or negedge rst_ni) begin : p_regs
  if (!rst_ni) begin
    key_req_pending_q <= 1'b0;
    // reset case does not use buffered values as the
    // reset value will be directly encoded into flop types
    key_q             <= RndCnstSramKey;
    nonce_q           <= RndCnstSramNonce;
  end else begin
    key_req_pending_q <= key_req_pending_d;
    if (key_ack) begin
      key_q   <= key_d;
      nonce_q <= nonce_d;
    end
    // This scraps the keys.
    // SEC_CM: KEY.GLOBAL_ESC
    // SEC_CM: KEY.LOCAL_ESC
    if (local_esc) begin
      key_q   <= cnst_sram_key;
      nonce_q <= cnst_sram_nonce;
    end
  end
end
```

Figure 3: Original SRAM Key Request

```
always_ff @(posedge clk_i or negedge rst_ni) begin : p_regs
  if (!rst_ni) begin
    key_req_pending_q <= 1'b0;
    // reset case does not use buffered values as the
    // reset value will be directly encoded into flop types
    key_q             <= RndCnstSramKey;
    nonce_q           <= RndCnstSramNonce;
  end else begin
    // key_req_pending_q <= key_req_pending_d; CWE-1272
    if (key_ack) begin
      key_q   <= key_d;
      nonce_q <= nonce_d;
    end
    // This scraps the keys.
    // SEC_CM: KEY.GLOBAL_ESC
    // SEC_CM: KEY.LOCAL_ESC
    if (local_esc) begin
      key_q   <= cnst_sram_key;
      nonce_q <= cnst_sram_nonce;
    end
  end
end
```

Figure 4: Buggy SRAM Key Request

```
logic init_d, init_q, init_done;
assign init_d = (init_done) ? 1'b0 :
                (init_trig) ? 1'b1 : init_q;

always_ff @(posedge clk_i or negedge rst_ni) begin : p_init_reg
  if(!rst_ni) begin
    init_q <= 1'b0;
  end else begin
    // init_q <= init_d; - CWE-1272
  end
end
```

Figure 5: Buggy SRAM Initialization Request

1.5.3. Bug 3: Unwritable Flash Memory

As discussed in section 1.4.12., a potential introduction of CWE 1277 into a design is through the inability to write to Flash memory. If we assume that there are no defects in the memory itself, any denial of service would originate from the Flash controller. The flash controller of the OpenTitan SoC is seperated into two "entities". The Flash Protocol Controller interacts with software and other hardware components while the Flash Protocol Controller is responsible for interacting with the memory itself. I focused on the protocol controller since all writes are issued from it. Figure 6 illustrates the original design, and the modification. Initially, `prog_op` is asserted if the incoming operation is a flash program (write) request. The buggy behaviour now incorrectly compares it to a read operation, `FlashOpRead`. This has two effects: (i) the flash controller cannot issue a write when desired, and (ii) there will be contention between the `rd_op` and `prog_op` when a read is desired. For this bug, we are concerned with the first effect.

```
assign rd_op         = op_type == FlashOpRead;
// CWE-1277 - Flash is not writable
// assign prog_op       = op_type == FlashOpProgram;
assign prog_op       = op_type == FlashOpRead;
assign erase_op      = op_type == FlashOpErase;
```

Figure 6: Flash Write Operation Bug

1.5.4. Bug 4: Software-Readable Key Register

This bug is meant to demonstrate that a seemingly CWE (CWE-1276) and "simple" bug can have irreparable consequences. As discussed above, module instantions are crucial to the secure behavior of hardware. Most times, incorrectly connected child modules will have noticeable functional impact that will alert of an issue during verification. On the security side however, it may go undetected unless the security feature which depends on the incorrectly connected port(s) are tested. In the OpenTitan SoC, every memory-mapped register is created as a module instantiation, as shown in Fig. 7. This specific register is meant to hold part of the AES key. As we can see, `.re` is

"hard-coded" to `1'b0`, signifying that is it a write-only register that cannot be read by software. This is crucial to the confidentiality of the system as malicious software could attempt to read the key and leak it. It follows that inverting that bit will make it always readable. This is shown in Fig 8. If this is repeated for every key register, the whole key can be stealthily leaked by software.

```
// Subregister 0 of Multireg key_share0
// R[key_share0_0]: V(True)
logic key_share0_0_qe;
logic [0:0] key_share0_0_flds_we;
assign key_share0_0_qe = &key_share0_0_flds_we;
prim_subreg_ext #(
  .DW    (32)
) u_key_share0_0 (
  .re     (1'b0),
  .we     (key_share0_0_we),
  .wd     (key_share0_0_wd),
  .d      (hw2reg.key_share0[0].d),
  .qre    (),
  .qe     (key_share0_0_flds_we[0]),
  .q      (reg2hw.key_share0[0].q),
  .ds     (),
  .qs     ()
);
assign reg2hw.key_share0[0].qe = key_share0_0_qe;
```

Figure 7: Memory-mapped register for AES Key

```
// Subregister 0 of Multireg key_share0
// R[key_share0_0]: V(True)
logic key_share0_0_qe;
logic [0:0] key_share0_0_flds_we;
assign key_share0_0_qe = &key_share0_0_flds_we;
prim_subreg_ext #(
  .DW    (32)
) u_key_share0_0 (
  .re     (1'b0),
  .we     (key_share0_0_we),
  .wd     (key_share0_0_wd),
  .d      (hw2reg.key_share0[0].d),
  .qre    (),
  .qe     (key_share0_0_flds_we[0]),
  .q      (reg2hw.key_share0[0].q),
  .ds     (),
  .qs     ()
);
assign reg2hw.key_share0[0].qe = key_share0_0_qe;
```

Figure 8: Buggy Memory-mapped register for AES Key

1.5.5. Bug 5: Memory Range Overlap Reversed Priority

As discussed in section 1.4.9., the priority between overlapping memory ranges is given to the lower "index" range in the RISC-V Privileged Specification. This idea is extended in the Flash controller -- the documentation states "Similar to RISCV pmp, if two region overlaps, the lower region index has higher priority". The memory regions in flash memory are configurable via two registers for each region (up to 7). In the first register, `MP_REGION_CFG_X`, the bit fields `EN_X` enables the region, `RD_EN_0` makes the region readable, `PROG_EN_0` makes the region programmable, `ERASE_EN_0` makes the region erasable, `SCRAMBLE_EN_0` makes the region scramable, `ECC_EN_0` makes the region ECC and Integrity checked, and `HE_EN_0` makes the region "high endurance enabled". The second register,

`MP_REGION_X` holds two bit fields, `BASE_X` and `SIZE_0` that configure the base page number and number of pages of the memory region. Both these registers also have `REGWEN` lock bits. Fig. 9 illustrates the memory region checking functionality in OpenTitan. The first `always_comb` block checks for the matching region starting from the lowest region index, 0. The second outputs the appropriate access control for that region, stored in `region_attrs_i[i].cfg`. Reversing this ordering to decrementing from the highest memory region (e.g., `Regions-1`) would break the "agreement" held by the software and hardware and result in improperly managed region overlap, possibly leading to loss of confidentiality, integrity, and availability. This bug is shown in Fig. 10.

```
// check for region match
always_comb begin
  for (int i = 0; i < Regions; i++) begin: gen_region_comps
    region_end[i] = {1'b0, region_attrs_i[i].cfg.base} + region_attrs_i[i].cfg.size;

    // region matches if address within range and if the partition matches
    region_match[i] = addr_i >= region_attrs_i[i].cfg.base &
                      {1'b0, addr_i} < region_end[i] &
                      phase_i == region_attrs_i[i].phase &
                      mubi4_test_true_strict(region_attrs_i[i].cfg.en) &
                      req_i;
  end
end

// select appropriate region configuration
always_comb begin
  sel_cfg_o = '0;
  for (int i = 0; i < Regions; i++) begin: gen_region_sel
    if (region_sel[i]) begin
      sel_cfg_o = region_attrs_i[i].cfg;
    end
  end
end
```

Figure 9: Secure OpenTitan Flash Memory Region Permission Check

```
// check for region match
always_comb begin
  // for (int i = 0; i < Regions; i++) begin: gen_region_comps
  for (int i = Regions-1; i >= 0; i--) begin: gen_region_comps // CWE-1260
    region_end[i] = {1'b0, region_attrs_i[i].cfg.base} + region_attrs_i[i].cfg.size;

    // region matches if address within range and if the partition matches
    region_match[i] = addr_i >= region_attrs_i[i].cfg.base &
                      {1'b0, addr_i} < region_end[i] &
                      phase_i == region_attrs_i[i].phase &
                      mubi4_test_true_strict(region_attrs_i[i].cfg.en) &
                      req_i;
  end
end

// select appropriate region configuration
always_comb begin
  sel_cfg_o = '0;
  // for (int i = 0; i < Regions; i++) begin: gen_region_sel
  for (int i = Regions-1; i >= 0; i--) begin: gen_region_sel // CWE-1260
    if (region_sel[i]) begin
      sel_cfg_o = region_attrs_i[i].cfg;
    end
  end
end
```

Figure 10: Buggy OpenTitan Flash Memory Region Permission Check

1.5.6. Discussion

Unfortunately, I was not able to setup the design verification flow due to challenges in obtaining a VCS license. This meant that my simulation environment would have to be created from scratch, an extensive task for a project of this scale. I was not able to have a platform ready in time.

Table 2 summarizes the 5 bugs inserted into the OpenTitan SoC, the way they were introduced, the number of lines modified, and their functional locations. I also included a theoretical CVSS 3.1 score that demonstrates the potential implications of these bugs. The Common Vulnerability Scoring System (CVSS) captures the characteristics of software, hardware and firmware vulnerabilities and outputs numerical scores indicating the severity of a vulnerability relative to other vulnerabilities. This is a generally accepted and used method to evaluate potential vulnerabilities. Here the scores are only "theoretical" because the bugs are not vulnerabilties. I focused on the *Base Score* as it reflects the severity of a vulnerability according to its intrinsic characteristics which are constant over time and assumes the reasonable worst case impact across different deployed environments. [^1]

[^1]: Definitions obtained from the CVSS Specification

Bug ID	Bug Name	CWE	Mode of Introduction	# of lines changed	Functional Location	Theoretical CVSS Score
B1	Incorrect Lock Bit Behaviour	CWE-1231	Incorrect assignment	1	KMAC	7.8 (CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:H/A:H)
B2	Persistent SRAM Data	CWE-1272	Missing assignment	2	SRAM Controller	6.4 (CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:L/A:N)
B3	Unwritable Flash Memory	CWE-1277	Incorrect comparison	1	Flash Controller	7.8 (CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:H/A:H)
B4	Software-Readable Key Register	CWE-1276	Incorrect assignment	16 (1 x 16 registers)	AES Configuration Registers	7.5 (CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:H/A:N)

Bug ID	Bug Name	CWE	Mode of Introduction	# of lines changed	Functional Location	Theoretical CVSS Score
B5	Memory Range Overlap Reversed Priority	CWE-1260	Incorrect looping	1	Flash Controller	7.8 (CVSS:3.1/AV:L/AC:H/PR:L/UI:N/S:C/C:H/I:H/A:H)

The CVSS scores for the 5 bugs considered worst-case scenarios. The scores consider the fact that the attacker requires extensize knowledge of the design (Attack Complexity is High), no non-malicious user interaction is required (User Interaction is Low), and all are software-exploitable (Privileges Required is low and Attack Vector is local as I ignore attacks over a network). Bug B3 is a bit unique in that, it can never be considered a vulnerability itself. The scenario I considered was a hardware vulnerability that should be patchable by firmware is now unpatchable.

1.6. HLS-Induced Security Weaknesses

High-Level Synthesis (HLS) is a design process which takes in an algorithm specification, in the form of High-Level Language (C/C++/SystemC) code, and translates it into a functionally equivalent cycle accurate RTL design. HLS shifts the abstraction one level higher to assist designers in creating and verifying (through co-simulation) designs faster. It provides directives or pragmas that give fine-grained control over optimizations and design decisions (pipelining, unrolling, etc.). This allows designers to explore the design space and establish tradeoffs more efficiently. Part of my work this semester, in collaboration with a Ph.D. student, involved investigating the potential security concerns that HLS can introduce into hardware designs. A previous work presented these possible concerns and our aim was to analyze them, determine their root cause (design patterns vs tool optimizations), and develop tools to detect/correct them automatically. It is important to mention that HLS-induced weaknesses are not a result of "incorrect" translation from HLL to RTL. Instead, they are either a result of the different intrinsic behaviors between software and hardware or the design optimizations that are enabled by default or user-enabled. A manuscript was written as a result of this work and is currently under review for publication in an IEEE/ACM conference proceedings. I will summarize the work below but the full paper is provided in the Appendix for reference.

1.6.1. Security Weakness: Intermediate Result Leakage

One of the security weaknesses discussed in the previous paper was the leakage of intermediate values. Non-trivial computation in hardware is usually pipelined to reduce the clock frequency of the system. For example, for AES, instead of computing all 10/12/14 rounds in 1 clock cycle, an alternative would be to compute 1 round per cycle and store intermediate results to be used in subsequent cycles. It was discovered that under specific conditions, these intermediate results would be observable at the module output. An example of this behaviour is shown in Fig ?. The leakage of these values (CWE-203) can have disastrous effect in security-critical IP, such as AES cores, where cryptanalysis becomes significantly easier given intermediate results. We initially conducted experiments to determine if the HLL code would indicate the presence of this weakness but there were too many factors to consider (i.e., C++ compiler, HLS tool algorithms, optimizations, etc.). This meant that it was practically impossible to correlate design patterns to the resulting RTL design concretely. Instead, we found it more practical to detect the issue directly from the RTL design. We took two approaches to this, static analysis and formal verification. The formal verification was used as a more rigourous check to determine the accurary of the static analysis and gain some insight on the performance benefits of using static analysis vs "traditional" verification methods.

We developed multiple buggy IP designs using HLS for our experimentation. These IP designs were intentionally written to identify the causes of introduction of the weakness, and the designs themselves can be considered hardware designs with "injected bugs", staying in theme with the rest of the project. For brevity, I will only discuss the simplest design we implemented, a fibonacci sequence calculator. The C++ code for this design is given as:

```
void fib(short int n, int *ret){
 int a = 0, b = 1, c, i;

 for (i = 2; i <= n; i++) {
     c = a + b;
     a = b;
     b = c;
 }
 *ret = b;
}
```

This is a simple function which computes the nth number of the fibonacci sequence and updates the value of pointer `ret` to the result. In this representation, this appears "secure" -- `ret` is only updated when the computation is complete due to the sequential execution of

software. In the hardware representation however, an issue arises, as shown in Fig. ?. Again, the intention of the design is to only update `ret` when the final value is calculated, in this case, `b=8`. However, we can see that ret is updated at every computation, starting from 2 all the way to the final result, 8, and `ap_done` is asserted.

Figure ?: Insecure Fibonacci Sequence Calculator Waveform

1.6.2. Automatic Detection: Static Analysis

We created a static analysis scanner that parses Verilog designs into Abstract Syntax Trees (ASTs) and processes them using the Visitor pattern. The visitor pattern is a popular design pattern to seperate algorithms from data structures, resulting in the ability to add new functionality without modifying the original source code. We heuristically determined the following patterns. First, the block-level outputs that are in interested in are always assigned using continuous assignment (`assign` keyword in Verilog/SystemVerilog). Second, register signals generated by the HLS tool we used always end with the postfix `_reg`. Finally, leakage occured when the signal being assigned to the output was not from a register. This inuitively makes sense as non-registered signals are always updated. We combined these patterns to create a scanner which visits all `assign` statements, checks if a relevant signal in its left-hand side and checks if the signal being assigned ends with `_reg`. If not, we can assume that passthrough is present. This is a simple, and as we discovered, error-free solution because HLS implements syntactically identical patterns for functionally similar designs thanks to its template-based nature. It can be considered the "perfect designer" in that sense. The scanner was implemented using Pyverilog and Python and tested on multiple synthetic and realistic benchmarks.

1.6.3. Automatic Detection: Formal Verification

A formal verification automated flow was also created to verify the results of the scanner. Formal verification provides a mathematically exhaustive exploration of the state space to determine if undesired and/or desired behaviour occurs. Specifically, we used SystemVerilog Assertions to specify the desired behaviour. The critical part of using formal verification is understanding the behaviour being defined in the properties and being to specify exactly the desired requirements. A *safety property* is one that specifies that "something bad doesn't happen". For example, "the output of this module never changes unless the block operation is complete". A *liveness property* ensures that "something good eventually happens". For example, "the output must eventually change if the input changes". Once the property is defined, there are also nuances between property assertions and covers. *Assertions* ensure that the property always holds true, e.g., "this register must **always** go to 0 if reset if high". *Covers* ensure that it is possible for the property to be true, e.g., "it must be possible to go from state A to state B". There are also constraints (`assume` statements) that are used to reduce the state space, typically to remove false positives and/or improve performance.

1.6.3.1. Property Formulation

In this scenario, we are attempting to verify that the output of the block **must** only update when the block operation is complete. "Must" indicates that this property will be asserted and the property is ensuring that something bad (i.e., the output updating prematurely) doesn't happen, so it is a safety property. Next, we must define the desired behaviour. We again take advantage of the consistency of HLS-generated blocks and the top-level control signals our HLS tool adds by default. Specifically, it will always add a `done` signal that is asserted for 1 cycle when block-level operation is completed. We can use this knowledge to define a property as shown in listing 1. The property states that a change in `in`, followed by a change in `out` after an indeterminate number of cycles implicates that `done` must be asserted in the same clock cycle. The `done` signal is asserted when the end of the block-level function is reached and de-asserted a clock cycle later. This specifies our desired behavior that the output should only change when the operation is complete.

```
assert property(
  always @ (posedge clk) disable iff(rst)
  $changed(in) ##[1:$] $changed(out) |-> $rose(done)
);
```

Listing 1: No Passthrough SVA Property

1.6.3.2. Verification Environment Generation

We automated the formal verification process by automatically generating properties and verification modules. This process begins by first obtaining the top-level module name and top-level I/O signal names. A verification module is then created using this information -- the relevant top-level IO is added to the verification module's IO. For each unique input-output pair, a new property is instantiated inside of the verification module. Finally, a `bind` statement is added to a "bind file" using the top-level module name. The names of all of the new files and existing design files are then used to create a TCL script that completes the setup and verification when called with Cadence JasperGold, the formal verification tool we used.

1.6.4. Automatic Correction: Directive Generation

Two conditions must be met to mitigate the passthrough weakness: (i) a registered output, and (ii) appropriate control logic. The registered output is necessary to separate the intermediate output net to the top-level output net. The control logic enables the added register only when the operation is complete. We intuited that weaknesses can be remediated using the directives of an HLS tool. The idea being that after we detect a weakness, we can add the required directive(s) to the corresponding file of the HLS project. After re-running the synthesis, we can scan the generated design again to validate that the weakness has been fixed. If it is still present, an error message is raised to get the attention of the designer for manual analysis. We investigated the documentation of a typical commercial tool, as an example, identifying three candidate solutions:

1. **`set_directive_interface [OPTIONS] <location> <port>`** specifies how a function interface is synthesized. This directive provides port-level granularity (i.e. the ability to specify a specific port of a specific function) using `<location>`and `<port>`. Using this directive as a solution requires enabling the `-register` option and adding control logic using `-mode ap_hs|ap_ack|ap_ovld|ap_vld`.
2. **`config_interface [OPTIONS]`** is a configuration command applied at the solution-level. This controls the default IO interface synthesized by the HLS tool for each function. We use this command with the `-register_io scalar_out|scalar_all` option. This specifies that all scalar outputs must be registered. There is no option to specify the mode of the registers.
3. **`config\_rtl [OPTIONS]`** is a configuration command. Using it with the `-register_all_io` option is similar to the configuration above but does not provide the ability to only modify outputs.
   Similar directives exist for other tools.

In this specific instance, we propose using the `config_interface -register_io scalar_out` directive. `set_directive_interface` provides the most control but we found experimentally that it was not always effective. It also requires manually specified security-critical outputs and introduces the possibility of error. `config_interface` and `config_rtl` are functionally similar but the former provides the control to only register the outputs. It allows for a context-free approach, not requiring any signal names. For secure IP like cryptographic accelerators, no output should be updated intermediately.

1.6.5. Experimental Results and Discussion

The results of the experiments are given in Table 1.

Table 1: Experimental Results

DESIGN	**LOC (.C)**	**TCL**	**LOC (.V)**	**Vulnerability Present**	**Design Characteristics**			**Time**		
					Latency	FF	LUT	Scanner Parse	Scanner Scan	Formal
Factorial	9	DEFAULT	597	N	-	104	172	1.224996838	0.001545164	0.028
		REG	497	N	-	171	177	1.222182221	0.001969594	0.02
Fibonacci	11	DEFAULT	314	Y	-	98	159	1.205799277	0.000912264	0.02
		REG	249	N	-	181	144	1.218891101	0.001232714	0.02
Combined	23	inline off	1067	Y	-	255	352	1.241450036	0.001459871	0.017
		inline off + reg	928	N	-	388	353	1.210921549	0.001183	0.033
		REG	786	N	-	338	313	1.256323047	0.001575409	0.045
		EMPTY	922	N	-	206	307	1.230187869	0.002145791	0.06
PRESENT	205	ALL	5405	Y	73	539	2105	1.323337347	0.005241983	0.073

DESIGN	LOC (.C)	TCL	LOC (.V)	Vulnerability Present	Design Characteristics			Time		
		REG	5535	N	110	808	2135	1.362591188	0.005700883	11.21
		PIPELINE	4488	Y	205	681	1697	1.357170563	0.005143854	0.048
		UNROLL	3463	Y	130	766	1598	1.317258597	0.004412583	0.069
		EMPTY	1717	Y	5956	659	2340	1.316133946	0.003116776	9.391
serpent	331	ALL	4129	Y	36	667	1543	1.242949633	0.002426735	0.165
		REG	5303	N	337	4433	1766	1.275816957	0.003447497	22.859
		PIPELINE	4129	Y	36	667	1543	1.274372542	0.002633463	0.187
		UNROLL	4174	-	98	1430	1532	1.241780602	0.002217242	-
		EMPTY	2574	Y	21944	4434	5964	1.248128677	0.002531933	7.513
AES	372	ALL	6751	Y	532	1266	5860	1.286926845	0.006199542	0.088
		REG	7038	N	563	1983	5953	1.298245312	0.006208187	2.689
		PIPELINE	6325	Y	584	1629	11346	1.31156072	0.006155954	0.085
		UNROLL	4516	Y	1048	1453	6343	1.307233934	0.005468802	0.097
		EMPTY	10535	Y	10542	2015	15082	1.303846266	0.005323912	0.068
AES (vitis lib)	303	ALL	23454	Y	60	14301	8122	1.27516172	0.004774896	28.638
		REG	23480	N	64	16203	8134	1.36882548	0.005540823	2.165
		PIPELINE	23454	Y	60	14301	8122	1.291424441	0.004844537	28.749
		UNROLL	3321	Y	484	1300	1646	1.272824068	0.003498296	1.813
		EMPTY	3120	Y	10844	2257	27440	1.262695441	0.002619167	0.77

These results show that the weakness we are concerned about, passthrough is present in some cases in the synthetic examples, and all the time in the real examples, without the added correction directive. They also show that the proposed correction directive is a valid solution and all generated designs using the directive did not indicate the presence of the weakness. The most interesting result is that the scanner was 100% accurate when compared to the formal verification results, while being much more efficient for larger designs. This promising result is thanks to the "robotic" nature of the HLS-generated designs. These results should motivate the development of more scanners that target HLS-generated designs to catch security concerns that may go undetected otherwise, all while achieving low (or zero) false positives and low time investment.

1.7. Conclusion

For this project, I studied the Hardware CWEs and identified the CWEs that are ammenable to injection at the RTL design stage. I also discussed the locations and operations that can result in errors and insecure behavior. I then implemented bugs in the OpenTitan Root-of-Trust SoC to demonstrate how easily they can get introduced into designs and discuss their impact. I was not able to verify their behaviour due to technical difficulties. Finally, I explored CWEs introduced due to High-Level Synthesis and detailed a research project I collaborated on with another research group.

1.8. Appendix A: OpenTitan

The OpenTitan SoC homepage can be found here, the documentation here, and the GitHub repository containing all source code here. OpenTitan is an open-source Root-of-Trust (RoT) SoC maintained by lowRISC and Google. It is the only open-source RoT currently available, making it an interesting case study for this assignment as it contains extensive security features and documentation. It implements various cryptographic hardware, such as the Advanced Encryption Standard (AES), HMAC, KMAC, and security countermeasures like access control to ensure the Confidentiality, Integrity, and Availability (CIA) of its functions.

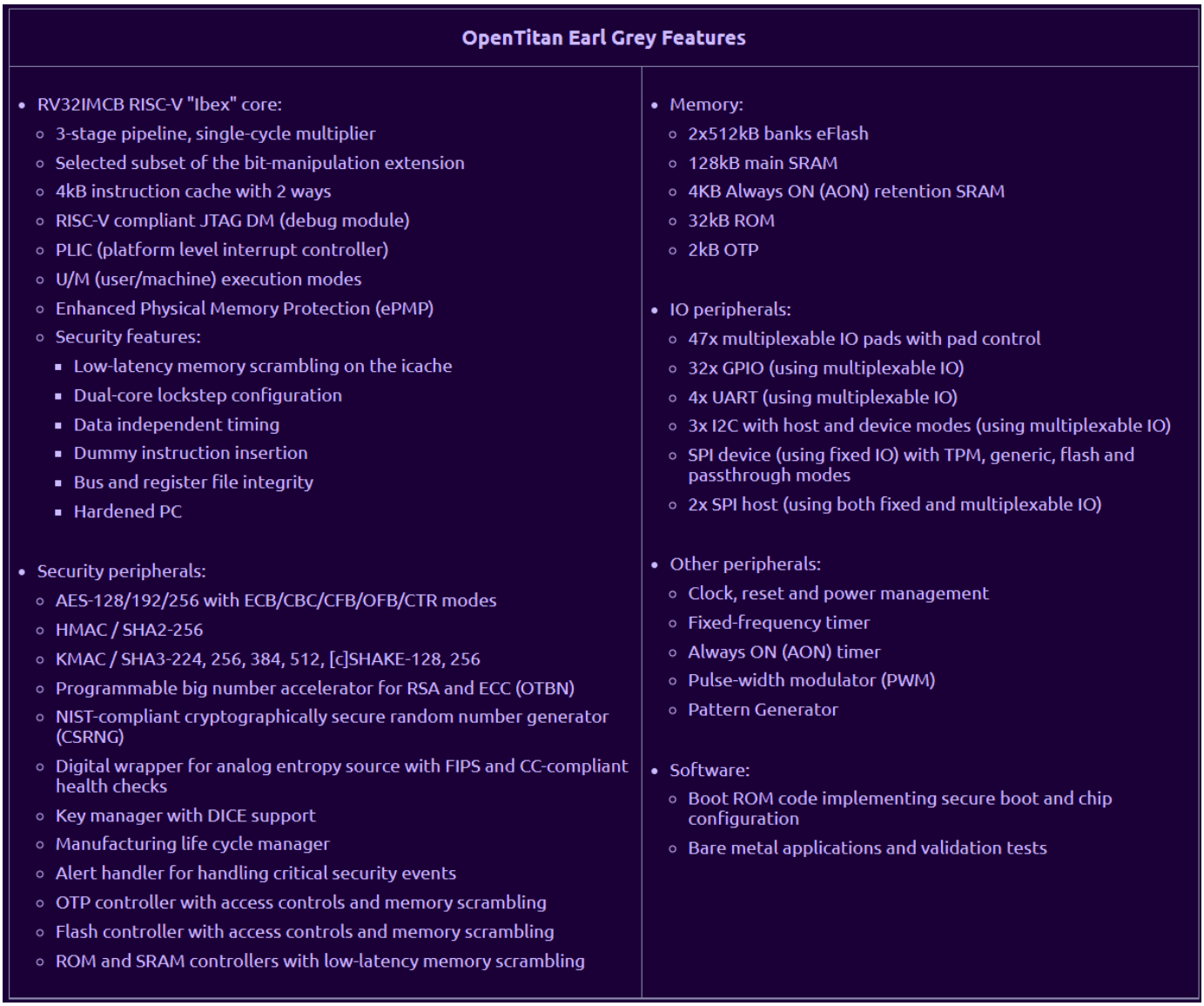

Figure 1: OpenTitan Features

The OpenTitan project has well defined and documented threat models and countermeasures. They outline the secure assets, adversary, attack surfaces, and attack methods. The assets are mainly centered around the cryptopgraphic keys, with other loosely defined statements such as "Integrity and authenticity of stored data".

The adversaries they consider are (i) a bad actor with physical access to the device during fabrication or deployment, (ii) a malicious device owner, (iii) malicious users with remote access.

1.8.1. Architecture

The OpenTitan SoC's architecture follows the standard Network-on-Chip (NoC) design paradigm, with various IP cores interconnected a high-speed communication protocol allowing them to communicate with one another. The processor is able to configure and use the pheripherals by writing and reading to memory-mapped IO registers.

The interconnect responsible for connecting all IP cores is a TileLink Uncached Lightweight (TL-UL) crossbar which is autogenerated using a custom crossbar generation tool. The top-level module dubbed *Earl Grey*, is also auto-generated using a top generation tool. Both tools are configured by using hjson files that are scattered throughout the project.

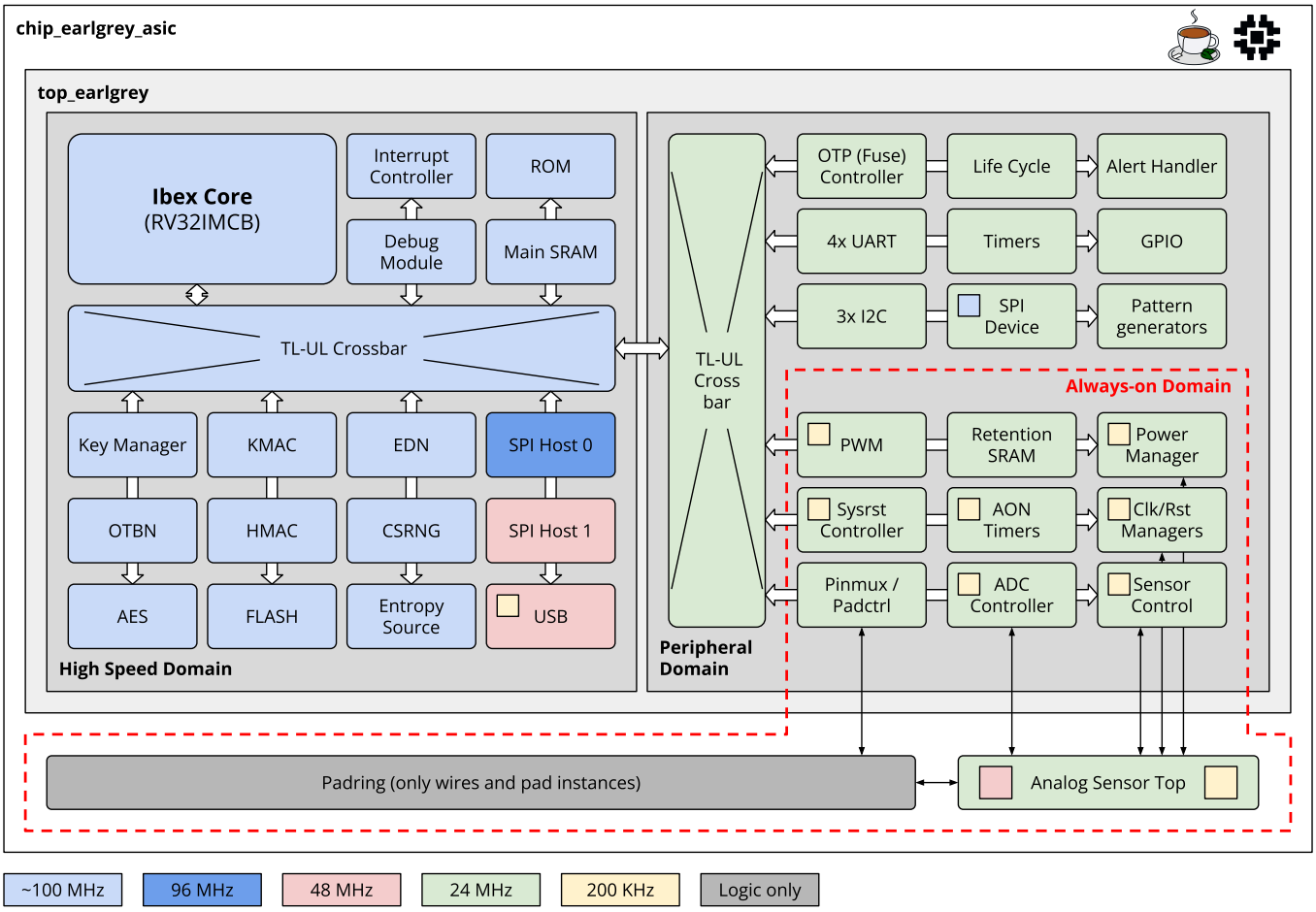

Figure 2: OpenTitan EarlGrey Top

The memories are integrated in the chip with configurable size and address. By default, the instruction ROM is 32 kB, the flash is 1024 kB, and SRAM is 128 kB. The processor core used is the RISC-V Ibex core which we discuss here. As seen in Figure 1, the SoC is seperated into high speed and peripheral domains, with many of its critical functions residing in the high speed domain.

It also provides bebug functionality by way of the RISC-V debug specification 0.13.2 and the JTAG TAP specification.

1.8.2. Security Features

As a RoT, the OpenTitan SoC implements various security features. Outside of its secure cryptopgraphic functions, it also provides a secure boot flow that integrates multiple memory integrity checks, various access control measures such as lock bits for pheripheral configuration registers and memory regions, an integrity scheme integrated into the TL-UL crossbar, and security alerts that are triggered under defined conditions that suggest suspicious behaviour.

There is curently no detailed documentation for the secure boot flow available, but at a high level, on boot-up the hard-coded instructions in the ROM memory are used for platform checking and memory integrity checking. At this stage, the integrity of the full contents of the non-hard-coded bootloader in the Flash memory is checked by an RSA-check algorithm.

Another fundamental piece of memory which is not directly mentioned in the secure boot process is the one time programmable (OTP) memory. An OTP controller is provided but the OTP IP (fuse memory) must be source externally. Together, they provide secure one-time-programming functionality that is used throughout the life cycle (LC) of a device. The OTP is broken up in partitions responsible for storing different attributes of the device. The specific attributes for each partition (and the partition themselves) are configurable and will likely vary widely for different applications. Critical data stored in the OTP include the root keys used to derive all other keys for cryptographic functions and memory scrambling.

The end-to-end cross integrity scheme consists of additional signals embedded into the interconnect that ensures the integrity of data as it travels through the SoC. There is no detailed documentation on its operation yet. From what is available -- the integrity scheme is an extension of the TL-UL specification and consists of 2 additional `SystemVerilog buses` that carry the "integrity" of the data, which is checked by the consumer. From inspecting the design, the integrity scheme utilizes uses Hsiao code (modified version of Hamming code + parity) as its error-detection code.

On the cryptographic side, the relevant IPs comprise of the Key Manager, KMAC, HMAC, AES, the Entropy source, EDN, and CSRNG. The key manager is responsible for generating the keys used for all cryptographic operations and identification. On reset, it rejects all software requests until it is initialized again. Initialization consists of first loading in random values from the entropy source then the root key from the OTP. This ensures that the hamming delta (the difference in hamming weights between the random number and the root key) are non-deterministic and the root key is thus not susceptible to power side-channel leakage **(This is my interpretation, I am probably wrong)**. The key manager interatively completes KMAC operations using the KMAC IP to progress to different states and generate different keys. The states transistions of the Key Manager are illustrated in Figure 3. The Key manager implements various security countermeasures such as sparse FSM encoding, and automatic locking of configuration registers during operation.

The Keccak Message Authentication Code (KMAC) IP core is a Keccak-based message authentication code generator to check the integrity of an incoming message and a signature signed with the same secret key. It implements the NIST FIPS 202 SHA-3 standard. The secret key length can vary up to 512 bits. The KMAC generates at most 1600 bits of the digest value at a time which can be read from the STATE memory region. It also implements masked storage and Domain-Oriented Masking (DOM) inside the Keccak function to protect against 1st-order SCA attacks. As mentioned earlier, the KMAC core is used extensively by the key manager. Its security countermeasures include sparse FSM encoding, counter redundancy, and lock bits to ensure configuration registers are not written during operation.

The Keyed-Hash Message Authentication Code (HMAC) IP Core implements the SHA256 hashing algorithm. It achieves similar functions to the KMAC core but is not hardened against power side-channels. It is meant as a faster alternative to the KMAC core. It does not contain any security countermeasures other than the bus integrity scheme present in all IP.

The final cryptographic core is the AES accelerator responsible for all encryption/decryption operations of the SoC. It implements NIST's Advanced Encryption Standard. It supports multiple standard block modes of operation (ECB, CBC, CFB, OFB, CTR) and 128/192/256-bit key sizes. The accelerator implements the same masking scheme as the KMAC core to protect itself against 1st order side-channel attacks. It also implements many other security countermeasures: lock bits, clearing of sensitive registers after operation, sparse FSM and control register encoding, and logic rail redundancy for FSMs.

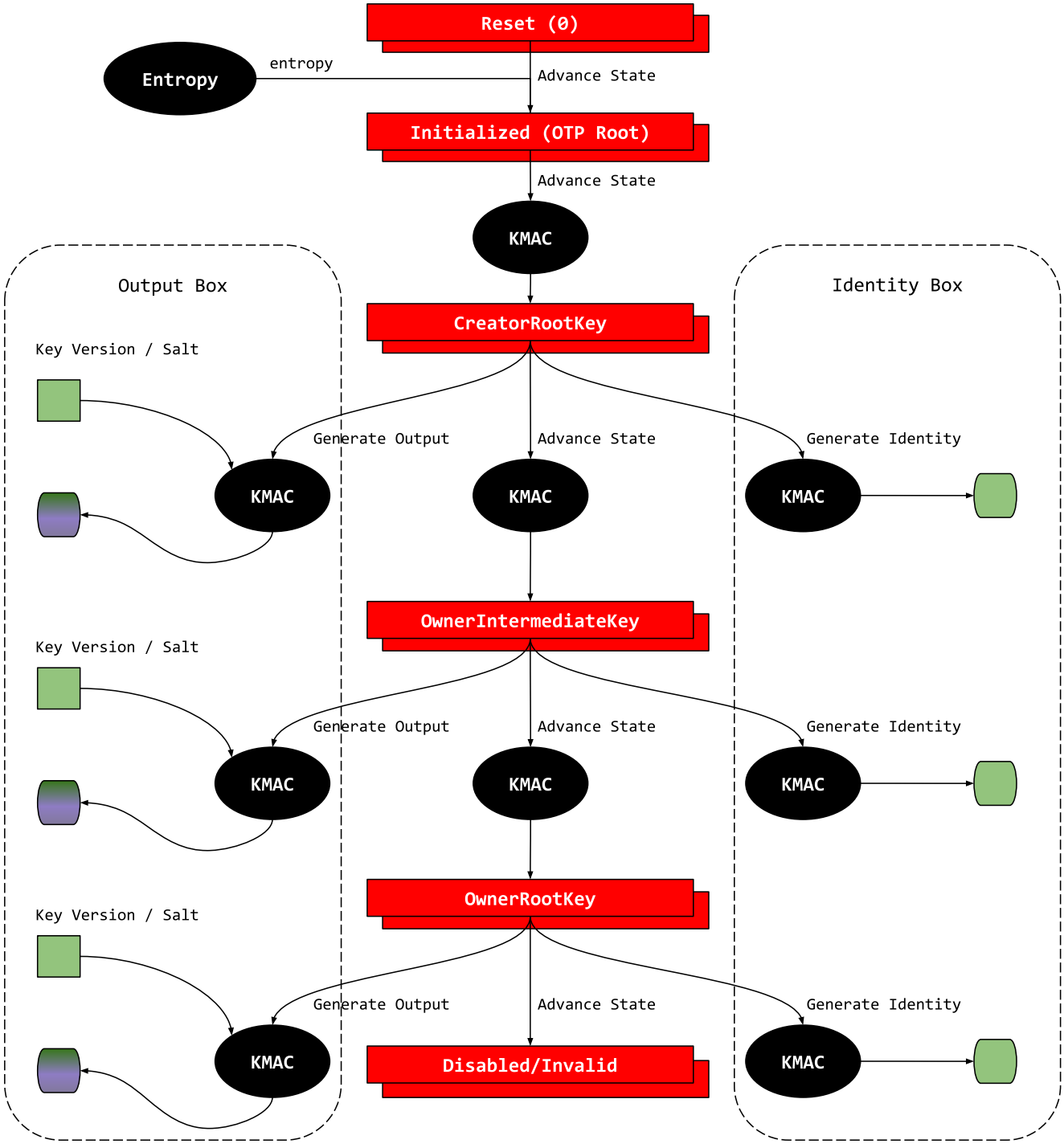

Figure 3: Key Manager State Transitions

Finally, the ROM, SRAM, and Flash controllers manage accesses to memory. They integrate multiple security features.

The ROM controller contains a startup checker which verify the integrity of its contents by utilizing the KMAC IP to hash all but the 8 top words of its data. The hash received from the KMAC operation is then compared to the 8 top words. The read addresses are passed through a substitution and permutation (S&P) block then passed to the ROM memory and a PRINCE cipher in parallel. The pre-scrambled data read from the ROM is also passed through an S&P block, and XOR from the results of the PRINCE cipher to obtain the final read data.

The data in the SRAM is also scrambled in similar fashion to the ROM, and additionally contains 7 integrity bits for each 32 bit word. It also provides a Linear Feedback Shift Register (LFSR) block to feature that can overwrite the entire memory with pseudorandom data via a software request.

The flash controller provides also optional memory scrambling and integrity bits. It also provides up to software-configurable 8 memory regions with configurable access policies.

1.8.3. Collateral

The OpenTitan SoC provides extensive collateral. Collateral in this context, refers to any additional information that describes the functionality of a design and its components. The collateral for this SoC consists of the documentation for all of its IP and contains its security features, interfaces, interactions with software, testplans, and block diagrams. Unique to this SoC are the hjson files that describe all of an IP's parameters, registers, security countermeasures, etc. This is extremely useful to obtain designer context behind the design. For example, from the AES hjson file, we can understand the function of parameter `SecMasking`, as shown in figure 4.

```
{ name:    "SecMasking",
  type:    "bit",
  default: "1'b1",
  desc:    '''
    Disable (0) or enable (1) first-order masking of the AES cipher core.
    Masking requires the use of a masked S-Box, see SecSBoxImpl parameter.
  '''
  local:   "false",
  expose:  "true"
},
```

Figure 4: AES SecMasking .hjson snippet

Another aspect of collateral is the test environment provided. OpenTitan currently provides automated Dynamic Verification (DV) for all IP which perform simulate the IP and perform automated checks using a Golden Reference model. They also an FPV test suite using SystemVerilog Assertions which mainly verify the compliance to the TL-UL protocol. The SoC was setup locally with relative ease, thanks to the detailed instructions and reliable scripts, and the UVM tests were successful run using Verilator.

Ache-Less: An Extensible Framework for HLS with Less Security Pain

Author(s)
dept. name of organization (of Aff.)
name of organization (of Aff.)

***Abstract*—Due to the increasing complexity of modern integrated circuits, High-Level Synthesis (HLS) is becoming a key technology in hardware design. HLS uses optimizations to assist during design space exploration. However, some of them can introduce security weaknesses. We propose a design framework that leverages static analysis to identify a class of weaknesses in HLS-generated code and correct them through the automatic generation of HLS directives. We evaluate our framework by comparing the static analysis results with formal verification. Our results show that the static approach has the same accuracy as formal methods while being 3× to 200× faster.**

I. Introduction

Systems-on-Chips (SoCs), which integrate multiple Intellectual Property blocks (IP), have increased computer system complexity. Design and verification methodologies and tools have not evolved and scaled as much as the size and complexity of SoC designs [1]. It has thus become increasingly difficult to properly design and verify hardware while meeting time-to-market commitments. High-Level Synthesis (HLS) aims to alleviate the design bottleneck by raising the abstraction level to the algorithmic one [2]. HLS enables designers to start with algorithms expressed in a High-Level Language (HLL) and explore the space of corresponding hardware designs using HLS tool *directives* or source-code *pragmas*. Designers can explore the large design space offered by HLS and evaluate Power, Performance, and Area (PPA) trade-offs, with recent work proposed to help designers explore the large space of configurations [3].

However, current HLS tools generate the output Register-Transfer level (RTL) designs without considering security. Starting from an algorithm and using different HLS directives, one can obtain different results in terms of latency, resources, and weaknesses, as illustrated in Fig. 1. Designers focusing only on traditional metrics can introduce unintentional security weaknesses, thus causing security pain [4].

HLL specifications can map to various RTL implementations depending on the used HLS directives, so it is hard to reason about weaknesses by examining only the high-level code. Moreover, commercial HLS tools are very complex and can be only seen as black boxes, so it is hard to correlate tuples of high-level code and HLS directives to the generated RTL code. Note also that HLL specifications are often human-written and similar algorithms and optimizations can be expressed in myriad ways. Working at the RTL allows us to analyze the outputs of HLS tool and to take advantage of its regular structure.

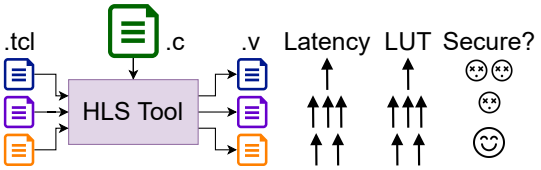

Fig. 1. Given a C file and directives (.tcl files), one can achieve different Latency, Resources and Weaknesses results. We aim to find certain security weaknesses stemming from HLS and remediate them.

With hardware security concerns driving regulations [5], we need security-conscious HLS tools and verification frameworks to systematically assess design security, even when designers are not security experts. Traditional security verification techniques rely on security properties created in an ad hoc manner by security and formal experts [6]. However, a pain point in the design flow is that the reliance on such expertise does not *scale* with the increasing adoption of HLS by non-hardware experts.

Thus, we propose a framework to detect and correct weaknesses early in the design process. We investigate static analysis to detect weaknesses in HLS-generated RTL and propose an automated repair flow to correct detected weaknesses by generating HLS directives that enable changes to the RTL without modifying input code. We evaluate our work on synthetic benchmarks that highlight how similar high-level patterns can yield different RTL designs and conduct a case study analyzing three block ciphers (AES, serpent, and PRESENT from [7]) to explore the weaknesses that can be introduced by an industrial HLS tool. We show that we can detect these weaknesses automatically and provide fixes to the designers. We validate the framework by implementing an automated property generation flow for formal verification. Our contributions are twofold:

- A framework for the detection and correction of weaknesses in HLS generated designs (Section III).
- A prototype implementation with automatic checking using a formal verification tool that we evaluate experimentally (Section IV).

Our results show that static analysis is effective on HLS-generated code, with all weaknesses being identified by our static scanner. Its runtime outperforms the formal verification tool by a factor of 3× to 200× and scales better with increasing design size.

II. Background

A. High-Level Synthesis

High-level synthesis (HLS) comprises methods to automatically convert a high-level software specification into a corresponding RTL description [8]. The resulting component is a specialized IP block tailored to execute the specific functionality. The HLS process is based on state-of-the-art compilers (e.g., LLVM or GCC), which extract a high-level representation of the functionality and assigns the corresponding operations to time (scheduling) and space (allocation and binding) to determine the final microarchitecture. To simplify the process, especially for register allocation, modern HLS tools are based on the Single Static Assignment (SSA) representation that requires each variable to be assigned once and only once. Existing variables in the original representation are split into *versions*, which are new variables defined by the original name and a progressive number. In this way, every definition produces its own version. This form creates shorter lifetimes for the variables. However, branches or loops create deviations in the control flow. This requires merging the versions coming from different paths. HLS compilers use ϕ operations, which define new versions every time multiple versions are potentially coming from different paths [9].

B. HLS-Induced Security Weaknesses

Recent work has begun to consider the potential for security issues induced by HLS. For example, Pilato et al. [10] and Basu et al. [11] examined the possibility of Hardware Trojan (HT) insertion through compromised HLS tools. HLS is a prime candidate for HT insertion since it is difficult to correlate the HLL description to the RTL. HTs can be partially remedied by equivalence checking for mismatches between HLL and RTL. To that end, [12] propose a C-to-RTL equivalence checking framework by extracting the RTL-level finite-state machine with datapaths (RTL-FSMDs).

A more pressing concern is the *unintentional* introduction of security weaknesses in HLS-generated designs, as discussed in recent work by Pundir et al. [4]. Unlike HTs, weaknesses are not deliberate modifications and equivalence checking might not detect them. Instead, weaknesses are flaws in the design that may lead to vulnerabilities if exploited. MITRE maintains a list of known weaknesses, the Common Weakness Enumeration (CWE) at [13]. Some vulnerabilities identified by Pundir et al. [4] map to the following CWE categories:

- CWE 1245: insecure finite state machine;
- CWE 1300: improper protection of side channel;
- CWE 1271: uninitialized value on reset for registers holding security settings;
- CWE 1189: improper isolation of shared resources on system-on-a-chip;
- CWE 203: Observable discrepancy.

These weaknesses may be a result of patterns in the HLL description, optimizations done by an HLS tool, or a combination. Their root cause is often hard to identify, making them difficult to detect and correct, especially when designers have little knowledge of *secure* hardware design. Due to these challenges, developing techniques to automatically detect and correct these weaknesses is crucial. Of the potential weaknesses identified in prior work [4], we argue that passthrough primary outputs (CWE 203) are the most concerning as it enables other weaknesses like uninitialized values on reset or unbalanced pipelines (CWE 203) to leak sensitive information through primary outputs. Thus, we focus on the passthrough weakness. The passthrough weakness consists in leaking intermediate states through primary outputs. This is a security flaw in all those ciphers like AES that guarantee security only at the final result of the computation and can be broken if intermediate states are accessible. The passthrough weakness could also leak information from other weaknesses through reset and flush if other weaknesses are present.

C. RTL Static Analysis

Our work is positioned at the junction of static analysis for weakness detection and security weaknesses in HLS-generated designs. Prior work [14] showed the potential of using static analysis of RTL to identify weaknesses, especially for designers that might not have security expertise. Their work targeted human-written RTL and addressed the challenge of identifying patterns that indicate possible weaknesses, even when designs assume different shapes. Furthermore, static approaches can support early detection and fixes, reducing the burden of challenging-to-perform formal verification. They rely on heuristically-determined keywords that suggest functionality (e.g., `clk` is the clock signal). In contrast, we focus on HLS-generated code which allows us to take advantage of the template-based nature of designs generated by commercial HLS tools. In using commercial HLS tools, we observe that control signals are named consistently by default and functionally similar patterns are syntactically identical. There is an opportunity for accurate and practical static analysis of the RTL code for HLS-generated designs, since it is likely that there are fewer unique weakness patterns that must be heuristically determined. Thus, we propose a framework for automatically detecting and correcting potential security weaknesses and explore this framework.

III. Detection and Correction Framework

In light of the potential introduction of security weaknesses during HLS, we propose a framework to detect/correct weaknesses in HLS designs, illustrated in Fig. 2. The framework takes a C/C++ design, a set of directive files D for optimizations (e.g., in `tcl`), and parameters specifying the weaknesses to scan for and environment settings. An HLS engine is run for each directive file $d \in D$ obtaining tuples (synthesized design, report) H, $|H| = |D|$ (e.g., as `.v` and `.xml` files, respectively). Each tuple is fed to the RTL scanner which analyzes the Verilog for weaknesses. The scanner uses HLS report to identify primary inputs/outputs and generated RTL.

A. RTL Scanner Design

We propose RTL scanners based on an exploration of the Abstract Syntax Tree (AST) of the Verilog file. At first, a

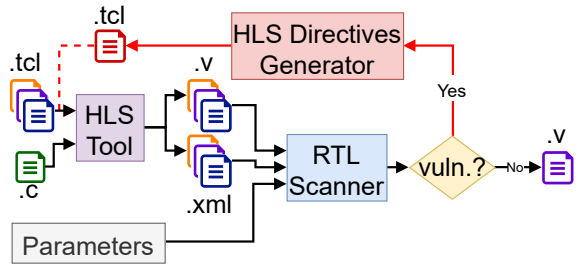

Fig. 2. Our proposed detection and correction flow, where source files in a high-level language like C are passed to an HLS tool with accompanying HLS directives (e.g., in a `.tcl` file). We feed the generated hardware design (RTL) through a scanner to identify weaknesses; if present, we generate a new directive file in an attempt to remediate the weakness.

Verilog parser extracts the AST, and an HLS report parser extracts information such as the top module and primary inputs/outputs from the structured report file generated by an HLS tool (e.g., in XML format). This information is passed to a Visitor object which explores the AST.

A Visitor is a common software design pattern for creating objects to explore graph data structures. It explores the tree starting from a given node (in this case the root of the tree) and performs user-defined functions only on specific kinds of nodes defined by the user. The RTL Scanner includes one or more Visitors for each weakness; new weakness scanners can be added by implementing new visitors. When exploring the AST, the Visitor looks for patterns that are related to the weakness in scope. Since the analyzed RTL is automatically generated, the scanners can use the patterns and structures which are characteristic of the specific HLS tool used, obviating the need for generality as desired in prior work [14].

As discussed in Section II, the passthrough weakness enables other weaknesses as it leaks data to primary outputs before the computation is complete. For this reason, we implemented a scanner for the passthrough weakness.

B. The Passthrough Scanner

Our scanner needs information such as the names of the top module, primary input/outputs. By inspecting synthetic benchmarks with and without the passthrough weakness after using a commercial HLS tool, we observed several patterns. In designs without passthrough typically:

- the primary output is assigned with non-blocking substitution;
- the primary output is assigned in continuous assignment to a signal with post-fix "`_reg`";
- the primary output is assigned to a Phi variable ϕ, which has one of the two properties above.

Taking into account these observations, one can design a visitor for the passthrough weakness, for example, like one illustrated in Fig. 3 (one can make adjustments for other HLS tools). Our visitor defines a function for analyzing continuous assignments. Before calling the function, the relevant signal list is initialized with the primary outputs. The function then checks if the left-hand side is in the list and if so, it checks if

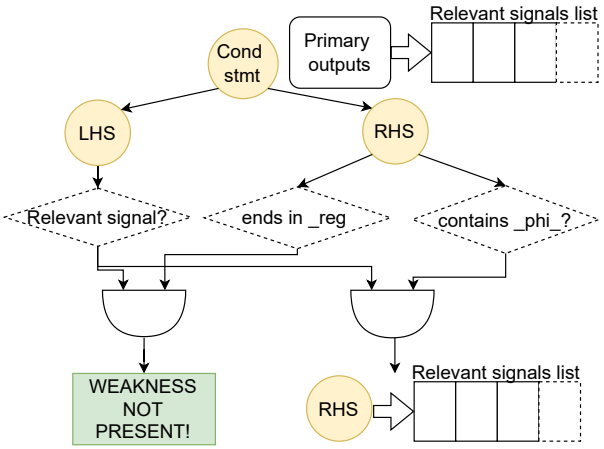

Fig. 3. Scheme of the passthrough visitor.

the right-hand side has a post-fix "`_reg`", or includes "`_phi`" in its name. If this condition is met then a flag to signal the weakness is not present is raised and the function exits. If the right-hand side includes "`_phi`", then its name is added to the relevant signal list `phi_list` if not already present. At the end of the traversal, if the list of relevant signals has increased, the AST is scanned again. If the scanning ends without raising the "not present" flag, we can infer that the weakness is present.

C. Correction via HLS Directives

Two conditions must be met to mitigate the passthrough weakness: (i) a registered output, (ii) appropriate control logic. The registered output is necessary to separate the intermediate output net to the top-level output net. The control logic enables the added register only when the operation is complete. We intuited that weaknesses can be remediated using the directives of an HLS tool, the idea being that after we detect a weakness, we can add the required directive(s) to the corresponding file of the HLS project. After re-running the synthesis, we can scan the generated design again to validate that the weakness has been fixed. If it is present, an error message is raised to get the attention of the designer. We investigated the documentation of a typical commercial tool, as an example, identifying three candidate solutions:

1) `set_directive_interface [OPTIONS] <location> <port>` specifies how a function interface is synthesized. This directive provides port-level granularity (i.e. the ability to specify a specific port of a specific function) using `<location>` and `<port>`. Using this directive as a solution requires enabling the `-register` option and adding control logic using `-mode ap_hs|ap_ack|ap_ovld|ap_vld`.
2) `config_interface [OPTIONS]` is a configuration command applied at the solution-level. This controls the default IO interface synthesized by the HLS tool for each function. We use this command with the `-register_io scalar_out|scalar_all` option.

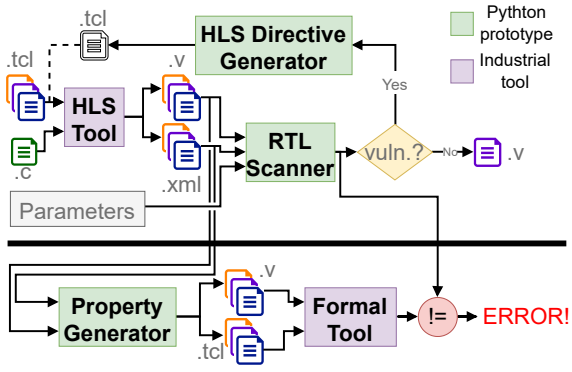

Fig. 4. Prototype and evaluation framework implementation.

```
void compute(unsigned short int n,
             unsigned short int m, bool mode,
             unsigned int *ret){
    if (mode) fib(n, ret);
    else fact(n, ret);
}
```

Listing 1. Combining two functions can yield different outcomes depending on whether the functions are inlined or not.

```
assert property(
    @(posedge clk) disable iff (!rst_n)
↪   $changed(in) ##[1:$] $changed(out) |->
↪   ($rose(done) || $fell(done));
);
```

Listing 2. No passthrough property.

This specifies that all scalar outputs must be registered. There is no option to specify the mode of the registers.

3) `config_rtl [OPTIONS]` is a configuration command. Using it with `-register_all_io` option is similar to the configuration above but does not provide the ability to only modify outputs.

Similar directives exist for other tools. In this specific instance, we propose using `config_interface -register_io scalar_out` directive. `set_directive_interface` provides the most control but we found experimentally that it was not always effective. It also requires manually specified security-critical outputs and introduces the possibility of error. `config_interface` and `config_rtl` are functionally similar but the former provides the control to only register the outputs. It allows for a context-free approach, not requiring any signal names. For secure IP like cryptographic accelerators, no output should be updated intermediately.

IV. Experimental Evaluation

A. Prototype Implementation

We implemented a prototype framework for a commercial HLS tool and integrated a formal verification tool for validating our work, as illustrated in Fig. 4. In this way, we can verify the correctness of our scanners when launching large batches of tests on several benchmarks. Formal verification can be enabled or disabled. We implemented the prototype in Python, a commercial HLS tool[1] and Cadence Jasper Gold as the formal verification tool. We used Pyverilog to parse the design and extract the AST [15] and the Python ElementTree XML API to parse the XML reports. Alternative commercial tools can be used and new scanners can be added by writing new visitors. Our implementation will be open-sourced.

B. Formal Properties Generation

Writing and checking formal properties is generally non-trivial, and our work aims to support early correction and detection to reduce the need for cumbersome formal verification. However, for experimentally validating our RTL scanner, we designed a formal verification flow that also identifies the presence of the passthrough weakness.

[1]We cannot disclose the tool due to license agreements.

1) Property Formulation: We use the default top-level control signals added to the design to ensure that the outputs do not change unless the `done` signal is asserted. This behavior can be described using a SystemVerilog Assertion (SVA) with the property in Listing 2. The property states that a change in `in`, followed by a change in `out` after an indeterminate number of cycles implicates that `done` must be asserted or de-asserted in the same clock cycle. The `done` signal is asserted when the end of the block-level function is reached and de-asserted a clock cycle later. We allow the signal to change when done is de-asserted in cases where the output is cleared after the operation. This specifies our desired behavior that the output should only change when the operation is complete.

Constraints are required to restrict the inputs to valid values. We constrain the `start` signal to `1'b1` to force the design into operation, as these are the states of interest. It has no effect on the accuracy of the property (i.e., false negatives) as the output will never change when `start=1'b0`. We do not constrain the values of the user-defined inputs as there should never be intermediate leakage.

2) Automated Generation Framework: We obtain the name of the top-level module and its IO from the synthesis report generated by the HLS tool. For each unique input-output pair, a new instance of the property template is created; for N inputs and M outputs, the set of properties will be of size $N \times M$. This ensures that a change in any input signal will not result in intermediate leakage in any of the outputs. The set of properties, along with verification statements such as the constraint, are added to a verification module template and listed inside a .sva file. The list of top-level IO is used to populate the verification module's IO. We instantiate the verification module inside the HLS-generated top-level using a SystemVerilog `bind` statement in a .sv file. A .tcl file is created using design and verification file names. This .tcl script is used to run JasperGold.

C. Benchmarks

For evaluation, we selected three synthetic benchmarks to explore the HLS-induced passthrough weakness. We use an implementation of the Fibonacci sequence and a factorial function to represent designs with loops updating the internal

TABLE I

EXPERIMENTAL RESULTS SHOWING BENCHMARK CHARACTERISTICS AFTER HLS AND WEAKNESS PRESENCE AS DETECTED BY OUR SCANNER. DESIGN COMPLEXITY IS REPRESENTED BY THE LINES OF CODE (LOC). TCL SUMMARIZES THE DIRECTIVE SET USED FOR SYNTHESIS.

DESIGN	LOC (.c)	TCL	LOC (.v)	Weakness Present	Design Characteristics			Time (s)		
					Latency	FF	LUT	Scanner Parse	Scanner Scan	Formal
Factorial	9	DEFAULT	597	N	**	104	172	1.225	0.0015	3.934
		REG	497	N	**	171	177	1.222	0.0020	3.757
Fibonacci	11	DEFAULT	314	Y	**	98	159	1.206	0.0009	3.678
		REG	249	N	**	181	144	1.219	0.0012	3.803
Combined	23	INLINE OFF	1067	Y	**	255	352	1.241	0.0015	3.866
		INLINE OFF + REG	928	N	**	388	353	1.211	0.0012	4.062
		REG	786	N	**	338	313	1.256	0.0016	3.831
		EMPTY	922	N	**	206	307	1.230	0.0021	3.977
PRESENT [7]	205	ALL	5405	Y	73	539	2105	1.323	0.0052	4.140
		REG	5535	N	110	808	2135	1.362	0.0057	80.66
		PIPELINE	4488	Y	205	681	1697	1.357	0.0051	4.274
		UNROLL	3463	Y	130	766	1598	1.317	0.0044	4.334
		EMPTY	1717	Y	5956	659	2340	1.316	0.0031	14.174
serpent [7]	331	ALL	4129	Y	36	667	1543	1.243	0.0024	5.399
		REG	5303	N	337	4433	1766	1.276	0.0034	300.83
		PIPELINE	4129	Y	36	667	1543	1.274	0.0026	5.871
		UNROLL*	-	-	-	-	-	-	-	–
		EMPTY	2574	Y	21944	4434	5964	1.248	0.0025	12.414
AES-v1 [7]	372	ALL	6751	Y	532	1266	5860	1.287	0.0062	4.344
		REG	7038	N	563	1983	5953	1.298	0.0062	204.19
		PIPELINE	6325	Y	584	1629	11346	1.312	0.0062	4.552
		UNROLL	4516	Y	1048	1453	6343	1.307	0.0055	4.427
		EMPTY	10535	Y	10542	2015	15082	1.304	0.0053	4.571
AES-v2 [16]	303	ALL	3660	Y	166	1979	1824	1.215	0.0016	6.669
		REG	3387	N	170	2368	1800	1.222	0.0019	189.241
		PIPELINE	3660	Y	166	1824	1824	1.205	0.0016	6.054
		UNROLL	3190	Y	67	1290	1594	1.249	0.0032	7.334
		EMPTY	2989	Y	1103	2247	27388	1.228	0.0024	6.234

*Output does not update **Latency depends on inputs

state to compute their results. Fibonacci implementation has the passthrough weakness when implemented with no HLS directives. The Factorial implementation has a similar structure but does not exhibit the weakness. Combining the two functions in a module that allows one to select one of them via an input as in Listing 1, does not present this weakness when the functions are inlined. When not inlined, Fibonacci sequence leaks to the output when selected. These benchmarks serve to illustrate that similar c structures can yield different results, motivating scanning for issues on the generated RTL.

Additionally, we selected block ciphers, AES(-v1), PRESENT and Serpent from [7] as benchmarks. We also selected the open-source AES(-v2) from the Vitis Library [16]. The cipher implementations came with HLS directives for each design as they were HLS-ready. We derived 5 sets of directives:

1) All directives (ALL) from the original set;
2) Register I/O (REG): directives in the original set plus the `config_interface -register_io scalar_out` directive;
3) Only pipelining (PIPE): Original set of directives removing all unrolling directives;
4) Only unrolling (UNROLL): Original set of directives removing pipeline directives;
5) No directives (EMPTY): no directives specified; tool's defaults as in [4].

This allows us to study choices that a designer could make.

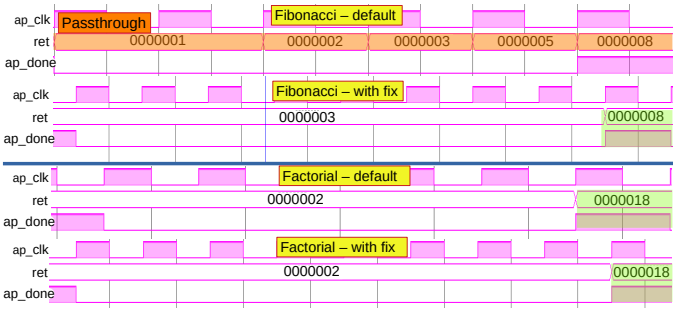

Fig. 5. Waveform of Fibonacci sequence (top) and factorial (bottom) with both default settings and register I/O directive. Notice how Factorial does not exhibit the weakness but adding the directives adds latency to the computation.

D. Experimental Setup and Results

To explore the feasibility of our proposed framework, we take a set of benchmark designs, D, and a set of directives P and check if the combination $< D, p >, p \in P$ contains the passthrough weakness after HLS. If it does, we want to check that $< D, p + f >$, with f being the fixing directive, does not contain the passthrough weakness. We feed the designs through our framework running on 4 cores on an Intel Xeon Gold 6248R with 16GB of RAM. We characterize the benchmarks in terms of their complexity (lines of C code and lines of generated Verilog), their resource requirements in terms of latency, flip-flops, and LUTs, and run times, with results in Table I. For brevity, we report the results of the fix applied only to the original set of directives (REG rows). The results show that the weakness was correctly identified in all

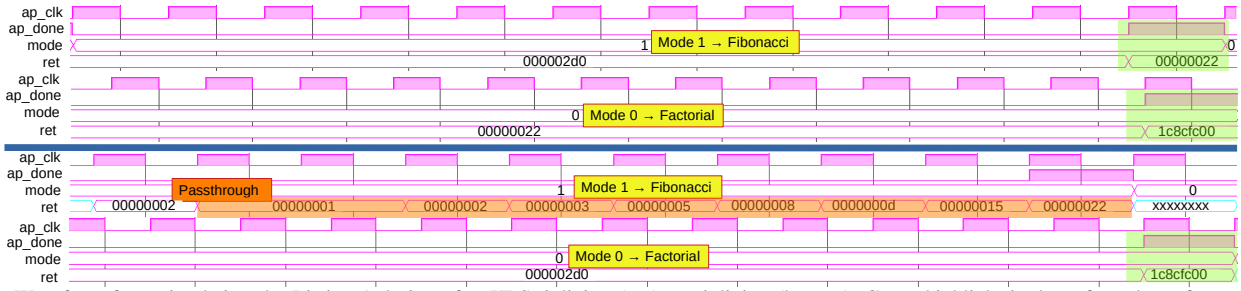

Fig. 6. Waveform from simulating the Listing 1 design after HLS, inlining (top), no inlining (bottom). Green highlight is the safe update of output at the end of computation.

Fig. 7. Waveform of AES from Vitis Libraries [16], with (top) and without (bottom) fix. Highlighted in orange is the passthrough which leaks the first used subkey at round 1.

cases by the scanner as verified with the formal tool.

Synthetic Benchmarks. As shown in the waveforms in Fig. 5, when the default settings are used, both the Fibonacci sequence and the factorial exhibit the passthrough weakness. The combined module, as shown in Listing 1, has interesting behavior. Without inlining, only the Fibonacci sequence leaks data on the output during the computation. When both functions are inlined the passthrough weakness is absent. This is illustrated in Fig. 6. For complex scenarios, it is hard to predict the security outcomes by looking only at HLL code.

Serpent. This block cipher exhibited passthrough when the directive to register all outputs was not used. However, using such a directive increased latency from 36→337 cycles.

PRESENT. This block cipher also exhibited passthrough in all cases without the directive to register the output registers. The directive increased latency from 73→110 cycles.

AES-v1. This also exhibited passthrough in all cases that did not include the directive to register the output registers. The directive increased latency from 532→563 cycles.

AES-v2. This also exhibited passthrough in all cases where we did not include the directive to register the outputs. In Fig. 7 we show the weakness, which leaks the subkey used at round 1. In this case, the directive had a minimal latency increase: 166→170.

Overall, our scanner identified all instances of the passthrough weaknesses. The scanner's bottleneck is the Verilog parser. We used Pyverilog [15] since it is open-source, while faster commercial Verilog parsers could reduce this bottleneck. However, this is not the focus of this work. Furthermore, our fixing directive increases latency from 1.1× to 9× in the worst case, emphasizing the need to scan for the weakness rather than applying it universally.

V. Conclusion

We proposed a framework to detect and correct a class of security weaknesses that can be induced by high-level synthesis. Our experimental results on synthetic algorithms and realistic block cipher benchmarks show that our prototype is as effective at detecting the weakness in generated Verilog as more cumbersome formal verification, while being up to 200× faster even without a fast Verilog parser. This shows the potential for static analysis for weakness detection on HLS generated designs that can take advantage of template-based code structures, motivating research for new static scanners.

References

[1] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable hardware bugs," in *28th USENIX Security Symp. (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230.

[2] Siemens, "Working smarter, not harder: Nvidia closes design complexity gap with hls."

[3] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2628–2639, 2020.

[4] N. Pundir *et al.*, "Analyzing Security Vulnerabilities Induced by High-level Synthesis," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, no. 3, pp. 1–22, Jul. 2022.

[5] M. Bartock *et al.*, "Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST IR 8320, 2022.

[6] J. Sepulveda *et al.*, "Towards the formal verification of security properties of a network-on-chip router," in *2018 IEEE 23rd Eur. Test Symp. (ETS)*, 2018, pp. 1–6.

[7] P. Socha, V. Miškovský, and M. Novotný, "High-level synthesis, cryptography, and side-channel countermeasures: A comprehensive evaluation," *Microprocessors and Microsystems*, vol. 85, p. 104311, 2021.

[8] R. Nane *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, 2016.

[9] P. Brisk *et al.*, "Optimal register sharing for high-level synthesis of ssa form programs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 5, pp. 772–779, 2006.

[10] C. Pilato *et al.*, "Black-Hat High-Level Synthesis: Myth or Reality?" *IEEE Trans. VLSI Syst.*, vol. 27, no. 4, pp. 913–926, Apr. 2019.

[11] K. Basu *et al.*, "CAD-Base: An Attack Vector into the Electronics Supply Chain," *ACM Transactions on Design Automation of Electronic Systems*, vol. 24, no. 4, pp. 38:1–38:30, Apr. 2019.

[12] M. Abderehman *et al.*, "BLAST: Belling the Black-Hat High-Level Synthesis Tool," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3661–3672, Nov. 2022.

[13] The MITRE Corporation, "CWE - CWE-1194: Hardware Design (4.1)," https://cwe.mitre.org/data/definitions/1194.html, 2022.

[14] B. Ahmad *et al.*, "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design," Sep 2022, http://arxiv.org/abs/2209.01291.

[15] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460.

[16] Xilinx, "Vitis accelerated libraries," https://github.com/Xilinx/Vitis_Libraries/tree/master.

Survey

Joey Ah-kiow
Department of Electrical and Software Engineering
University of Calgary
joey.ahkiow@ucalgary.ca

***Abstract*—This survey paper was prepared for assignment 1 of ENEL 592. Recent hardware-based attacks have resulted in newfound motivation to improve hardware security verification. In this survey, we investigate the meaning of hardware security bugs, the Security Development Lifecycle (SDL) implemented to address these bugs, and state-of-the-art techniques proposed to assist in security bug detection in the SDL.**

I. Introduction

Recent attacks [1]–[4] have shone a spotlight on hardware vulnerabilities. These attacks invalidated the assumption that hardware exploitation required physical access and exposed the immature state of hardware security verification, which has proven insufficient to thoroughly verify modern System-on-Chips (SoCs) [5]. This has motivated industry and academia to explore new methodologies to assist in the creation of secure hardware. Bugs are an abstract concept, pervasive throughout the development lifecycle, and must be addressed as such. The vagueness of the term *bug* further compounds this challenge as no definition can be drawn upon to guide the direction of these techniques. Although we can reason about a software bug from the extensive studies of software errors [6], [7], the same cannot be said for hardware.

In this paper, we attempt to address this issue by providing a definition for *hardware security bugs*, a necessary step in our efforts to address them effectively. We then inspect the hardware Security Development Lifecycle (SDL) [8], an initiative to introduce a security-centric approach to product development, to understand its operation and the challenges present within. Afterward, we continue our exploration of hardware security bugs by reviewing the new works in static code analysis, hardware fuzzing, and property generation that have shown promise in detecting hardware security bugs. We observe that the distributed point solutions have made it difficult to evaluate the current state-of-the-art and attempt to address this challenge. Our motivation is to identify the gaps present in each technique and provide an overview of the current hardware security verification landscape. Prior work has surveyed the techniques for the security verification of computers [9], but in this paper, we focus on the security verification of hardware.

II. Hardware Security Bugs

One of the earliest documented uses of the term "bug" in computer science was when it was famously used by Mary Grace Hopper, to document a moth – a literal bug – being found inside the Mark II computer. Hopper also expanded its definition when she defined "debug" as "to remove a malfunction from a computer or an error from a routine" [10] which suggested, for the first time, that a "bug" applied to both hardware and software. Since then, the term has been extensively studied in the software domain [6], [7]. However, to our knowledge, this has not been the case for hardware. In this section, we attempt to extrapolate from the software domain into the hardware domain to establish a definition of a *hardware bug*. Furthermore, we formalize how a software and hardware bug differ from one another, and the distinction between a hardware functional and security bug, to serve as a starting point and motivate more work in this field.

A. Definition

[6] refers to a *bug* as an "amalgam of one or more software errors, runtime faults, and runtime failures". A *software error* is a mistake made in the implementation of software. A *runtime fault* is a machine state (e.g. improper branching, incorrect register value, etc.) that is a result of one or more software errors. A *runtime failure* is the observable divergence of the software's behavior from the intended requirements, as a result of one or more software faults. Although *debugging* – the act of removing bugs – generally involves starting from runtime failure(s) and working backward to find and correct the software error(s), it does not mean that software errors are the cause of bugs. Software errors are simply the "physical" manifestation of a possibly bigger issue, such as the miscomprehension of requirements and/or specifications.

[7] states that "software development involves the translation of information from one representation to another", a largely manual and error-prone process. We can thus generalize the notion of bugs such that they are defined as the errors that occur during this translation process. Since hardware development undertakes a similar translation, we can apply this knowledge to define a hardware bug as **an error in the translation of the design requirements, resulting in an output that does not properly convey the intent of its input**. This can be applied to any stage of translation, e.g., bugs can be introduced during the translation from requirements to specifications early in the development lifecycle, just as well as during the fabrication from layout to silicon. The differences are in the forms of the input and output, the collateral available, the human effort required, and the potential impact that an error can cause if it is caught (detection and correction) or otherwise (reputational damage, information leakage, etc.).

The next step is to draw the distinction between functional and security bugs. In general, a security bug is one that

violates the desired confidentiality, integrity, or availability of a product and the data it operates on. Confidentiality, integrity, and availability are widely considered tenets of information security. They respectively refer to the notion that only those who have permission to assets can access and understand them, that the assets are trustworthy and accurate, and that assets are available when required. We apply this to hardware in a hierarchical manner. First, we consider the function of the hardware. Let us use an AES accelerator as an example. We must ensure the confidentiality and integrity of the data being encrypted, commonly called *plaintext*. We must also ensure that the encrypted data, commonly called *ciphertext*, is available when required. So, we can argue that any bug which has an impact on these outcomes can be deemed as a security bug. These bugs may range from implementing the AES specification properly (e.g., incorrect number of rounds) to ensuring that the intermediate result cannot be read during operation.

III. Security Development Lifecycle

The Security Development Lifecycle (SDL) runs concurrently with the development lifecycle to assist in the creation of products that are secure and trustworthy by design. It can take various forms, but at its root, it consists of a holistic set of processes to ensure that the security of products is scrutinized and tested at every stage of development. The SDL is an established practice for software security [11], and recent attacks [1]–[4] have motivated industry to translate the practice to hardware [8]. In this paper, we utilize the Intel SDL as an industry representative of the hardware SDL. It consists of 6 stages: (1) Planning and Assessment, (2) Architecture, (3) Design, (4) Implementation, (5) Security Validation, and (6) Release and Post Deployment.

A. Stage Summary

1) Planning and Assessment: In this stage, the project is assessed, security requirements are created, and tasks are planned throughout the SDL to verify that these requirements are fulfilled. The security requirements transcribe, in natural language, the behavior of the product such that it operates securely. This represents the first challenge. Assuming that the requirements are appropriate, they must be translated and refined throughout the development lifecycle, from requirements to specifications to properties. However, natural language, where implicit meaning and ambiguity are pervasive, is ill-suited for such detailed translations. These translations may thus introduce undocumented assumptions or misunderstandings, deviating the design from the original intent and introducing bugs.

2) Architecture: In this stage, the security requirements are translated into design specifications using the appropriate adversary models and threat modeling techniques. The assets are identified, the attack surface is assessed, and past vulnerabilities and attacks are considered to engineer the architecture such that it optimally satisfies the requirements. Here the biggest challenges are the appropriate translation and understanding of the requirements, and the human expertise required.

3) Design: In the design stage, the design specifications are further refined into lower levels of abstraction suitable for implementation, such as RTL. The documentation is then used to create a validation strategy to ensure that all requirements are covered in the validation phase.

4) Implementation: In this stage, the design of the product, with its included security features is implemented. This is the stage most associated with the introduction of bugs, but as we discussed above, the root cause of the issue may be further upstream. The security measures in this stage include manual code reviews and static analysis. We note that dynamic analysis with simulation and emulation is not completed comprehensively at this stage, as the design is not fully complete. The main challenges in this stage consist of the large manual effort required for code reviews and lackluster available collateral, such as testing environments. Manual code reviews are a challenging task because they require extended bouts of attention, extensive functional and security knowledge, navigation between different parts of the design and the associated documentation, and assumptions of inter-component interactions.

5) Security Validation: The majority of the validation and verification is done in this stage after the implementation of the design is complete. Here, techniques like simulation and emulation are used to verify that the design satisfies the desired properties, usually represented in the form of assertions. These assertions encode the 'proper' behavior of the design and trigger if they are not satisfied, notifying verification engineers of the presence of a bug. These assertions are also additional documentation embedded within the design, and the most precise representation of the requirements created in the Planning and Assessment stage. There are a tremendous amount of challenges in this stage. First, the assertions are prone to the translation issues mentioned in Section III-A1. This may result in a set of properties that fail to accurately capture the desired behavior of the design. Furthermore, once potential bugs are found, how can their severity be assessed and a proper remediation plan put into action?

6) Release and Post Deployment: The last stage is release and post-deployment. A final check is completed to verify that all previous issues were resolved before the product is released. Once released the product – hardware – is largely unpatchable. This makes any uncaught weakness have potentially disastrous implications as demonstrated by Meltdown [1] and Spectre [2]. If a vulnerability is found by a responsible party post-deployment, the firmware running above the hardware may allow for ad hoc fixes; however, the root cause still lays dormant somewhere in the silicon, waiting to be exploited.

B. Discussion

Each stage requires extensive manual effort, functional and security expertise, the collaboration between many parties, and clear documentation, which may not always be available. This can result in ad hoc or ineffective solutions, exposing systems to unnecessary risk. While the hardware SDL integrates a security-centric mindset into the development lifecycle, it is ultimately held back by the challenges faced at each step.

Researchers have taken various routes to attempt these challenges, many of which have shown promise, but this has made it difficult to understand the current landscape of the start-of-the-art.

IV. Bug Detection

TABLE I
Prior Works

Technique	Prior Work
Static Code Analysis	CWEAT [12]
Fuzzing	RFUZZ [13], DiFuzzRTL [14], ProcessorFuzz [15], Fuzzing Hardware like Software [16], TheHuzz [17]
Property Generation	SCIFinder [18], Undine [19], Isadora [20]

The detection of hardware security bugs is a daunting task. The search for bugs employs a concert of techniques such as manual analysis, simulation, and formal methods, each with its own tradeoffs. However, recent work has demonstrated that current methods are lackluster [5] and motivated hardware security researchers to investigate new approaches. In this section, we discuss three emerging techniques for hardware security bug detection, Static Code Analysis, Hardware Fuzzing, and Property Generation. For each technique, We provide a brief background, their motivations, and challenges, their suitability to the design cycle, and explore a sample of work that apply these techniques.

A. Static Code Analysis

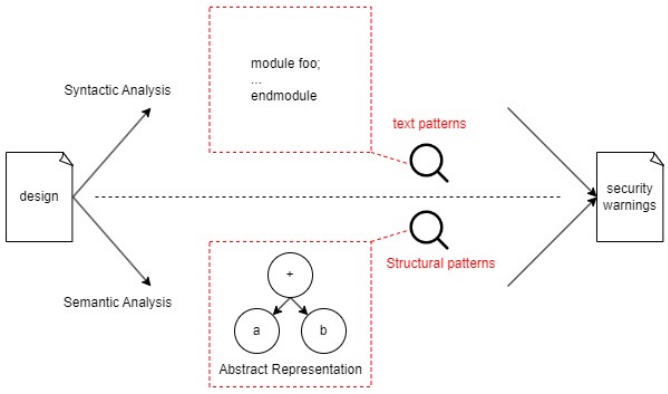

Fig. 1. Static Code Analysis Flowchart. Syntactic analysis processes the source code before elaboration, parsing, etc., looking for code patterns that may introduce security weaknesses. Semantic analysis first translates the code into a more abstract representation (e.g., Abstract Syntax Tree, Control Flow Graph) to detect more complex bugs [21].

Static code analysis acts directly on source code to detect potential weaknesses using two possible approaches – syntactic and semantic analysis. Figure 1 illustrates the two approaches. Syntactic analysis processes the source code before elaboration, parsing, etc., looking for code patterns that may introduce security weaknesses. Syntactic analysis is best suited for simple checks, such as searching for `default` statements in FSM case statements in (System)Verilog designs. On the other hand, semantic analysis first translates the code into a more abstract representation (e.g., Abstract Syntax Tree, Data Flow Graph, Control Flow Graph) to detect more complex bugs. Syntactic analysis allows for easier on-the-fly scanning since it doesn't require complete source code to construct a more abstract representation. On the other hand, a more abstract representation allows for more powerful and efficient processing. Regardless, both approaches rely on a well-documented database of potential concerns to search for which may not always be available. This database of ground truth provides the 'rules' that guide the analysis and may provide other supplementary information such as severity.

Static code analysis is a potent technique because it is easy to adopt, thanks to the minimal collateral and setup effort required. It does not require a complete design, any testing environment or instrumentation, and can be easily integrated at various points of the design process. This could lead to the detection of bugs, security or otherwise, earlier in the development process, where it is much cheaper and easier to correct. This is a 'shift-left' compared to traditional methods, a crucial effort to provide designers with feedback during design time – the best time to make design changes and address potential issues. Existing functional and security tools are widely available for software scanning, with both open-source [22], [23] and industry [24] options. Hardware static code analysis tools, working at the RTL, also exist but have focused solely on functional verification [25].

There are challenges that must be addressed to make security static code analysis viable. One challenge is to prune the search space for security-criticality. The distinction between functional and security bugs is hazy, and static analysis' pattern-matching nature makes it challenging to accurately identify security context. A solution proposed in [12] is to utilize keywords that are commonly security-relevant. For example, a naive approach would be to deem any signal/register with an identifier that contains "aes" as security-relevant and apply this keyword match to all scanners. However, not all potential concerns are related to cryptography and this could quickly lead to false negatives (weaknesses that are not flagged) and false positives (non-weaknesses that are flagged). Thus, it is imperative that the combination of keywords and scanners is judiciously chosen. It is also important to optimize the keywords used such that they offer the best trade-off between accuracy (the ratio between true positives and false negatives) and noise (the ratio between true positives and false positives). This could be done through interactions with the designer but relies on their understanding of what is "security-relevant".

Another challenge is that the performance of static analysis is driven by the underlying database of vulnerable patterns. Code analyzers are only as good as the patterns they are searching for. An approach in both hardware [12] and software [22], [23] is to utilize MITRE's CWEs [26]. The CWEs provide common security weaknesses and organize them in a hierarchy. They present the weaknesses in a generic manner, making them applicable across multiple designs/instances within the same design. However, we identify two concerns that we believe must be addressed to properly utilize them for static analysis. The first concern is the translation from a CWE to a rule. The proper interpretation of a CWE is

critical to be able to derive a rule (or collection of) that can effectively capture the root cause of the concern. The second, related, concern is obtaining the necessary contextual information. There is a baseline level of knowledge about the design which must be known for each found potential weakness to assess its legitimacy. These are crucial to not only the detection efficacy but also the assessment afterward, as pinpointing the appropriate CWE is important to determine corrective measures.

Let us consider CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks and CWE-1231: Improper Prevention of Lock Bit Modification. They are adjacent, belonging to the same parent, and refer to seemingly similar issues – the overriding of lock bits. However, CWE-1234 concerns itself specifically to debug integration, typically done at other stages of design. The ability to differentiate between the two is a result of having the ability to reason about the context of a given scenario and understanding the CWEs themselves. How can we define a set of rules to formalize the intent behind the CWEs? What contextual information is required and how can we obtain it? These questions, and many more, are still largely unresolved. Even in the more advanced software domain, the performance of security-critical static code analysis remains questionable [21], [27] with little-to-no improvement in recent years [27].

1) CWEAT: To our knowledge, Ahmad et al. [12] were the first to propose security-dedicated static code analysis for hardware. They propose the use of MITRE's hardware CWEs [26] as a database of potential concerns and further categorize them based on amenability to static analysis. They find that 11 CWEs are detectable from context-free static analysis and 22 are detectable with additional context – this represents 38% of the RTL-relevant CWEs. Non-RTL-relevant CWEs include issues with documentation and fabrication. They take a semantic approach by first parsing the RTL source code into Abstract Syntax Trees and leverage keywords to guide towards security relevancy. They implement scanners for 5 of the context-free CWEs:

1) **CWE-1234: Hardware Internal or Debug Modes Allow Override of Locks.** The scanner for this CWE checks the conditionals of if-else statements and searches for a combination of keywords and logical operations that indicate overriding (i.e., OR).
2) **CWE-1271: Uninitialized Value on Reset for Registers with Security Settings.** This scanner operates by first iterating through every register declaration (e.g., logic, reg) and storing the registers it believes are security-relevant based on their identifier. Afterward, it searches for and traverses reset blocks to check whether these registers are reset.
3) **CWE-1245: Improper Finite State Machines (FSMs) in Hardware Logic.** This scanner first obtains the state variable of an FSM. Next, it searches for the identifier of the state variable obtained to search for assignments to the state and stores information like previous state, new state, conditions for assignment, and location. Finally, it uses this information to scan for FSM weaknesses such as FSM deadlocks (a state that can be entered but not exited) and unreachable states.
4) **CWE-1280: Access Control Check Implemented After Asset is Accessed.** The CWE-1280 scanner begins traversal at sequential blocks ("always" blocks with blocking assignments). It creates a list of signals that are read and written and stores their identifiers, parent block identifier and line number. If a signal is read and written in the same sequential block, the line numbers are checked to ensure that the read happens after the write.
5) **CWE-1262: Improper Access Control for Register Interface.** This scanner searches for every register with identifiers containing "wdata" as the authors note this suggests a write-data register that is not write-protected, presumably a heuristically determined characteristic. For each of these registers, they store the conditions checked before the register is written. They remove 'non-controlling' conditions such as `@posedge clock` and if the final list is empty then the register is deemed as unprotected.

Of the warnings generated by their scanners, 35 (24.31%) were true positives (issues that have security implications), 19 (13.1%) were indeterminate, and 90 (62.5%) were false positives. While this may seem like an unpromising result at first, the authors discuss that these results are simply meant to show that such an approach can successfully identify weaknesses in hardware designs. Even factoring in the false positives, this is a tremendous reduction in search space when looking for possible bugs, from thousands of LoC to 144 locations.

The authors discuss the limitations of their approach. They point out the trade-off between accuracy (true positives/total) and comprehensiveness. On one hand, casting a wider net should output more warnings, resulting in more 'noise' but also more true positives. On the other hand, increasing specificity should be more accurate (less false positives) but might miss weaknesses that would have been flagged otherwise. They show that the scanners can be improved by modifying the list of keywords based on the project. In their example, they manage to raise the percentage of positives/total hits from 3.1% to 33.3% for the CWE-1271 scanner on the Hack@DAC 2021 SoC. While this shows promise in increasing scanning performance, it is a manual process guided by heuristics, which can quickly lead to errors and missed weaknesses.

2) Summary: CWEAT demonstrates that static code analysis is able to find potential security concerns in hardware designs. CWEAT uses the hardware CWEs at its database of vulnerable patterns, the only security weaknesses list for hardware to our knowledge. They take a semantic approach and parse RTL designs in ASTs on which to operate. They address the challenge of pruning for security-criticality by applying a list of keywords, which can be further improved by the designer. They prototype five scanners that are able to find 35 security concerns. We cannot do a comparison across works due to the lack of other literature applying static code analysis in hardware security.

B. Hardware Fuzzing

1) Background: Originally motivated by its success in software testing, hardware fuzzing has shown to be a promising alternative to constrained random verification (CRV) [28] in the hardware domain. It also scales better with increasing design complexity compared to formal methods. Hardware fuzzing consists of instrumenting a Device Under Test (DUT), stimulating it with some automatically generated, semi-random input, and evaluating its output to detect behavior that diverts from its specifications. Fuzzing's primary advantage is that it does not piece together a sequence of previous inputs like CRV, rather it mutates previously known 'interesting' inputs and utilizes those as new inputs. The intuition behind this is that minor modifications to inputs that increased coverage will result in other inputs that will also increase coverage. There are three classes of fuzzers, *white-box, grey-box, and black-box*. These terms signify the amount of internal information that the fuzzer has when evaluating the effectiveness of an input – White-box refers to full access, grey-box refers to partial access, and black-box refers to no access. The internal information is obtained from either an instrumented design which can feed the information directly to the fuzzer or from execution traces gathered during simulation. Fuzzers are typically run until they "converge", that is, until running new test inputs results in no noticeable increase in coverage. At this point, the cost of continuing to run the fuzzer is not worth the extra coverage. The distinctive characteristics of fuzzers are (1) the inputs they utilize and their mutations, (2) how they define and obtain coverage, and (3) how they evaluate the behavior of the DUT to detect an error.

The main advantages of fuzzing are that it requires less effort to set up and use compared to other traditional methods. After the construction of the fuzzer, it should be a fairly self-contained testing unit with minimal human intervention required. There is no manual constraint setup required, which reduces error when compared to CRV and allows the fuzzer to test some inputs that do not conform to specification. This is useful to model attacks such as fault injection where such input can occur. Fuzzing's ability to automatically learn from what is 'new', a feature absent in CRV, is what truly sets it apart as it can explore new states more effectively. Since a fuzzer's only motivation is to obtain more coverage, it will focus on getting as much coverage as possible instead of attempting to exhaustively explore every state like formal methods. In some sense, fuzzing is a greedy approach to formal verification, as it 'ignores' the states that are hard to reach. Depending on the implementation, a fuzzer can also be highly portable. For example, [16] implements a generic fuzzing harness by leveraging the standardized TL-UL bus protocol to fuzz multiple IPs that implement the protocol. Finally, a fuzzer can be guided to specific features/parts of the design by careful selection of the test input. This is useful for compositional approaches, where an empty seed can be used first to obtain as much "easy" coverage as possible, followed by targeted seeds that focus on aspects of the design that were not tested previously due to convergence.

There are three main challenges to hardware fuzzing: (i) defining and measuring coverage, (ii) generating optimal inputs, and (iii) detecting when a potential bug has been triggered.

a) Coverage: A meaningful definition of coverage has yet to be found for hardware fuzzing. Previous work has attempted to address this issue by utilizing *mux coverage* [13], *register coverage* [14], *Control and Status Registers (CSR) coverage* [15], and other coverage metrics [16], [17] that are translatable from software to hardware. However, the effectiveness of these coverage metrics has not been investigated and it is currently unknown how useful they are to guide fuzzers to "interesting" states. The challenge in hardware fuzzing coverage is finding a coverage metric that guides fuzzers toward security-critical states, is understandable to traditional design verification engineers, and is practically possible to obtain without too much instrumentation/runtime overhead.

b) Test Inputs: The optimal input generation of hardware fuzzers that will result in the best speed to coverage is also an open question. Most hardware fuzzers to date have focused on fuzzing processors, where operating at the ISA level is possible and logical. Other works attempting to fuzz generic IP have done so at the RTL level, instrumenting the top level of the IP blocks. The concern with the first approach is that it is not amenable to fuzzing anything but a processor. IP cores generally do not have ISA-level inputs and although SoCs have one or more processors, obtaining coverage of the whole SoC is challenging if only operating at the ISA level. Conversely, operating at the RTL I/O is applicable to all IP, but is more useful for IP that is relatively small such that changes at the input can result in significant changes to its state. For larger designs, this is generally hard to achieve and will negatively affect the number of test inputs and time required to gain meaningful coverage of the design. The impact of the initial seed of the fuzzer, the first test it is given, has also yet to be investigated even though it can help guide the fuzzer toward more meaningful exploration. In the context of processor security, a meaningful seed might include a system call to guide it toward security-critical behavior, as this involves changing the privilege level. Another aspect of the inputs which has not been investigated is the set of mutations that a fuzzer can do. The possible mutations will depend on the form of the input which is a factor of the abstraction level, grammar, and granularity. Prior work has mainly relied on AFL-style mutations which they have assumed to be effective. To our knowledge, the optimal set of mutations that offers the best balance between speed and coverage has not been investigated.

c) Test Failure: The final challenge discussed in this paper is detecting when a fuzzer has encountered an error. Software errors trigger crashes (segmentation faults, etc.) but there is no equivalent for hardware. This makes it challenging to evaluate the results of fuzzing since it traditionally relies on these crashes to detect bugs. Many have taken the approach of *differential testing* to circumvent this, by feeding the same input to two representations of the same system and detecting when they behave differently. This is a reasonable approach for processors as there are many robust golden reference

models (GRMs) [29] and Instruction Set Architecture (ISA) simulators available for this purpose. However, there are no ISAs for non-processor designs, and GRMs are substantial feats of engineering themselves, making them prone to error. For IP where GRMs are not available, prior work has resorted to utilizing properties defined in SystemVerilog Assertions (SVAs) to verify the behavior of a design [16] but this makes the fuzzer largely dependent on the properties defined, another error-prone, manual effort.

2) RFUZZ: To our knowledge, [13] was the first work to utilize the term "fuzzing" for hardware. They define and use *mux control coverage* to evaluate their fuzzer and guide it towards interesting inputs. They decompose all muxes into 2:1 muxes, a straightforward task, and utilize the value of the select bit as the coverage metric. If a test input toggles the select bit of a mux to a new value, the input is deemed as 'interesting' and added to the set of test inputs so that it can be mutated and re-tested. Their mutations, inspired from AFL [30], operate at the bit-level and occur over two stages – deterministic and havoc – to obtain the most value out of every known interesting input. They tested their methodology on a range of designs from TileLink peripherals to RISC-V CPU cores, and find that their approach achieves mux coverage faster than random verification and is amenable to FPGA emulation which drastically speeds up verification time by up to 212x for the RISC-V Rocket core. Notably, RFUZZ's evaluation only demonstrated its ability to achieve coverage and not its effectiveness in detecting bugs. While this work left many open challenges, it nonetheless demonstrated that hardware fuzzing is possible and has inspired many others to attempt it.

3) DiFuzzRTL: DiFuzzRTL [14] improves on RFUZZ by introducing register coverage. They instrument the control registers, registers with outputs used as mux control signals because they better capture sequential events such as FSM state transitions. This also results in better time performance, as the output of a control register is used to control multiple muxes and can thus be monitored with less instrumentation added to the design. They identify control registers by first building a graph representation of the circuit, then starting at the control signal of a mux, they recursively traverse backward until they reach a register. If the traversal reaches the boundary of the circuit or reaches a point already traversed, it stops. This algorithm's time complexity is $O(V_2 * E)$, where V represents the number of circuit elements, and E represents the number of connections, which may cause issues for larger designs. The register coverage is obtained every clock cycle, a feat they state is not possible for mux coverage, by inserting three additional registers into every module: regstate, covmap, and covsum. The value of every control register in the module is hashed, an operation inspired by AFL, and stored in regstate. The value of regstate is used to try to update the appropriate bit in covmap from 0 to 1. If the write succeeds (i.e., the bit was not already 1), covsum is updated to signify that a new state was encountered. This process is replicated in every module of the design, where each parent module's covsum is the sum of its children's covsum and its own.

They are the first to apply differential testing in hardware fuzzing to detect potential bugs. They test their inputs on both an ISA simulator and an RTL simulator and compare the behavior of both simulations to detect potential concerns. Specifically, they first observe if both simulations complete successfully and then compare the architectural state (i.e., memory, etc.) to find any differences. The authors note that the use of an ISA simulator means they cannot detect microarchitectural errors, but that this can be partially resolved using assertions. This pitfall is applicable to all fuzzers that use differential testing with an ISA simulator to detect bugs.

4) ProcessorFuzz: ProcessorFuzz [15] also operates at the ISA level and improves on DifuzzRTL [14] by introducing Control and Status Register (CSR) transition coverage. Their intuition is that all meaningful state transitions of a processor will result in transitions in the CSRs. They utilize transitions rather than values because they believe that some bugs are dependent on both the old state and the new state of the processor, hence by using transitions they ensure that every processor state transition is tested. Furthermore, they restrict the CSRs used for coverage since some, such as *instret* which holds the number of retired instructions, would transition after every executed instruction and would misguide the fuzzer into thinking every instruction is interesting. They select which CSRs to monitor based on two criteria: (i) if it contains status information, like *mstatus*, which holds the reason for an exception, and (ii) if it contains configuration information, like *medeleg* which is used to delegate certain exception handling to modes below Machine mode.

Their second novel contribution is that they utilize ISA simulation to obtain coverage feedback, instead of RTL simulation. This speeds up simulation time, allowing feedback to be received faster – they find that ISA simulation was x79 faster than RTL simulation on average for the RISC-V BOOM processor. However, we note this approach is only possible because CSR values can be easily obtained from an ISA execution trace and cannot be extended to other aspects of the design that are less architecturally defined. They utilize an extended execution log obtained from the ISA simulation to determine if a value transition occurred in one or more CSRs of interest and add the test input to the seed corpus if that is the case so that it can be mutated and re-tested. They also filter out transitions that occur because of explicit writes to the CSRs. Finally, they store transitions in a tuple (I_m, S_0, S_1), where I_m is the instruction mnemonic (e.g., add), S_0 is the previous state of the CSRs, and S_1 is the new state. This ensures that different instructions that result in the same CSR transitions are still deemed interesting if they have never been tested before.

The authors demonstrate that their approach is faster than DiFuzzRTL, thanks to its ISA simulation coverage feedback while still obtaining line, toggle, FSM, and branch coverage values comparable to DiFuzzRTL. However, we note that these coverage values were not compared to any other fuzzer or any other verification methodologies and that FSM coverage, in particular, sits only slightly above 36%. We also note that they do not mention the number of instructions per test, which is crucial to determine how realistic the test inputs are. Finally, we speculate that the small time performance

improvement of the fuzzer (1.21-1.23x), when compared to the time performance of ISA simulation (79x), is because the limiting factor is the RTL simulation of the processor which must be completed to test the hardware behavior, not the coverage feedback mechanism.

5) Fuzzing Hardware like Software: Trippel et al. [16] argue that hardware is suitable to be fuzzed like software because it is already translated into a software representation for simulation, and there are many existing software fuzzers, such as AFL [30]. They utilize Verilator [31], a popular open-source simulator, to convert a design into a software model and instrument it with a software fuzzer. They first create a test harness using the TileLink Uncached Lightweight (TL-UL) protocol, a common bus protocol, to abstract away the I/O interface of the DUT. This enables the fuzzer to be used on all hardware designs that have the same bus protocol. To make this possible, they also utilize a bus-centric grammar which ensures that test inputs conform to the bus protocol and reach the DUT. To obtain coverage, they find that software fuzzers already provide a meaningful mechanism using binary instrumentation. They utilize *edge coverage*, which monitors the outcome that has been exercised in conditional statements, and observe that Verilator conveniently generates straight-line code that does not include loops.

In their evaluation, they find that instrumenting only the DUT and not including the testbench and simulation engine which it uses to test the DUT does not impair the fuzzer's ability to test a design, and improves fuzzing speed instead. Their experiments show that instrumenting only the DUT for coverage feedback reaches 100% FSM coverage faster, with the additional time required when including the testbench and engine decreasing as the number of states increases. This result can be explained by the fact that the added instrumented binary becomes less of a factor as the size of the DUT binary increases. Intuitively, it also makes sense that instrumenting only the DUT leads to better performance as coverage of the DUT is what we are interested in. They also determine that the overhead of resetting the DUT after each fuzz run is minimal. A crucial finding as fuzzing relies on resetting the DUT to a known state if an error is found or if a new fuzzing run is started.

6) TheHuzz: Kande et al. [17] propose TheHuzz, a RISC-V processor fuzzer. They define 6 coverage metrics that capture the behavior of hardware and demonstrate their effectiveness using case studies with CWEs. The first metric is branch coverage, this ensures that both branches of a conditional are taken. The second metric is condition coverage which ensures that every possible value combination in a conditional is tested. These two combined are akin to mux coverage defined in [13] since they guarantee that every possible outcome is tested. The third metric is FSM coverage which ensures that every possible value of the state encoding is tested. This is useful in catching potential weaknesses related to CWE-1245. The fourth coverage metric is expression coverage, this ensures that every possible value combination of an assignment with a combinational expression is tested. The fifth coverage metric is toggle coverage, this ensures that every DFF in a design takes on the value of 1, 0, and floating. Finally, statement coverage ensures that all lines of code are executed. This is the most comprehensive set of coverage metrics of any proposed hardware fuzzer.

They provide inputs at the ISA level, which are a set of instructions that are loaded into memory and executed during simulation. Each test input is thus a set of instructions separated into configuration instructions and test instructions. The configuration instructions set up the bare metal test environment while the test instructions are instructions used to fuzz the design. Each test run is limited to 20 test instructions, which raises the question of how "deep" into the design a fuzz test can get considering that real applications are made up of thousands of instructions. They utilize two types of mutations, one where only the opcode is mutated to modify the operation, and the other where the operand bits are mutated to modify the data being operated on. The authors also implement a novel optimizer to optimally select combinations of instructions and mutations. First, the profiler evaluates the data flow and control flow used by an instruction-mutation pair during a profiling stage. Second, the optimizer determines the smallest set of instruction-mutation pairs that covers all coverage points identified in the profiling stage. TheHuzz uses this information to determine weights for both instructions and mutations to steer itself to high-performing inputs. They do not instrument the design directly to obtain coverage feedback, rather they utilize the output logs from the RTL simulator. Finally, they evaluate the results of a fuzz run using differential testing by comparing the output of the RTL simulation to a GRM, the Spike ISA simulator, similar to other works.

TABLE II
OF BUGS FOUND

Design	Fuzzer	# of bugs found
Rocket Core	DiFuzzRTL	1
	ProcessorFuzz	2
	TheHuzz	1
Boom Core	DiFuzzRTL	7
	ProcessorFuzz	9
Mor1kx	DiFuzzRTL	5
	TheHuzz	3
CVA6	TheHuzz	4
OR1200	TheHuzz	3
BlackParrot	ProcessorFuzz	7

7) Summary: RFUZZ introduced hardware fuzzing and mux coverage. It is evaluated on various designs to show that it obtained coverage faster than CRV however, no bugs were reported. DiFuzzRTL extended RFUZZ by defining control register coverage and was evaluated using various processor cores. ProcessorFuzz introduced CSR coverage. TheHuzz utilized 6 coverage metrics and a novel instruction-mutation optimization technique. Fuzzing Hardware like Software demonstrated the ability to use a software fuzzer on hardware designs converted to "equivalent" C++ using Verilator.

We focus the rest of the discussion on the three processor fuzzers evaluated on the same processor designs. Rocket core was used to evaluate all three fuzzers, with all three

discovering a bug where the retired instruction count does not increase when on an EBREAK instruction. An additional bug found by ProcessorFuzz was not discovered by the other 2. The Boom core was used to evaluate both DiFuzzRTL and ProcessorFuzz. ProcessorFuzz was able to find all bugs found with DiFuzzRTL and an additional 2 bugs. Mor1kx was used to evaluate TheHuzz and DiFuzzRTL, and each found 3 and 5 bugs, respectively. Surprisingly, these bugs had no overlap and all 8 bugs were unique. This interesting result suggests that the defined coverage metrics and mutations impact more than just the speed of coverage. It is impossible to tell if these two fuzzers explored mutually exclusive sets of states and we argue that more investigation must be done to determine the impact that these characteristics can have on the explored states. Across the board, we observe that the # of bugs found is fairly similar for each core, with ProcessorFuzz pulling slightly ahead of DiFuzzRTL.

Next, we compare the coverage obtained by the fuzzers. ProcessorFuzz finds that it performs similarly to DiFuzzRTL when considering the industry-standard line, toggle, FSM, and branch coverage. This result is interesting considering the fact that ProcessorFuzz found more bugs than DiFuzzRTL. It suggests that although the amount of coverage is similar, ProcessorFuzz's CSR coverage metric guides the fuzzer toward more bug-prone sections of the design. The coverage obtained by TheHuzz was also compared to DiFuzzRTL's for the Rocket core using the total amount of coverage points, as defined by TheHuzz. After 1,000,000 instructions, TheHuzz was able to obtain approx. 20,000 more coverage points than DiFuzzRTL. Although this seems significant at first, no additional bugs were found thanks to those coverage points and DiFuzzRTL's coverage is a subset of the coverage defined by TheHuzz so this is an expected result.

C. Security Property Generation

One of the biggest challenges in security verification is developing a comprehensive set of security properties that convey the intended, secure behavior of a design. Given such a set of properties, one can leverage methods like simulation or formal verification to verify the design and also use them as documentation embedded inside of the design. This an extremely difficult task because one must have in-depth knowledge of the specific design at hand and the requirements which it must fulfill. To address this challenge, hardware security property generation draws inspiration from hardware functional property generation, and software functional and security property generation. To our knowledge, all prior work on hardware security property generation has focused on mining properties from execution traces. The basis of this approach is to simulate an RTL design to obtain its execution traces – the value of every signal in the design at every clock cycle – and mine properties from that information using miners like Daikon and Texada. This implies that a set of testbenches that obtain sensible coverage of the design is available. A misconception might be to believe that this also requires a bug-free design. This is not the case since a bug in the design would likely manifest as an incorrect property, which would still warn of the presence of bugs. One nuance is that the impact of a bug that is triggered sometimes, but not all the time, would be up to the miner used.

Security property generation has proven to be a promising avenue, with success in generating a variety of properties. By taking humans out of the loop, partly or completely, we remove the error that can be introduced with incorrect assumptions, bias, etc., and simply summarize and convey how the design operates in a succinct and explicit manner. This is also extremely useful in combination with property translation [32], as this set of properties can be used as a starting point to verify other, perhaps incomplete, designs. The generated properties on their own are extremely powerful since they can be used in a breadth of existing verification methods such as simulation and formal verification, and emerging techniques like hardware fuzzing IV-B to detect errors. Another strength of these properties is that they do not specify what a bug might "look like", they specify what the behavior of a design must be to be "correct", hence one property can be used to detect multiple bugs provided they all violate the same specified behavior. Finally, these techniques also focus on security-critical properties, that are generally more abstract in nature and thus harder to explicitly specify, and further compounded by the lack of security expertise in the standard design/verification design flow.

This specific approach to property generation suffers from a few shortcomings. First, as mentioned earlier, the generated properties largely depend on the RTL design and testbenches used to simulate it. Any defects in the design or testbenches may thus cause issues in the properties that are difficult to isolate. The backward path from property error to root cause is not straightforward as it may be the result of lackluster coverage of the testbench, a bug in the design, or a combination thereof. Even worse, if the erroneous property is not caught and used to test a design, the property may trigger and cause a false positive. Secondly, the runtime requirement makes this approach currently unpractical for real-life designs. The repeated simulation of a design takes a considerable amount of time, which is further compounded by the time needed by the miner used to generate properties from the traces. The final shortcoming we point out is the misclassification of functional properties to security properties. Misclassified functional properties are effectively noise that may distract the user from the valuable security properties that they are searching for. In general, the line between functional and security is not always clear and depends greatly on the context under consideration.

1) SCIFinder: To our knowledge, SCIFinder [18] was the first attempt to automate the generation of security-critical hardware properties. They propose a semi-automated approach by (i) generating invariants describing the general behavior of a processor, (ii) using published silicon errata to classify a subset of the invariants as functional bugs or security bugs, and (iii) using machine learning to utilize the classified subset to classify the remaining invariants. The invariants describe the post-conditions of a given instruction. To collect the invariants, they simulate the RTL of a processor with a variety of software tests, collect execution logs of the simulations, and use a

modified version of Daikon to infer a set of invariants. They operate at the ISA level, whereby only the software-visible state is monitored and one instruction is considered an execution step. This should provide the advantage of applicability of the generated properties to all processors implementing the same ISA. They perform the following set of optimizations on the collected invariants: constant propagation, deducible removal, and equivalence removal. Constant propagation refers to using invariants that describe assignment to a constant to simplify other invariants. Deducible removal is the removal of invariants that are implicitly defined from other invariants. Equivalence removal is the removal of equivalent invariants that describe the same semantic property.

The classification of the invariants is done in a two-step approach. First, the published security errata of processors are studied to find real-life security bugs from previous designs. The processor design under test is then modified to intentionally introduce a bug. Finally, an exploit that exploits the newly introduced bug is written and used to simulate the design and collect execution traces. The execution traces are used to identify which invariant(s) are violated by the new buggy design and are subsequently marked as security-critical. The second step consists of using machine learning techniques to infer if the remaining invariants are security-critical. Given a set of security-critical and non-security-critical invariants, they implement a logistic regression model which can predict if a given invariant is also security-critical. The authors discuss two possible sources of false positives in the final set of security-critical invariants. The first source of error is an initial set of invariants that is incorrect due to errors from Daikon, an improper set of benchmarks to generate the invariants, or a processor design that was already buggy. These concerns can be partially addressed by using a coverage-driven set of benchmarks and using an ISA simulator to obtain the execution traces for the initial set of invariant generation, which should be possible if only using the architectural state. The second source of error is the incorrect classification of invariant which is a result of incorrect classification by the machine learning model which can be partially addressed with more training data.

2) Undine: Undine [19] extends SCIFinder by mining for Linear Temporal Logic (LTL) properties. Similarly to SCIFinder, a processor's RTL design is first simulated to obtain execution traces. However, before mining, they introduce a pre-processing step that converts traces into a filtered set of *typed events*. They define five events types: (1) register-register (RR): two registers have the same value, (2) delta-register (DR): a register changes by a value y after a clock cycle, (3) register-value (RV): a register has a value y, (4) slice-value (SV): a slice of a register has a value y, and (5) bit-value (BV): a bit of a register has a value y. Their observation is that security-critical registers are used in predictable ways in properties, which they capture through these five events by associating each register to a specific type (i.e. each register can only be used for one type). They define property templates that contain specific event types. This mapping enables the miner to filter out registers whose types are not present in a given property template. They utilize the Texada LTL Specifications Miner, which takes a trace of events and a property template and initializes a set of properties that are true for the given trace, and modify it to handle typed events. They also include a post-processing step to optimize the set of properties by grouping them by antecedents and combining consequents using AND operations, or grouping them by consequents and combining antecedents by OR operations.

We identify a few limitations of this approach. First, as the authors discuss, the properties generated by Undine is limited to the user-provided templates. They note that Undine mined 25 of the 28 known properties for the designs used in their evaluation. The missing three resulted from a missing template from Undine's grammar. While this seems like a promising result, the underlying issue implicates that any unknown or undefined template leaves out a whole category of bugs. The other concern is the need for manual information for the classifications of registers to typed events. This is a crucial aspect of Undine as the authors demonstrate that it is impractical to mine for LTL properties with no typing. However, this involves manually classifying every register of a design as security-critical or not and further associate it with a specific type. Any error could quickly lead to missed or incorrect properties, whose root cause would be hard to identify.

3) Isadora: Isadora [20] is another promising work in property generation. Isadora automatically generates information flow properties by combining information flow tracking (IFT) and property mining. Information flow properties are *hyperproperties* that are normally not mineable using regular trace property techniques, however, the authors demonstrate that this is circumventable by first instrumenting the design with IFT. First, IFT logic is added to the original design and it is subsequently simulated using user-provided testbenches and an industry-grade IFT simulation tool. Second, the execution traces of the IFT simulations are used to identify all flows and no-flows that occurred. Then, Daikon is used to identify the flow conditions for conditional flows. Unconditional flows are also detected but are not useful in the context of security verification. However, they are useful for designers to validate design intent and can be used as documentation. Finally, a post-processing step optimizes the set of flow conditions to generate the final set of conditional flow properties. These, combined with the no-flow properties, represent the set of information flow properties that Isadora generates. Isadora also has the ability to combine flows that occur at the same time and depend on the same condition into multi-source to multi-sink flows which reduce the number of properties required to convey the behavior of the design.

Although Isadora is promising, it has a few limitations. First, Isadora is heavily reliant on user-provided testbenches, which it utilizes to stimulate the DUT and obtain execution traces. The flow properties that are mined from these traces are thus heavily reliant on the quality of the testbenches. The authors did not assess the quantity or quality of their testbenches or discuss how they constructed them. Second, Isadora takes a significant amount of time to generate properties. The authors evaluated Isadora on 2 designs, the Access Control Wrapper (ACW) proposed in [33] and the PicoRV32 RISC-V

CPU. The Multi ACW design which resembles an SoC the closest, consists of two AXI controllers, an AXI interconnect, and three peripherals, for a total of 984 unique signals and 4447 LoC. The trace generation for this design took over 24 hours which makes it very unpractical and unusable for larger, "real-world" designs. We believe this is a result of their trace generation method, which only tracks one source signal for each testbench simulation. Finally, Isadora does not differentiate between functional and security properties. The authors reported a 10% false positive rate with regard to misclassification, but this number was only determined by taking a sample of 10 properties – this perfectly demonstrates why this is a limitation. In a "real-world" situation, the design/verification engineer would have to manually verify all generated properties to determine if they are relevant, an arduous task given the high amount of properties.

4) Summary: The first work that demonstrated hardware security property generation was SCIFinder. They generate invariants describing the post-conditions of an instruction. Undine extends this by generating LTL properties that specify behavior over multiple instructions. Isadora generates information flow properties by first instrumenting a design with IFT. While SCIFinder uses a semi-automated approach to classify security properties by manually injecting bugs, Undine search for security-criticality by limiting itself to security-critical registers. SCIFinder's logistic regression model reduced the set of mined invariants from 88,199 to 3,146, however, the authors report a total of 852 false positives (invariants incorrectly labeled as security-critical). The authors of Undine do not report any false positives but their approach inherently restricts them to registers they deem as "security-critical". They show that their approach is not practically feasible when considering every register in a design as the operation takes over 4 hours. Isadora does not attempt to filter out functional properties and the authors report a 10% false positive rate. It is difficult to gain any valuable sense of performance from these metrics as they all take different approaches and target different subsets of behavior.

Next, we compare based on the number of previously defined properties these works were able to generate. SCIFinder generates 19 of the 22 (86%) properties previously defined in SPECS [34] and Security-Checker [35] that are applicable to the architectural state of a processor. It also generates 3 additional properties not previously defined. Undine cannot make any comparisons as, to our knowledge, there is no available source of LTL properties for the evaluated designs. Isadora fully covers the properties defined in [33] for the single ACW design using only 9 of the 303 generated properties, but no comparison can be made for the PicoRV32 design for the same reason as Undine. Overall these are promising results as they show that these techniques are able to define existing properties and new ones. One of the biggest questions about the practicality of these three techniques is if the extra generated properties are worth the substantial time commitment required.

D. Discussion

The three discussed techniques take different approaches to assist in bug detection. Static code analysis requires the least supplementary effort but also the weakest guarantees. It does not require any environment setup, test harness, etc., and can theoretically even operate on incomplete designs. This makes it applicable to practically all stages of design and validation. It is also the least time-consuming technique, with CWEAT taking minutes to analyze complete SoCs. Fuzzing and property generation are only applicable in the validation stage. They both require complete designs. Fuzzing also requires a mechanism to evaluate if an error has occurred, generally an ISA simulator for processors and assertions for other IPs. Property generation requires an extensive set of testbenches to simulate a design and considerable time investment to generate the traces that it relies on. They also both take time in the number of hours to complete. This is a significant investment, especially in the case of property generation, where there is still much manual effort involved in reviewing the generated properties for quality. Fuzzing's time cost is easier to justify, as it resembles the industry standard CRV and may partly replace it as such. It is noteworthy to mention that property generation and fuzzing have not been demonstrated on a whole SoC yet, and putting aside the challenges of making such a thing possible, will require even more time invested. Another factor to consider is the cost to repair any issues found. Since static analysis is "cheap" (time, effort, etc.), it can be used through the design stage as sanity checks by designers which can lead to issues being found earlier. This can tremendously reduce the cost of repairing the error and result in a higher-quality final product. Conversely, since fuzzing and property generation can only occur during the validation stage, any errors would be more costly and challenging to fix and companies may decide to address them with software workarounds instead, still leading to vulnerable hardware.

Fuzzing has shown the most promise in finding bugs in real designs and does not suffer from "false positives" as static analysis does. Every detected error in fuzzing is an error in the design or the GRM in the case of differential testing. This is demonstrated by the fact the only works studied in this paper that found "real" bugs which were not previously known were all hardware fuzzers [14], [15], [17]. Although property generation has never been used to find new bugs, its greatest strength is that it can specify secure behavior that humans may otherwise not have found. These properties are also extremely versatile and have the ability to be applied with "standard" verification approaches such as simulation, formal verification, and even fuzzing [16]. They also effectively condense the behavior of a design and can act as embedded documentation inside of the design. Static analysis, by nature, is not as concrete as the other two and can currently only provide warnings of possible issues, with many false positives. It takes a more generic approach which has the trade-off of wide applicability but low precision. The less contextual is considered (e.g., signal name), the more false positives are outputted. However, the issues found by static analysis are highly localized to a certain range of source

code. This is extremely useful when assessing and repairing bugs. Fuzzing provides the input(s) that lead to the detected error. The verification engineer verifying any errors must then consider the error itself and the input(s) to localize the bug, a non-trivial effort. It is a similar idea for property generation. Regardless of the technique with which the properties are used, it will only trigger to suggest there is an error but will not localize it. It is up to the verification engineers to consider the output of the test (e.g., waveform) to reason about the error.

With all of these considerations, it is clear that a complementary approach is required for the holistic security evaluation of a design. Static analysis can be used throughout the design stage, where it can provide feedback early and fast. Then, in the validation stage, property generation can be used with the final design and the testbenches created for general verification. Finally, fuzzing can be used in tandem with other verification approaches, leveraging the assertions from property generation for error detection.

REFERENCES

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[2] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution," 2018.

[4] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, p. 88–105.

[5] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *Usenix Security Symp.*, 2019, pp. 213–230.

[6] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, p. 971–987, Dec 2006.

[7] G. S. Walia and J. C. Carver, "A systematic literature review to identify and classify software requirement errors," *Information and Software Technology*, vol. 51, no. 7, p. 1087–1109, Jul 2009.

[8] Intel, "Building More Secure Technology with Intel SDL," https://newsroom.intel.com/articles/building-more-secure-technology-intel-sdl.

[9] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, "Survey of approaches and techniques for security verification of computer systems," 2016.

[10] P. Kidwell, "Stalking the elusive computer bug," *IEEE Annals of the History of Computing*, vol. 20, no. 4, p. 5–9, Oct 1998.

[11] Microsoft, "Microsoft Security Development Lifecycle," https://www.microsoft.com/en-us/securityengineering/sdl.

[12] B. Ahmad, W.-K. Liu, L. Collini, H. Pearce, J. M. Fung, J. Valamehr, M. Bidmeshki, P. Sapiecha, S. Brown, K. Chakrabarty, R. Karri, and B. Tan, "Don't cweat it: Toward cwe analysis techniques in early stages of hardware design," Sep 2022, arXiv:2209.01291 [cs].

[13] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2018, p. 1–8.

[14] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, p. 1286–1303.

[15] S. Canakci, C. Rajapaksha, A. M. Nataraja, L. Delshadtehrani, M. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Guiding processor fuzzing using control and status registers," Sep 2022.

[16] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," no. arXiv:2102.02308, Feb 2021, arXiv:2102.02308 [cs].

[17] A. Tyagi, A. Crump, A.-R. Sadeghi, G. Persyn, J. Rajendran, P. Jauernig, and R. Kande, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," no. arXiv:2201.09941, Jan 2022, arXiv:2201.09941 [cs].

[18] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, Apr 2017, p. 541–554.

[19] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, Dec 2018, p. 18–23.

[20] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, "Isadora: Automated information flow property generation for hardware designs," in *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*. Association for Computing Machinery, Nov 2021, p. 5–15.

[21] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, Jul 2022, p. 544–555.

[22] D. Wheeler, "flawfinder," https://github.com/david-a-wheeler/flawfinder.

[23] D. Marjamäki, "cppcheck," https://github.com/danmar/cppcheck.

[24] Synopsys, "Coverity Static Application Security Testing," https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

[25] ——, "SpyGlass Lint," https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html.

[26] MITRE, "CWE VIEW: Hardware Design," https://cwe.mitre.org/data/definitions/1194.html.

[27] E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati, and G. Uddin, "An empirical study on bug severity estimation using source code metrics and static analysis," no. arXiv:2206.12927, Jun 2022, arXiv:2206.12927 [cs].

[28] A. B. Mehta, *Constrained Random Verification (CRV)*. Cham: Springer International Publishing, 2018, p. 65–74.

[29] Imperas. ImperasDV - industrial quality risc-v processor verification made easy. URL: https://www.imperas.com/imperasdv (Accessed: 03-10-2022).

[30] "Technical "whitepaper" for afl-fuzz," https://lcamtuf.coredump.cx/afl/technical_details.txt, last Accessed: 2022-10-06.

[31] W. Snyder, "Verilator," https://www.veripool.org/verilator, last Accessed: 2022-10-06.

[32] R. Zhang and C. Sturton, "Transys: Leveraging common security properties across hardware designs," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, p. 1713–1727.

[33] F. Restuccia, A. Meza, and R. Kastner, "Aker: A design and verification framework for safe and secure soc access control," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Nov 2021, p. 1–9.

[34] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*. Istanbul, Turkey: ACM Press, 2015, pp. 517–529.

[35] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, Jun. 2011, pp. 34–39, iSSN: null.