Semester Long Internship Spring 2026

On

eSim Hardware Security Linter: DevSecOps Integration for Physical PCB Vulnerability Detection

Submitted by

Saksham Gupta
B.Tech CSE, 3rd Year
VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran
Principal Investigator
Department of Aerospace Engineering
Indian Institute of Technology Bombay

May 14, 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, develop IC subcircuits in eSim, and gain exposure to real-world circuit modeling and simulation workflows. The knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Evolution of Hardware Security

The evolution of electronics design over the past two decades has been defined by exponential increases in component density, operating frequencies, and system complexity. Historically, Printed Circuit Boards (PCBs) were viewed merely as mechanical supports and electrical conduits for discrete components. Security, therefore, was traditionally treated almost exclusively as a software domain. Billions of dollars have been invested in developing robust cryptographic algorithms, Trusted Execution Environments (TEEs), secure boot protocols, and advanced firmware encryption standards.

However, as software security hardens, malicious actors consistently seek the path of least resistance. The assumption that a device's hardware is intrinsically secure relies on the outdated premise that attackers will not have persistent physical access to the device. In the modern era of ubiquitous computing, this assumption is fundamentally flawed. When an attacker has physical possession of a device, the most sophisticated software encryption can be completely bypassed by attacking the underlying physical hardware layer.

During the prototyping and development phases, hardware engineers frequently place debugging interfaces—such as Universal Asynchronous Receiver-Transmitter (UART), Joint Test Action Group (JTAG), Serial Wire Debug (SWD), and Serial Peripheral Interface (SPI) Flash ports—directly near the physical edges of a PCB. This is done to facilitate easy connection to test jigs, oscilloscopes, and debugging probes without needing to completely disassemble the prototype. If these ports are not physically removed, permanently disabled (e.g., via blowing eFuses), or cryptographically secured before the final production devices reach consumers, they create a massive physical vulnerability surface. An attacker can perform invasive probing or Electromagnetic Fault Injection (EMFI) attacks by accessing these exposed boundary ports, effectively compromising the device's Root of Trust (RoT).

1.2 The IoT and Embedded Systems Threat Model

The proliferation of the Internet of Things (IoT) has drastically altered the threat model for hardware security. Unlike enterprise servers locked in climate-controlled data centers with biometric access controls, IoT devices—such as smart utility meters, automotive Electronic Control Units (ECUs), medical implants, and consumer smart home devices—are deployed in completely uncontrolled physical environments.

Because these devices exist in the public domain, it must be assumed that highly motivated attackers will acquire them, dismantle them, and subject them to rigorous reverse-engineering attempts. If an attacker can extract proprietary firmware binaries or cryptographic keys from a single smart meter by exploiting an exposed JTAG port near the edge of the PCB, they can potentially reverse-engineer the communication protocols and compromise the entire metropolitan smart grid. Thus, physical layout security is no longer an optional luxury; it is a critical baseline requirement for modern embedded systems.

1.3 Overview of the FOSSEE eSim and KiCad Ecosystem

The Free/Libre and Open Source Software for Education (FOSSEE) project, based at the Indian Institute of Technology Bombay (IITB) and funded by the Ministry of Education (MoE), Government of India, is dedicated to promoting the adoption of open-source software in engineering education and research.

A flagship outcome of this initiative is **eSim**, a comprehensive Electronic Design Automation (EDA) ecosystem. eSim seamlessly bridges the gap between circuit schematic creation, ngspice-based analog/digital simulation, and physical PCB layout. For the physical layout phase, eSim deeply integrates with **KiCad**, the world's leading open-source PCB design software.

By developing native **Action Plugins** for the KiCad environment, we can significantly extend the capabilities of the eSim ecosystem. KiCad Action Plugins utilize a powerful Python API (`pcbnew`) that hooks directly into KiCad's underlying C++ memory structures. This allows engineers to programmatically manipulate graphical objects, extract copper netlists, and automate complex verification tasks that would otherwise require painstaking manual review. Developing this tooling within the FOSSEE ecosystem ensures that enterprise-grade DevSecOps capabilities are democratized and made freely available to students, academic researchers, and professional PCB engineers worldwide.

1.4 Specific Physical Hardware Threats

To understand the necessity of a physical hardware linter, it is critical to analyze the specific attack vectors it aims to mitigate:

1.4.1 Boundary Probing and Firmware Extraction

Probing attacks involve an attacker physically attaching a logic analyzer, an oscilloscope, or a hardware debugger (such as a Segger J-Link or a Bus Pirate) to exposed test points or debug headers on the PCB. If an engineer carelessly routes a UART TX/RX line or a SWDIO header dangerously close to the plastic enclosure's edge, an attacker does not even need to decap the device. They can simply drill a microscopic hole through the plastic chassis, insert pogo-pins, and interface directly with the debug port to dump plaintext system logs or extract the raw firmware binary.

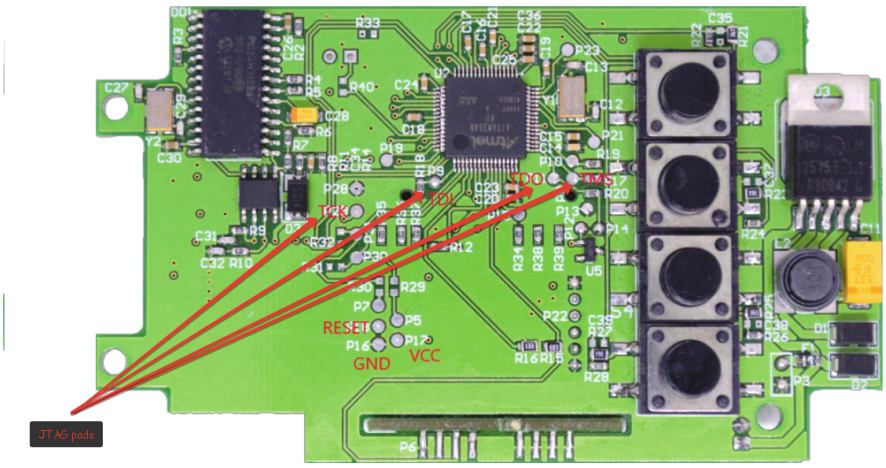

Figure 1.1: A conceptual visualization of an invasive hardware attack (e.g., interfacing with an exposed UART/JTAG port).

1.4.2 Electromagnetic Fault Injection (EMFI)

EMFI attacks are highly sophisticated side-channel attacks that involve inducing transient voltage faults in a processor's memory buses by exposing sensitive copper traces to focused magnetic pulses. Attackers use high-voltage coils to generate a magnetic field that couples with the PCB traces, creating localized eddy currents.

The physics of electromagnetism dictate that the effectiveness of a magnetic pulse decays exponentially with distance (following the inverse-square law). Therefore, EMFI probes require extremely close proximity to the target traces to successfully inject sufficient energy. If critical traces (such as the processor's reset line, clock line, or memory bus) are routed along the absolute periphery of the board to save routing layers, they become ideal targets for EMFI. Attackers can simply place the EMFI coil against the outside of the plastic case, directly over the peripheral trace, and successfully glitch the processor into bypassing secure boot verification.

1.5 Objectives and Scope of the Project

The primary objective of this semester-long internship project was to bring the concept of "Shift-Left Security" to the hardware domain. In the software industry, "shifting left" refers to the practice of integrating automated security testing as early in the development lifecycle as possible, typically using Static Application Security Testing (SAST) tools integrated into the CI/CD pipeline.

Instead of discovering hardware vulnerabilities after the PCB is manufactured, fabricated, and deployed—where the cost of remediation is astronomical—the goal of this project was to build a sophisticated **eSim Hardware Security Linter**.

This tool was engineered with the following specific objectives:

1. **Mathematical Detection**: To actively parse the physical coordinate geometry of the PCB and calculate the absolute Euclidean distance of sensitive components from the board's physical edges.

2. **Deep Copper Inspection**: To transcend basic component metadata and analyze the actual electrical netlist associated with individual copper pads using Regular Expressions.

3. **Visual Integration**: To provide immediate, graphical feedback to the design engineer directly on the KiCad canvas using custom drawing layers.

4. **DevSecOps Automation**: To bridge the gap between local EDA software and cloud infrastructure by automatically generating GitHub Actions workflows and implementing a "Pre-Push Firewall" that physically blocks the uploading of vulnerable hardware designs to production repositories.

Chapter 2

Literature Survey

2.1 Limitations of Traditional and Commercial EDA Tools

A comprehensive review of the existing electronic design landscape—ranging from commercial titans like Altium Designer, Cadence Allegro, and Mentor Graphics, to open-source suites like KiCad and Fritzing—reveals a structural blind spot regarding physical hardware security.

Modern Electronic Design Automation (EDA) tools rely heavily on Design Rule Checks (DRC) and Electrical Rule Checks (ERC) to validate a board before it is sent to a fabrication facility. These engines are phenomenally advanced at ensuring manufacturability and signal integrity. They rigorously evaluate trace-to-trace clearance, calculate differential pair impedance, enforce minimum drill hole annular rings, and detect overlapping silkscreens that might obscure component designations.

However, these engines evaluate the board purely from an electrical and manufacturing standpoint. Standard DRC systems do not evaluate the spatial placement of components through the lens of an adversarial threat model. Standard KiCad plugins and built-in rules often rely solely on schematic attributes or electrical connectivity. A JTAG port or an exposed UART terminal placed directly on the perimeter of the board—or even left entirely unshielded on the top copper layer—will pass all standard DRC checks with zero warnings, despite representing a catastrophic security failure for a production device. The literature shows that while electrical validation is fully automated, security validation remains entirely manual.

2.2 Probing and Physical Extraction Techniques

Academic literature and vulnerability reports from security researchers heavily document the exploitation of exposed debug interfaces. The Hardware Hacking Handbook by Colin O'Flynn outlines the varying degrees of physical attacks: non-invasive, semi-invasive, and fully invasive.

Fully invasive attacks require stripping the epoxy resin off integrated circuits (decapping) and using focused ion beams (FIB) to tap into microscopic die-level traces. However, the literature overwhelmingly concludes that attackers rarely need

to resort to such expensive, million-dollar laboratory techniques. Instead, attackers capitalize on poor PCB layout choices.

If an engineer routes sensitive test points, SWDIO headers, or SPI Flash traces along the physical boundary of the PCB, the attack becomes functionally non-invasive. An attacker can simply drill a microscopic hole through the plastic chassis of an IoT device, insert spring-loaded pogo pins, and attach a logic analyzer (such as a Saleae or a Bus Pirate). This allows them to passively sniff memory buses, intercept plaintext cryptographic keys being transferred between the microcontroller and the SPI Flash, or actively halt the CPU to dump the proprietary firmware binary.

2.3 Electromagnetic Fault Injection (EMFI)

Beyond passive probing, the literature emphasizes the severe threat of active fault injection. Electromagnetic Fault Injection (EMFI) is a highly effective side-channel attack where an adversary uses a high-voltage pulse generator connected to a magnetic coil to induce localized eddy currents within the PCB's copper traces.

By precisely timing these magnetic pulses during the device's boot sequence, an attacker can induce transient voltage drops that cause the processor to skip instructions—such as bypassing the password verification check or jumping over the secure boot cryptographic signature validation.

The critical limiting factor of EMFI, according to Faraday's law of induction and the inverse-square law, is physical distance. The magnetic field's intensity decays exponentially as the coil is moved away from the target trace. Therefore, EMFI probes require absolute physical proximity to the target traces to inject sufficient energy. Traditional layout practices often encourage routing "noisy" power planes or low-priority debug traces along the periphery of the board to save prime routing real estate in the center. This practice inadvertently creates an ideal attack surface, allowing an attacker to place an EMFI coil against the outside of the plastic enclosure to achieve a successful glitch, completely negating the need to dismantle the device.

2.4 The Rise of DevSecOps in Hardware Design

In the software engineering realm, DevSecOps (Development, Security, and Operations) is a mature and widely adopted philosophy. Modern CI/CD pipelines successfully automate software vulnerability scanning. Source code is aggressively analyzed by Static Application Security Testing (SAST) tools and Dynamic Application Security Testing (DAST) frameworks. If a developer accidentally commits a plaintext AWS API key or writes a SQL injection vulnerability, the DevSecOps pipeline automatically fails the build, preventing the code from ever being merged into the production branch.

Conversely, applying these identical automated CI/CD gating principles directly to hardware schematic and layout files remains an emerging, bleeding-edge, and largely unsupported field. A review of current industry practices shows that most hardware engineering teams still rely on manual "Hardware Design Review" meet-

ings. During these meetings, engineers visually inspect the Gerber files or KiCad canvas to check for security flaws.

These manual reviews are notoriously prone to human error, visual fatigue, and oversight—especially on complex, 12-layer boards with tens of thousands of individual nets. The literature underscores a pressing need to bridge this gap. The lack of an automated "Hardware DevSecOps" pipeline that can programmatically enforce physical layout security rules is the core deficiency that the eSim Hardware Security Linter project aims to solve.

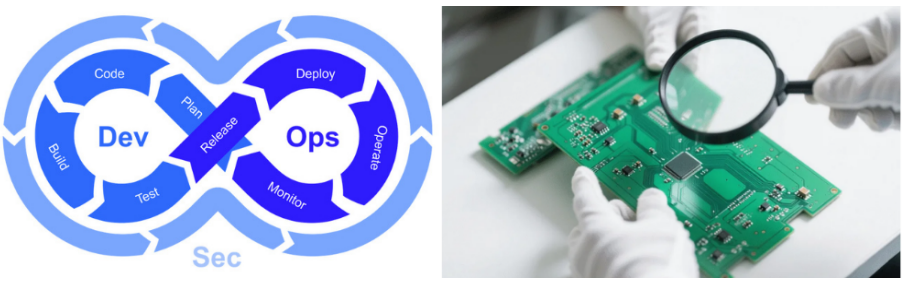

Figure 2.1: The DevSecOps gap: Highlighting the disconnect between automated software CI/CD and manual hardware reviews.

Chapter 3

Problem Statement and Scope

3.1 Problem Statement

Hardware security is structurally disadvantaged in the electronic design lifecycle because it is almost always treated as a secondary concern compared to functional electrical design and manufacturability.

The core problem this project addresses is the human element in PCB layout. During the prototyping phase of hardware development, engineers prioritize testability. They frequently place sensitive debugging interfaces—such as JTAG, SWD, UART, and logic analyzer test points—directly along the absolute perimeter of the printed circuit board. Placing these headers near the edge makes it significantly easier to plug in ribbon cables, oscilloscope probes, and FTDI programmers while the board is mounted in a test jig.

However, as projects transition from prototype to mass production, these debug ports are often inadvertently left in the layout. This occurs because moving the components inward would require completely ripping up and re-routing complex copper traces, potentially violating impedance constraints or delaying the manufacturing schedule. Consequently, vulnerable ports are shipped in consumer products, resting mere millimeters behind a thin plastic enclosure, creating a trivial attack vector for malicious actors equipped with pogo-pins or EMFI coils.

3.1.1 The Illusion of Local Security Warnings

A secondary, yet equally critical problem, is the disconnect between local Electronic Design Automation (EDA) software and remote cloud infrastructure. Even if an engineer utilizes a basic local script or a manual checklist that warns them about a vulnerable UART port, the human engineer retains the ultimate authority to simply ignore the warning.

Because traditional EDA tools operate entirely independently of version control systems (like Git), there is no programmatic oversight. An engineer under a strict deadline can effortlessly bypass local security warnings and push their vulnerable hardware design to a production repository on GitHub or GitLab. The lack of a structural enforcement mechanism—an automated connection between the local KiCad software and the cloud-based CI/CD pipeline—guarantees that vulnerable

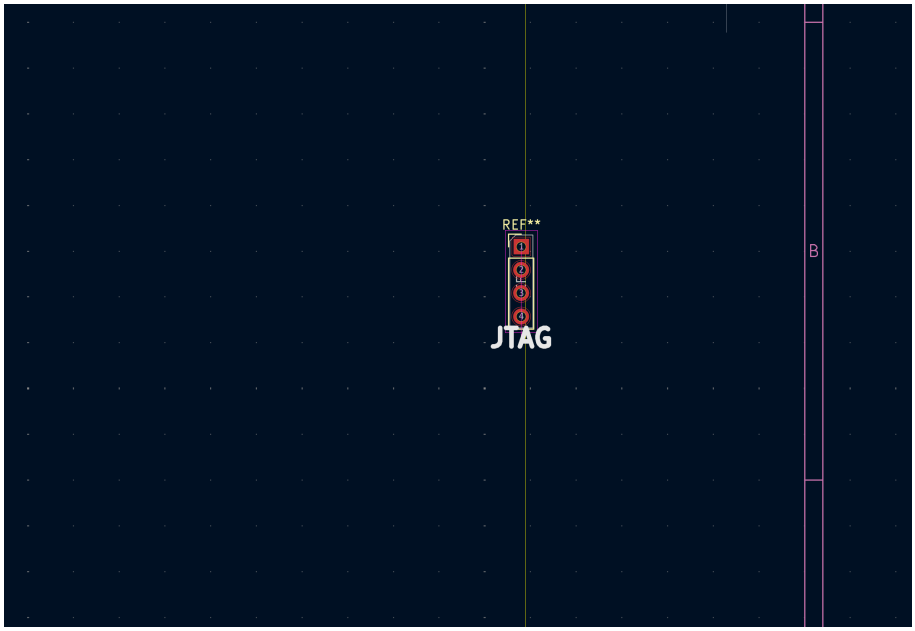

Figure 3.1: A typical design oversight: A JTAG port left completely exposed on the absolute perimeter of a production PCB.

hardware will eventually reach production.

3.2 Scope of the Project

To effectively solve both the mathematical routing problem and the human compliance problem, the scope of this project encompasses the development of a complete, end-to-end enterprise security solution for KiCad PCB layouts within the FOSSEE eSim ecosystem.

To achieve a true "Secure-by-Design" environment, the developed solution must fulfill the following comprehensive architectural requirements:

3.2.1 1. Mathematical and Spatial Evaluation Engine

The solution must feature a localized mathematical engine capable of evaluating physical PCB space. It must be able to programmatically extract the absolute geometry of the board (the `Edge.Cuts` boundary layer) and calculate the precise Euclidean distance of target components from the enclosure's physical edges, compensating for rotational offsets and non-standard board shapes.

3.2.2 2. Deep Copper Netlist Scanning

Standard hardware linters rely on high-level component metadata or footprint library names (e.g., assuming a component named `Conn_01x04` is safe because it does not explicitly say `UART`). The scope of this project requires the development of a "Deep Copper Scanner." The algorithm must dive into the physical copper, extract the underlying electrical net names bound to individual pads, and evaluate those

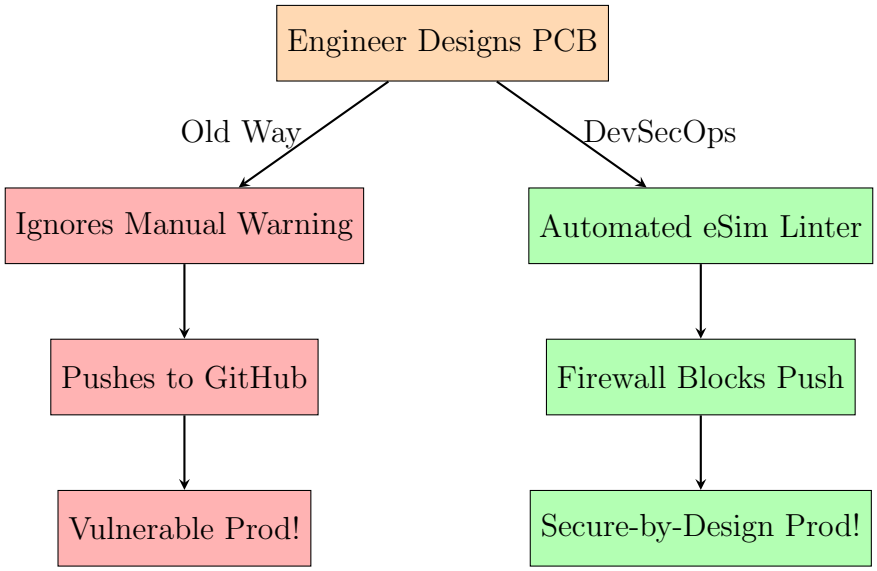

Figure 3.2: Traditional vs. Automated DevSecOps Workflow

nets against a comprehensive dictionary of known sensitive strings using aggressive Regular Expressions (Regex).

3.2.3 3. Real-Time Graphical User Interface (GUI)

Security tools are only effective if they are intuitive. The project must integrate a native `wxPython` graphical interface directly into the KiCad software. This GUI must allow engineers to seamlessly tune security tolerances (Critical vs. Warning thresholds) and must provide immediate visual feedback by drawing high-visibility red markers directly over vulnerable components on the designer's canvas.

3.2.4 4. DevSecOps and Pre-Push Firewall Automation

To solve the human compliance problem, the tool must act as a bridge between the local EDA environment and cloud infrastructure. The scope requires the implementation of a "Pre-Push Firewall"—a mechanism that violently intercepts standard Git communications. If the mathematical engine detects a physical hardware vulnerability, the plugin must actively abort the `git push` sequence, physically preventing the vulnerable code from leaving the local machine.

3.2.5 5. Headless CI/CD Cloud Infrastructure

Finally, the project must extend into the cloud. The plugin must be capable of automatically generating dynamic GitHub Actions workflows (`.github/workflows/security.yml`). Furthermore, it must feature a "Headless Mode," allowing the exact same Python codebase to run natively inside an Ubuntu Docker container on GitHub servers using an `xvfb` virtual framebuffer, completely independent of the local KiCad graphical environment.

Chapter 4

Mathematical Threat Modeling

4.1 The Coordinate System and Internal Units (IU)

Before any mathematical threat modeling can occur, the plugin must normalize its understanding of physical space within the KiCad environment.

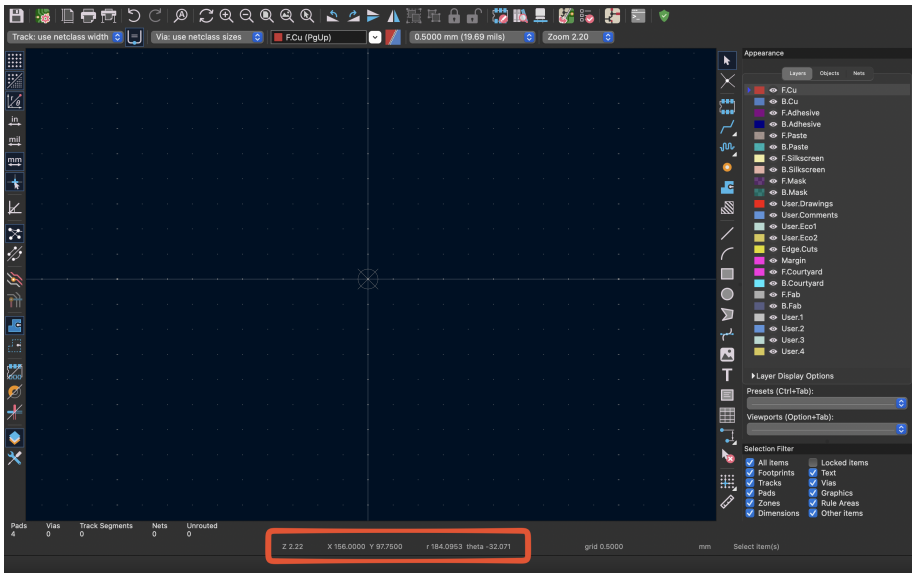

Figure 4.1: The KiCad PCB Editor canvas highlighting the inverted Y-axis coordinate system origin.

Unlike standard Cartesian coordinate systems where the origin $(0, 0)$ is situated at the bottom-left corner and the Y-axis increases upwards, the KiCad canvas operates on an inverted Y-axis system. The origin $(0, 0)$ is located at the top-left corner of the infinite canvas. As you move to the right, the X-axis values increase positively. As you move down the canvas, the Y-axis values increase positively. Any mathematical vector logic applied to the board must account for this inversion to avoid calculating negative, absolute distances incorrectly.

Furthermore, KiCad stores all physical dimensions in its memory using Internal Units (IU). In modern KiCad versions, 1 IU is equal to exactly 1 nanome-

ter (10^{-9} meters) to prevent floating-point rounding errors during extremely dense IC routing. Because security tolerances (such as a 5.0mm firewall rule) are defined in human-readable millimeters, the plugin actively utilizes the `pcbnew.ToMM()` and `pcbnew.FromMM()` scaling functions to bridge the gap between nanometer-scale memory and macroscopic security thresholds.

4.2 Vector Geometry and Bounding Box Extraction

The core function of the linter relies on establishing the physical boundary of the device. A PCB is rarely a perfect square placed exactly at the $(0, 0)$ origin; it is typically drawn freely somewhere on the canvas.

The algorithm extracts the absolute boundary of the PCB by iterating through all graphical items explicitly drawn on the `Edge.Cuts` layer.

Listing 4.1: Extracting the Edge.Cuts Bounding Box

```
board_edges = {"left": float('inf'), "right": float('-inf'),
               "top": float('inf'), "bottom": float('-inf')}

for drawing in board.GetDrawings():
    if drawing.GetLayerName() == "Edge.Cuts":
        bbox = drawing.GetBoundingBox()
        # Evaluate global extremes across all line segments
        board_edges["left"] = min(board_edges["left"], bbox.
           GetLeft())
        board_edges["right"] = max(board_edges["right"], bbox
           .GetRight())
        board_edges["top"] = min(board_edges["top"], bbox.
           GetTop())
        board_edges["bottom"] = max(board_edges["bottom"],
           bbox.GetBottom())
```

By parsing the top, bottom, left, and right coordinates of every line segment, arc, and polygon on the `Edge.Cuts` layer, the algorithm constructs a dynamic, absolute bounding box. This mathematically represents the physical perimeter of the board, regardless of where the designer arbitrarily placed it on the KiCad canvas.

4.3 The Pythagorean Distance Algorithm

Once a sensitive component (such as a UART RX pad) is identified via Regex, the system must calculate its spatial vulnerability. The plugin calls the `pad.GetPosition()` method, which returns the exact (X, Y) center of the copper pad in Internal Units.

To evaluate the vulnerability, the algorithm projects four orthogonal vectors from the center of the target component's pad to the nearest horizontal and vertical board boundaries.

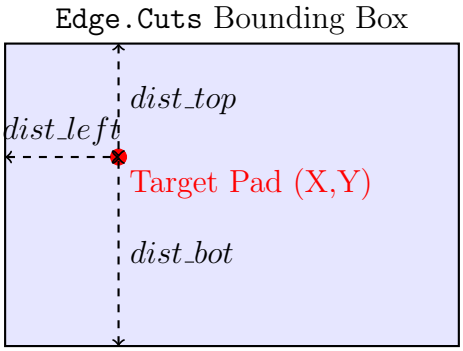

Figure 4.2: Geometric Projection for Euclidean Distance Calculation

Using the standard Euclidean distance formula derived from the Pythagorean theorem:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.1}$$

Because the vectors are cast orthogonally (straight up, down, left, and right), one of the delta coordinates is always zero, simplifying the calculation to a direct absolute difference. The algorithm calculates the distance to all four edges and takes the minimum value:

Listing 4.2: Calculating Minimum Distance to Edges

```
pad_pos = pad.GetPosition()
pad_x = pad_pos.x
pad_y = pad_pos.y

# Calculate orthogonal distances to the 4 boundaries
dist_left = pad_x - board_edges["left"]
dist_right = board_edges["right"] - pad_x
dist_top = pad_y - board_edges["top"]
dist_bottom = board_edges["bottom"] - pad_y

# The vulnerability is determined by the closest edge
min_dist_iu = min(dist_left, dist_right, dist_top,
   dist_bottom)
min_dist_mm = pcbnew.ToMM(min_dist_iu)
```

If `min_dist_mm` is less than the specified threshold, the component is mathematically proven to be a security risk.

4.4 Clearance Tolerances and Threat Zones

Security tolerances are defined mathematically in millimeters. By default, the system establishes two strict zones based on the physics of probing and EMFI attacks:

- **Critical Risk Zone (0.0mm - 5.0mm)**: Any sensitive node located within 5.0mm of the edge. Standard plastic enclosures (like ABS or Polycarbonate) are typically 2.0mm to 3.0mm thick. A component placed 5.0mm from the

board edge is practically pressing against the inside of the plastic casing. This distance is considered trivial for an attacker to access by slightly modifying the enclosure, inserting a needle, or using a very low-power EMFI probe. Any violation in this zone triggers the Pre-Push Firewall.

- **Warning Zone (5.1mm - 15.0mm)**: Any sensitive node located within 15.0mm of the edge. While harder to access physically with pogo pins, high-powered EMFI probes with larger magnetic coil diameters may still inject transient faults at this range if no internal copper shielding or ground planes are present. Violations in this zone generate local GUI warnings, encouraging the engineer to move the component further toward the protected center of the board.

Chapter 5

Core Plugin Architecture

5.1 System Architecture Overview

The eSim Hardware Security Linter was architected to be a lightweight, highly modular Action Plugin within the KiCad ecosystem. Unlike traditional standalone python scripts that require the user to manually export GERBER files, IPC-D-356 netlists, or intermediate XML files, this plugin interacts dynamically with KiCad's internal C++ memory structures via native Python bindings.

This approach ensures zero friction for the engineer. The architecture consists of three primary, decoupled layers:

1. **The Presentation Layer**: Developed using the `wxPython` library, this layer provides the interactive `SettingsDialog` and the DevSecOps pipeline triggers directly inside the KiCad GUI.

2. **The Logic Layer**: The Python mathematical backend that invokes the `pcbnew` API to extract physical layout geometries, parse electrical nets, and draw vulnerability markers on the canvas.

3. **The Automation Layer**: The subsystem responsible for intercepting Git commits, dynamically generating YAML CI/CD configuration files, and spawning parallel `subprocess` calls to the cloud.

5.2 Action Plugin Registration

To seamlessly integrate into the KiCad environment, the linter must be structured as a formal KiCad Action Plugin. The core class, `ESimHardwareLinter`, inherits directly from `pcbnew.ActionPlugin`. By overriding the `defaults()` method, the plugin registers its name, description, and the specific icon that will appear in the KiCad toolbar.

The true entry point of the software is the overridden `Run()` method. When the engineer clicks the FOSSEE icon in the toolbar, KiCad natively instantiates the plugin and calls `Run()`, passing the execution thread context directly to our Python backend.

5.3 Local Execution Flowchart

The following architecture diagram illustrates the localized execution flow of the mathematical engine, detailing the steps from invocation to physical marker generation.

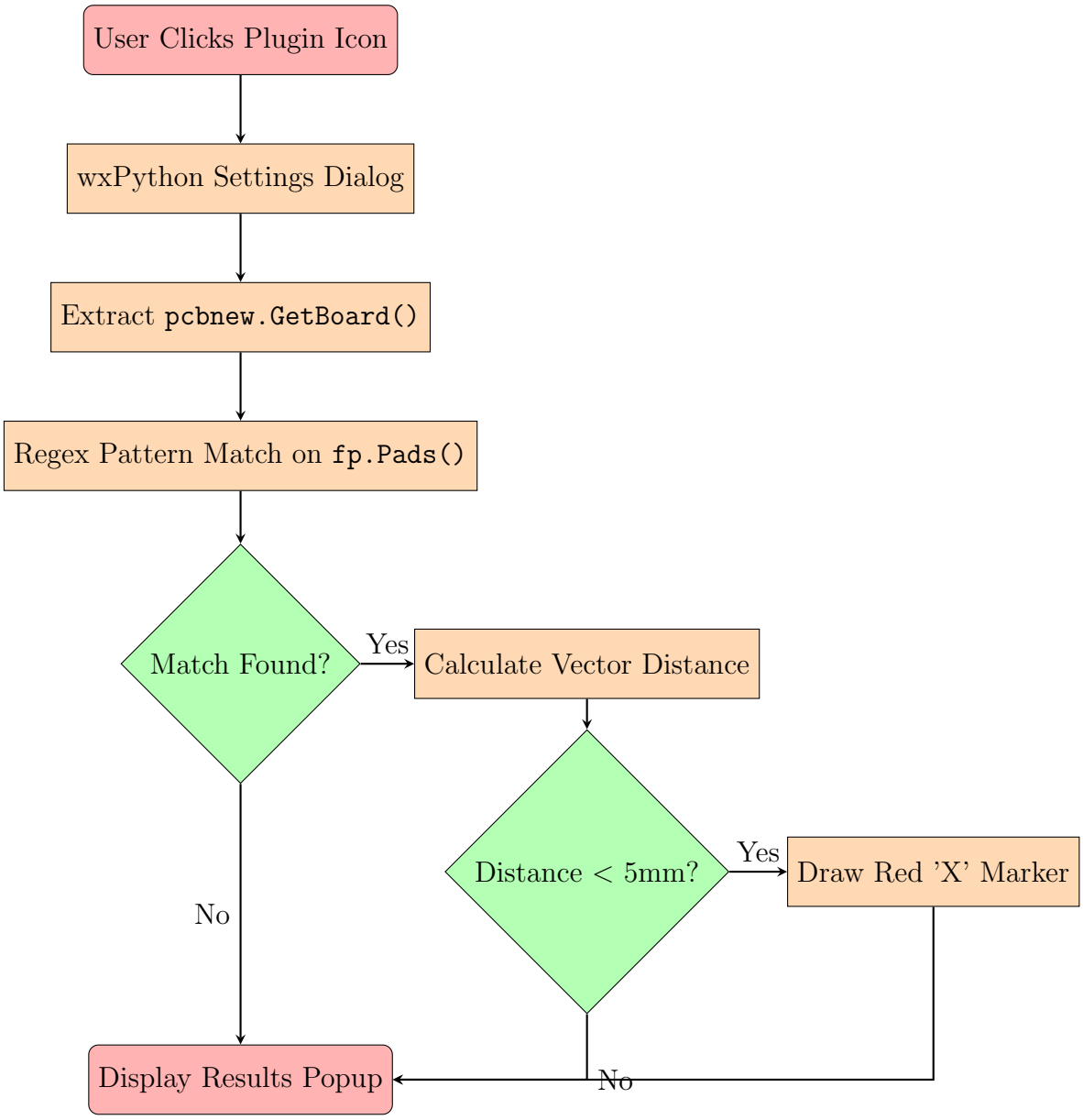

Figure 5.1: System Architecture: Local Mathematical Engine Flow

5.4 wxPython GUI Integration Detailed Breakdown

Because KiCad itself is built using the wxWidgets C++ framework, the official GUI toolkit embedded within KiCad's Python interpreter is `wxPython`. Upon launching

the plugin, the ‘Run()‘ method instantiates a custom `wx.Dialog` window.

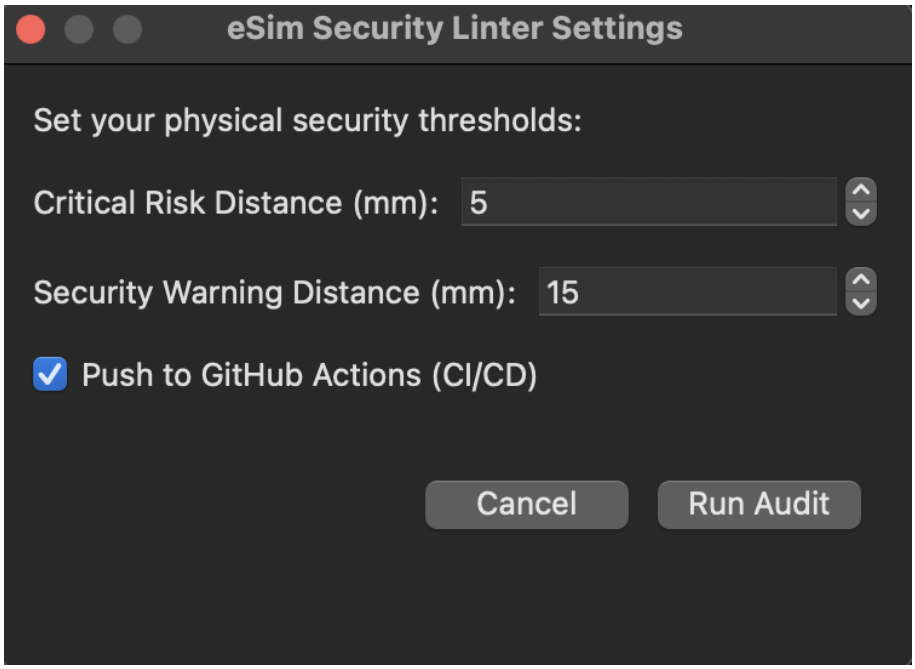

Figure 5.2: The eSim Hardware Security Linter GUI interface in KiCad.

The `SettingsDialog` class constructs a highly structured graphical interface. It utilizes a vertical box sizer (`wx.BoxSizer(wx.VERTICAL)`) to cleanly align the UI elements.

To ensure mathematical precision and prevent engineers from accidentally inputting text strings or invalid characters into the threshold fields, the GUI employs `wx.SpinCtrlDouble` widgets instead of standard text boxes. These spin controls explicitly restrict user input to floating-point numbers between 0.0mm and 100.0mm.

Crucially, the GUI incorporates a simple `wx.CheckBox` labeled ”Push to GitHub Actions”. This seemingly simple UI element acts as the primary gateway for the DevSecOps 1-Click Deployment Engine. By intercepting the state of this checkbox via `self.checkbox_github.GetValue()`, the `Run` method determines whether to execute the cloud deployment protocols or simply finish the local mathematical audit.

5.5 The pcbnew Python API and Marker Generation

The plugin interfaces directly with KiCad’s internal memory structures. `pcbnew.GetBoard()` returns a pointer to the `BOARD` object currently loaded in the user’s active session. This is vastly superior to traditional file-parsing linters because it ensures the tool is always analyzing the live, unsaved state of the user’s canvas.

When the mathematical engine determines that a component violates the security threshold, it provides immediate visual feedback. It does this by instantiating

a new pcbnew.PCB_SHAPE object. The plugin sets the shape type to a line segment (pcbnew.SHAPE_T_SEGMENT) and explicitly assigns it to the pcbnew.Dwgs_User (User Drawings) layer. By drawing two intersecting 2mm line segments exactly over the component's (X,Y) coordinates, the plugin generates a high-visibility Red 'X' marker directly on the engineer's active canvas.

5.6 JSON Organizational Security Profiles

To enforce enterprise-level consistency across multiple engineering teams, the plugin implements a hierarchical configuration override system. It searches the project root directory for a specific file named esim_security.json.

Listing 5.1: Parsing JSON Security Profiles

```
def load_organization_config(board_path):
    if not board_path: return None
    dir_path = os.path.dirname(board_path)
    config_path = os.path.join(dir_path, "esim_security.json"
       )
    if os.path.exists(config_path):
        try:
            with open(config_path, "r") as f:
                return json.load(f)
        except Exception:
            return None
    return None
```

If found, the plugin parses this JSON file using the standard json module and silently overrides the local GUI parameters. This guarantees that the organization's standardized security thresholds (e.g., locking the minimum critical distance to 5.0mm) are strictly enforced across all engineers. It effectively prevents a rogue or junior engineer from simply lowering the threshold slider in their local GUI to blindly bypass a security failure.

Chapter 6

Automated Netlist Scanning

6.1 The Failure of Generic Footprint Analysis

A major challenge in automating the security audit of PCB layouts is the widespread use of generic footprints in schematic capture. In standard eSim and KiCad workflows, when an engineer decides to expose a hardware debugging interface (such as a UART port or an I2C bus), they rarely use a dedicated, uniquely identifiable component. Instead, they typically place a standard 4-pin or 6-pin male header and assign it a generic reference designator (e.g., `J1`, `P1`, or `Conn_01x04`).

Traditional automated layout linters—which rely heavily on parsing component metadata, footprint library designations, or simple Bill of Materials (BOM) values—will completely fail to identify this component as a high-risk security threat. To a standard linter, it is just a harmless generic connector used for mounting a fan or an LED. Identifying the true purpose of the component requires deep contextual awareness of the underlying electrical circuit.

6.2 Deep Copper Inspection Architecture

To circumvent the limitations of generic metadata, the eSim Hardware Security Linter employs a novel architecture known as "Deep Copper Inspection." Rather than analyzing the top-level schematic symbol, the algorithm physically dives into the copper layer.

The execution flow begins by iterating through every single physical footprint currently placed on the KiCad canvas using the `board.GetFootprints()` API call. However, instead of stopping at the footprint level, the algorithm recursively drops a layer deeper. It actively iterates through the internal structure of the footprint, identifying every single individual copper pad using `fp.Pads()`.

Because KiCad rigidly binds schematic nets to physical copper pads to enforce its Electrical Rule Checks (ERC), the plugin leverages this binding by calling `pad.GetNetname()`. This API extracts the official, human-readable electrical net string that the engineer assigned to that specific trace (e.g., `/MCU_TX` or `Net-(U1-Pad4)`).

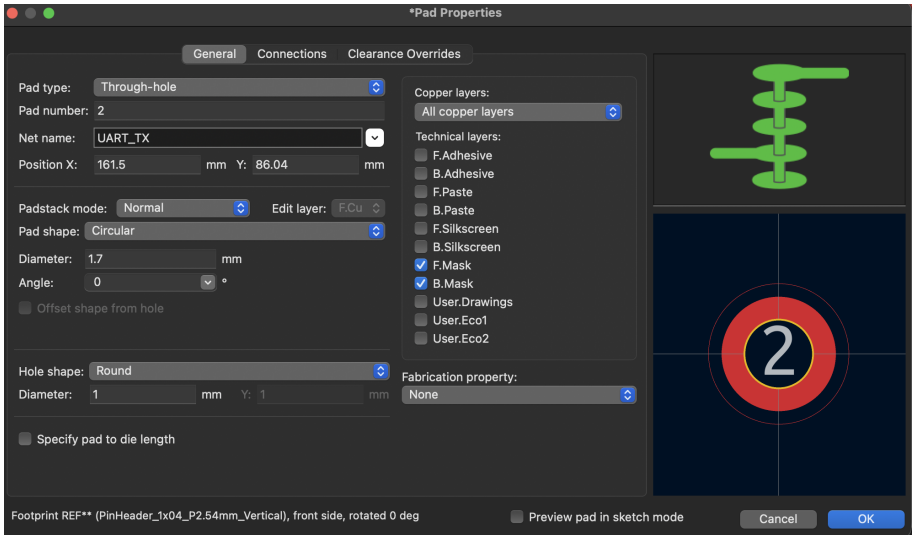

Figure 6.1: The KiCad Pad Properties dialog showing the exact Net Name bound to a physical copper pad.

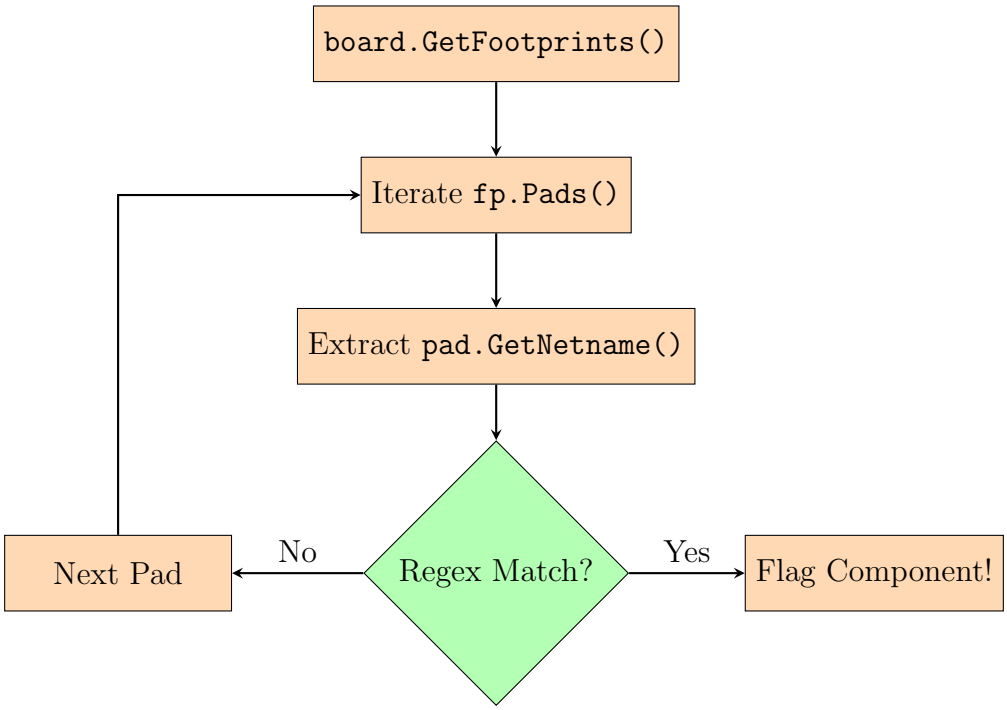

Figure 6.2: Deep Copper Netlist Scanning Execution Loop

6.3 Exhaustive Regular Expression (Regex) Parsing

Once the raw netlist strings are extracted directly from the copper pads, they are evaluated against an exhaustive dictionary of known sensitive interfaces. To accomplish this with maximum efficiency and flexibility, the plugin utilizes Python's built-in `re` (Regular Expression) module.

Listing 6.1: Exhaustive Regex Threat Detection Engine

```
# Compile the regex pattern once globally for execution
   performance
DEBUG_NET_PATTERN = re.compile(
    r'(SWD|JTAG|UART|RX|TX|MISO|MOSI|SCL|SDA|TCK|TMS|TDI|TDO)
       ',
    re.IGNORECASE
)

# Traverse all footprints on the canvas
for fp in board.GetFootprints():
    fp_vulnerable = False

    # Traverse all physical pads inside the current footprint
    for pad in fp.Pads():
        net_name = pad.GetNetname()

        # Cross-reference the net name against the Regex
           dictionary
        if DEBUG_NET_PATTERN.search(net_name):
            fp_vulnerable = True
            break # No need to check other pads if the
               component is compromised

    # Escalation to the Pythagorean distance engine
    if fp_vulnerable:
        evaluate_distance_to_edge(fp)
```

6.3.1 Threat Signatures and Protocol Mapping

The compiled regex string acts as a comprehensive signature database for hardware vulnerability protocols:

- **JTAG (Joint Test Action Group)**: Flags `JTAG`, `TCK` (Clock), `TMS` (Mode Select), `TDI` (Data In), and `TDO` (Data Out). This is the most critical interface to detect, as it allows attackers to halt the CPU and directly dump SRAM and flash memory.
- **SWD (Serial Wire Debug)**: Flags `SWD` and `SWDIO`. The modern, two-wire ARM equivalent of JTAG.
- **UART (Universal Asynchronous Receiver-Transmitter)**: Flags `UART`, `TX` (Transmit), and `RX` (Receive). Commonly abused by attackers to access root UNIX shells or view plaintext system boot logs.
- **SPI / I2C (Serial Peripheral Interface / Inter-Integrated Circuit)**: Flags `MISO`, `MOSI`, `SCL`, and `SDA`. Attackers tap these lines to intercept unen-

crypted cryptographic keys passing between the microcontroller and external EEPROM chips.

6.3.2 Algorithmic Optimization

To optimize execution speed—a critical requirement for a CI/CD DevSecOps pipeline—the regex pattern is explicitly compiled *once* at the top of the script using `re.compile()` rather than evaluating a raw string on every loop iteration.

Furthermore, the implementation utilizes `re.IGNORECASE`. Because naming conventions vary wildly between engineers (e.g., `MCU_Tx`, `mcu_tx_debug`, or `TXD`), the case-insensitive regex search guarantees that a vulnerability is flagged regardless of the engineer's typographical choices. If a match is found on even a single pad, the `break` statement instantly halts the pad iteration loop, flags the entire generic footprint as a compromised security risk, and forwards the component to the mathematical engine for spatial evaluation.

Chapter 7

The Pre-Push Firewall

7.1 The Concept of the DevSecOps Firewall

The ultimate goal of the eSim Hardware Security Linter is not just to politely warn engineers, but to structurally prevent vulnerable hardware designs from ever reaching production. In software engineering, this is accomplished using Git Hooks (specifically, the `pre-push` hook), which trigger automated security scans before code is uploaded.

However, traditional Git hooks suffer from a massive flaw: they are strictly client-side suggestions. A developer in a rush can easily bypass them by simply typing `git push --no-verify` in their terminal.

To accomplish true "Secure-by-Design" enforcement, the plugin implements a novel "Pre-Push Firewall" directly inside the KiCad GUI. This mechanism acts as an intelligent proxy, completely intercepting the standard Git workflow. When an engineer attempts to deploy their layout to GitHub using the plugin's UI, the plugin refuses to spawn the `git push` subprocess until it forces the local mathematical audit to run and pass *first*. Because the Git command itself is hidden behind the Python execution layer, it is impossible for the engineer to use the `--no-verify` flag.

7.2 Pre-Push Execution Flowchart

The following flow diagram illustrates how the firewall intercepts the cloud deployment process, enforcing strict security compliance before granting network access.

7.3 Blocking Vulnerable Deployments

If the mathematical engine detects a "Critical Risk" (e.g., a JTAG port less than 5.0mm from the edge), the deployment sequence is instantly and violently aborted. The Boolean logic within the `Run` method strictly enforces that the `push_to_github()` function is mathematically unreachable if the `passed` flag evaluates to `False`.

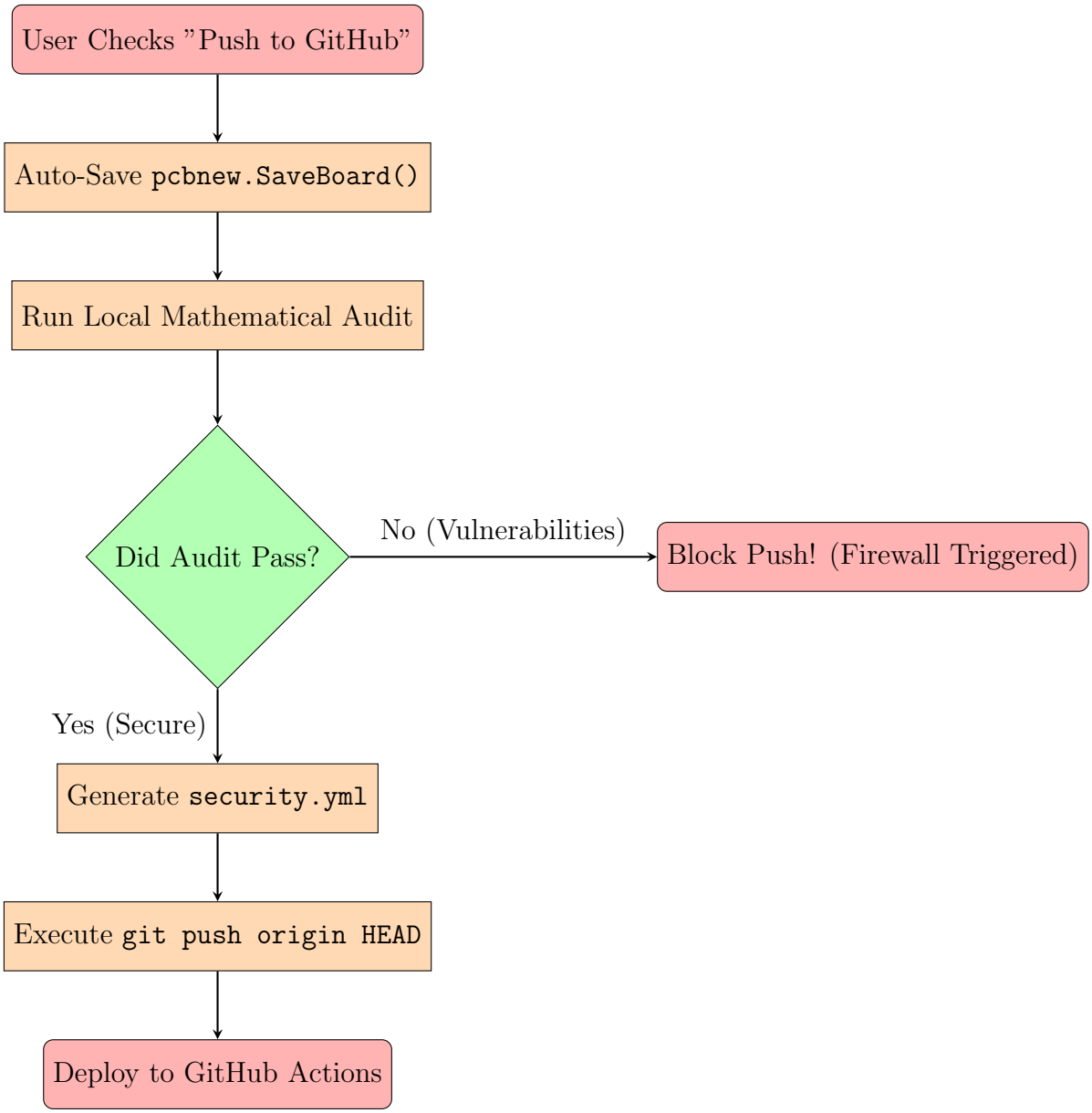

Figure 7.1: DevSecOps Pre-Push Firewall Execution Flow

Listing 7.1: Pre-Push Firewall Logic in linter.py

```
# Run local audit first as an inescapable Pre-Push Hook
passed = audit_board(board, crit_threshold, warn_threshold,
    headless=False)

if push_github:
    if passed:
        push_to_github(board_path, crit_threshold,
            warn_threshold)
    else:
        wx.MessageBox("GitHub Deployment Blocked!\n\nThe eSim
            Linter detected hardware vulnerabilities. You
            must fix these issues before the code can be
            pushed to the cloud.", "Pre-Push Hook Triggered",
            wx.OK | wx.ICON_ERROR)
```

By utilizing `wx.MessageBox(..., wx.ICON_ERROR)`, the Python script halts the main execution thread of the application and throws a high-visibility, modal error box on the screen. The layout code literally cannot leave the local machine until the hardware vulnerabilities are physically corrected and moved inward by the designer.

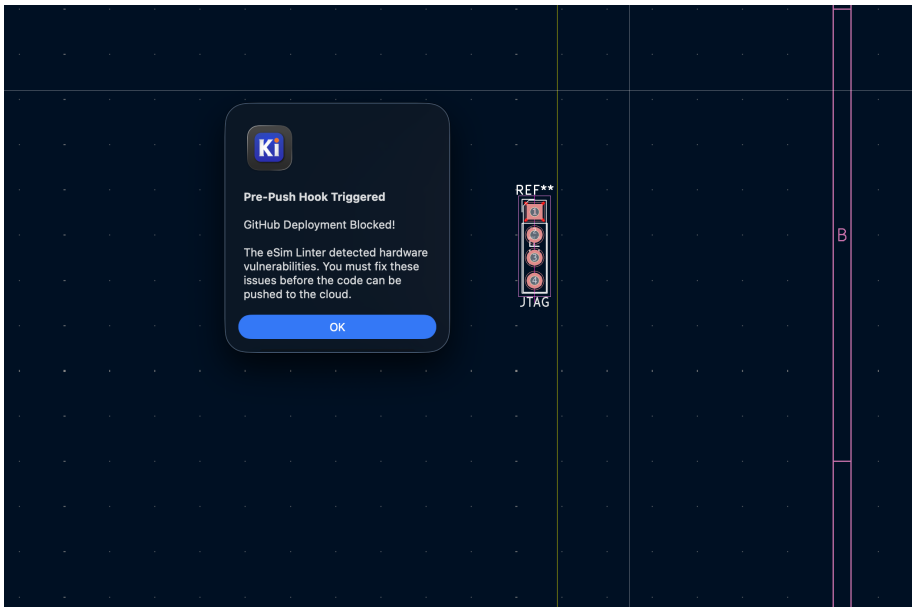

Figure 7.2: The modal wx.MessageBox physically preventing the local `git push` execution.

7.4 Auto-Save Integrity Protocol

A critical logical flaw was identified early during the testing phase of this project, known as a "State Mismatch" bug. In KiCad, when an engineer edits a trace or moves a component, those changes exist purely in volatile Random Access Memory (RAM). They are not written to the actual `.kicad_pcb` file on the hard drive until the user explicitly saves.

This created a severe security loophole: an engineer might fix a vulnerability on their screen (in RAM), but forget to press `CTRL+S` before clicking the GitHub deployment button. The local Python audit would pass (because it evaluates the secure RAM state using `pcbnew.GetBoard()`), but the subsequent `git commit` subprocess would package and upload the older, vulnerable file resting on the hard drive.

To solve this fatal mismatch, an "Auto-Save Integrity Protocol" was deeply integrated into the plugin's execution flow. The Python code explicitly invokes `pcbnew.SaveBoard(board_path, board)` at the very beginning of its execution sequence. This seamlessly and silently flushes the current, secure RAM state to the hard drive before any Git commands are initiated. This protocol guarantees absolute cryptographic parity between the local mathematical evaluation and the final cloud deployment artifact.

Chapter 8

Headless Cloud Infrastructure

8.1 The Dual-Mode Architecture

A central design challenge of this project was engineering a single Python codebase (`linter.py`) capable of operating natively within two fundamentally opposed environments: a highly interactive, graphical local desktop environment, and a completely sterile, non-graphical remote server environment.

To achieve this, the plugin utilizes a "Dual-Mode Architecture" managed via Python's `argparse` module. When the script is executed by a cloud server, it is invoked with a specific `--headless` CLI argument. When the execution engine detects this flag, it completely detaches itself from the `wxPython` library. It bypasses all GUI initialization routines, suppresses all `wx.MessageBox` popup alerts, and instead formats its security warnings as standard output strings (`sys.stdout`) that can be captured by the cloud server's logging facility.

Most importantly, if the headless mathematical engine detects a component inside the Critical Risk Zone, it executes a `sys.exit(1)` command. Continuous Integration (CI/CD) runners (like GitHub Actions and GitLab CI) monitor the exit codes of their child processes. By throwing a non-zero exit code, the plugin forces the entire pipeline to fail, blocking the vulnerable layout from progressing to the fabrication pipeline.

8.2 The X11 Display Server Problem and xvfb

Deploying the KiCad Python API to a cloud server introduces a severe technical hurdle. The `pcbnew` C++ library is inextricably linked with the `wxWidgets` graphical framework. Even if a Python script is only attempting to perform pure mathematical distance calculations without rendering a GUI, the underlying KiCad bindings still attempt to initialize an X11 display context upon execution.

Because GitHub Actions Ubuntu Docker containers are headless servers—they lack physical monitors and graphical display managers—running `python3 linter.py` directly results in an immediate, fatal X11 Display Assertion crash.

To solve this, the pipeline generation engine relies on `xvfb` (X Virtual Framebuffer). `xvfb` is an in-memory display server that performs all graphical operations in virtual memory without requiring any screen output. By wrapping the python

command inside `xvfb-run`, the server is tricked into believing a physical monitor exists. The KiCad API initializes successfully, the math is calculated in virtual memory, and the pipeline proceeds without crashing.

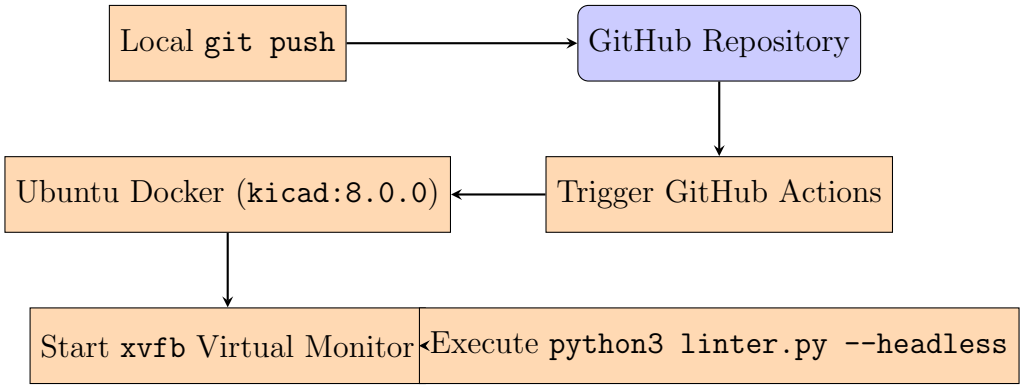

Figure 8.1: Headless Cloud Infrastructure Execution Pipeline

8.3 Dynamic CI/CD Workflow Generation

The DevSecOps pipeline is entirely "1-Click." The engineer does not need to know how to write YAML configuration files or understand Docker networking. The plugin automatically generates the infrastructure.

When the "Push to GitHub" sequence is initiated, the Python script dynamically authors a completely functional GitHub Actions workflow file. It creates the `.github/workflows/security.yml` directory structure, injects the exact filename of the active PCB, and hardcodes the specific Critical and Warning thresholds that the engineer configured in the local GUI.

Listing 8.1: Generated GitHub Actions YAML Output

```
jobs:
  security-audit:
    runs-on: ubuntu-latest
    container:
      image: kicad/kicad:8.0.0
    steps:
      - name: Install Headless Display Server
        run: |
          apt-get update
          apt-get install -y xvfb

      - name: Run eSim Hardware Security Linter
        run: xvfb-run python3 linter.py --board "demo.
           kicad_pcb" --crit 5.0 --warn 15.0
```

The script utilizes the official `kicad/kicad:8.0.0` Docker image, which provides a clean, standardized environment equipped with all necessary KiCad python

dependencies.

```yaml
name: eSim Security Audit

on: [push, pull_request]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    container:
      image: kicad/kicad:9.0
      options: --user root

    steps:
      - name: Checkout Code
        uses: actions/checkout@v5

      - name: Install Headless Display Server
        run: |
          apt-get update
          apt-get install -y xvfb

      - name: Run eSim Hardware Security Linter
        run: |
          # Use xvfb-run to simulate a virtual monitor for KiCad/wxPython
          xvfb-run python3 linter.py --board "demo1.kicad_pcb" --crit 5.0 --warn 15.0
```

Figure 8.2: The automatically generated security.yml file residing in the GitHub repository.

8.4 Cloud-to-Local UX Bridge

The final element of the DevSecOps architecture is the User Experience (UX) bridge. When an engineer clicks "Run Audit" and the code is successfully pushed to the repository, the engineer wants immediate confirmation that the cloud server is processing their layout.

To achieve this, the plugin utilizes the `webbrowser` module to automatically open the engineer's default internet browser directly to their project's GitHub Actions URL.

However, a race condition exists: if the browser opens immediately after the `git push` command finishes, it takes approximately 1 to 2 seconds for GitHub's backend servers to register the new commit and instantiate the Ubuntu runner. The engineer would see a blank page. To prevent this, a 2500 millisecond delay was required.

Standard Python delays (e.g., `time.sleep(2.5)`) are completely unacceptable in a GUI application because they block the main execution thread, causing the entire KiCad interface to freeze and become unresponsive (resulting in a "Application Not Responding" operating system error). Instead, the plugin implements a non-blocking asynchronous timer using `wx.CallLater(2500, self.open_github_logs)`. This instructs the GUI event loop to wait exactly 2.5 seconds in the background while keeping the KiCad interface perfectly fluid and responsive, finally popping open the Chrome browser right as the cloud server spins up.

Chapter 9

Case Studies and System Evaluation

To rigorously evaluate the efficacy, reliability, and graceful degradation of the eSim Hardware Security Linter, the plugin was subjected to two distinct, real-world engineering scenarios. These case studies demonstrate the system's ability to handle both offline prototyping environments and strict, enterprise-level cloud deployment pipelines.

9.1 Case Study 1: Local Offline Audit and Graceful Degradation

9.1.1 Scenario Setup

In the first evaluation scenario, the goal was to test the linter's core mathematical engine and its ability to handle unconfigured environments gracefully. A hypothetical junior engineer was tasked with designing a prototype IoT Smart Water Meter. The engineer started a brand new KiCad project locally. Crucially, this project was strictly a local prototype and had *not* been initialized as a Git repository (no `.git` folder existed).

During the layout phase, the engineer placed a standard 4-pin generic header (`Conn_01x04`) to act as a debug port. They routed the `MCU_TX` and `MCU_RX` nets to this header and placed it a mere 2.0mm from the top `Edge.Cuts` boundary to make it easier to attach an FTDI serial cable.

9.1.2 Execution and Graceful Failure

1. The engineer launches the plugin via the KiCad toolbar. Unaware of the pipeline requirements, the engineer checks the "Push to GitHub" box in the GUI and initiates the audit.

2. The plugin's automation layer attempts to resolve the remote repository by executing `subprocess.run(["git", "remote", "get-url", "origin"])`.

3. Because the project is not a Git repository, the command fails and throws a `CalledProcessError`. Instead of allowing this Python exception to violently crash the KiCad application, the plugin gracefully catches the exception.

4. It halts the cloud deployment thread and displays a localized `wx.MessageBox` warning the user that no Git environment was detected, but it intelligentally allows the local mathematical audit to proceed seamlessly.

9.1.3 Mathematical Evaluation and Visual Output

1. The Deep Copper Inspection engine traverses the `Conn_01x04` footprint and extracts the `MCU_TX` net string. The compiled Regex dictionary immediately flags this as a highly sensitive UART interface.

2. The Pythagorean mathematical engine extracts the `Edge.Cuts` bounding box and calculates orthogonal vectors from the center of the UART pad. It determines the minimum Euclidean distance to be exactly 2.0mm.

3. This distance severely violates the 5.0mm Critical Risk threshold enforced by the organization's `esim_security.json` profile.

4. Using the `pcbnew.PCB_SHAPE` C++ bindings, the plugin instantly draws mathematically precise 2mm red intersecting line segments directly over the UART header's `(X,Y)` coordinates on the `User.Drawings` layer.

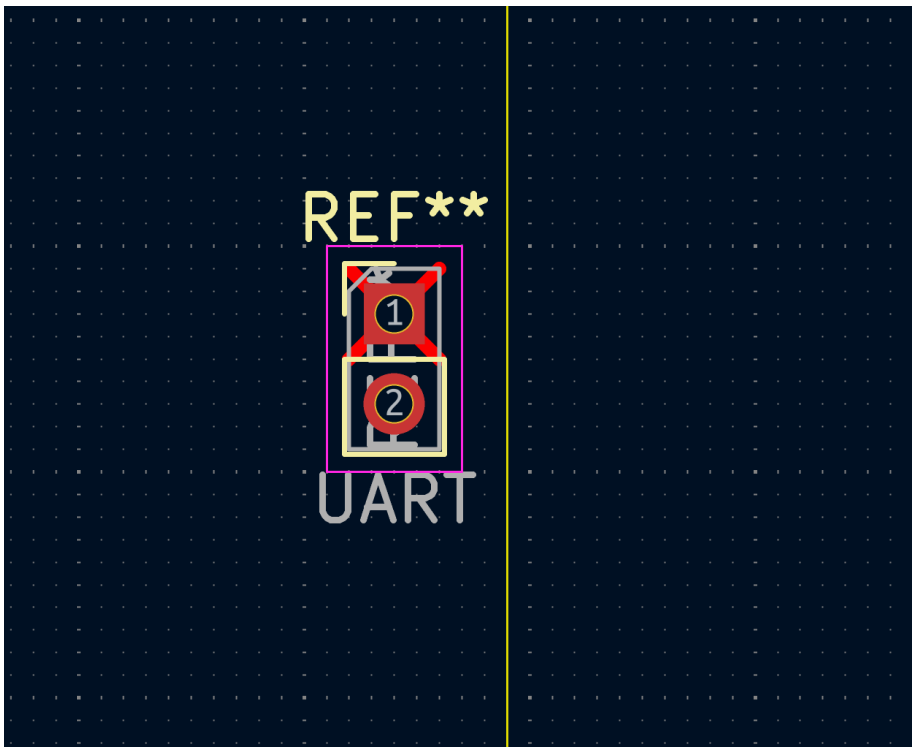

Figure 9.1: Visual identification of a physical hardware vulnerability. Red "X" markers dynamically rendered by the Python API on the User.Drawings layer.

The GUI displays a detailed localized failure report, successfully forcing the engineer to recognize and resolve the physical vulnerability offline without crashing the host software.

9.2 Case Study 2: Enterprise Cloud Deployment and the Pre-Push Firewall

9.2.1 Scenario Setup

In the second evaluation scenario, the goal was to test the structural DevSecOps enforcement mechanisms. A senior engineer is finalizing the layout for an automotive Electronic Control Unit (ECU). The KiCad project is officially initialized and connected to a remote FOSSEE GitHub organization repository.

During routing, the engineer places a high-density JTAG debugging header (containing the sensitive nets `TCK`, `TMS`, `TDI`, `TDO`). Due to dense routing constraints in the center of the board, the engineer places the JTAG header 3.5mm from the physical edge of the PCB.

9.2.2 Firewall Interception

1. Convinced the board is ready for production, the engineer opens the plugin, checks the "Push to GitHub" box, and clicks "Run".

2. The Pre-Push Firewall intercepts the execution thread. Before any Git subprocesses are spawned, the Pythagorean math engine evaluates the JTAG header and calculates the 3.5mm distance.

3. Because 3.5mm is within the 5.0mm Critical Risk Zone, the Firewall logic triggers. It actively aborts the `git push` deployment sequence and throws a modal "Deployment Blocked!" error on the screen. The vulnerable file is physically prevented from leaving the local machine.

9.2.3 Remediation, Auto-Save, and Cloud Execution

1. Acknowledging the error, the engineer manually moves the JTAG header inward to a safe distance of 12.0mm and surrounds it with a protective copper ground polygon.

2. The engineer clicks the "Run" button again.

3. The Auto-Save Integrity Protocol fires first. The plugin silently invokes `pcbnew.SaveBoard()`, seamlessly writing the secure 12.0mm RAM state to the hard drive to prevent a State Mismatch bug.

4. The mathematical engine confirms the board is now physically secure (12.0mm $>$ 5.0mm). The Pre-Push Firewall drops, granting the Python script access to the internet.

5. The automated deployment engine dynamically authors the `.github/workflows/security.yml` file, stages the layout, commits it, and pushes the code to GitHub in the background.

6. The non-blocking `wx.CallLater` UX bridge executes. It waits exactly 2500 milliseconds—allowing the GitHub backend servers to register the new commit and spin up the Ubuntu Docker container—before utilizing the `webbrowser` module to automatically pop open Google Chrome directly to the live GitHub Actions CI/CD pipeline logs.

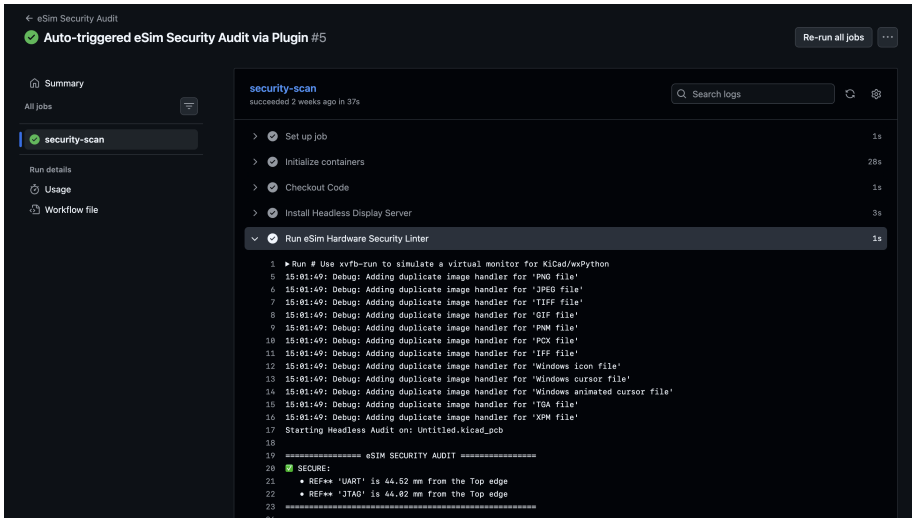

Figure 9.2: The Headless Cloud Infrastructure executing the eSim Hardware Security Linter natively inside a GitHub Actions Ubuntu container via xvfb-run.

The live console proves that the `xvfb` (X Virtual Framebuffer) successfully initialized the KiCad Python environment in a headless server, executed the mathematical audit in the cloud, and passed the pipeline, achieving a complete, end-to-end "Secure-by-Design" DevSecOps workflow.

Chapter 10

Limitations, Future Scope, and Conclusion

10.1 Current Limitations and Constraints

While the eSim Hardware Security Linter successfully establishes a new baseline for automated hardware security, the current mathematical models and algorithmic architectures possess specific limitations that must be acknowledged.

10.1.1 Complex Non-Manifold Board Geometries

The current vector projection mathematics assume relatively standard board geometries (i.e., rectangular or mildly contoured polygons). By calculating orthogonal minimums against a global bounding box, the algorithm performs exceptionally well for 95% of commercial IoT devices. However, highly complex, abstract board shapes with extreme internal cutouts or non-manifold concave edges may produce inaccurate edge distance calculations.

For instance, if a PCB is shaped like a "U," the bounding box logic might interpret the empty space in the center as "safe," whereas it is physically exposed to the outside environment. Solving this constraint requires moving beyond simple Cartesian bounding boxes and implementing advanced Ray-Casting algorithms or Polygon Winding Number algorithms capable of evaluating complex, concave geometric topologies.

10.1.2 Algorithmic Time Complexity on Enterprise Layouts

The "Deep Copper Inspection" algorithm guarantees extreme accuracy by aggressively traversing every single physical pad on every footprint. However, this introduces an $O(N \times P)$ time complexity constraint, where N is the number of footprints and P is the number of pads per footprint.

While execution is nearly instantaneous (under 50 milliseconds) on standard 2-layer or 4-layer IoT boards, scaling this tool to incredibly dense, 16-layer enterprise ATX motherboards with tens of thousands of High-Density Interconnect (HDI) BGA components may introduce noticeable computational delays. Future iterations will

require multi-threading the Python backend using the `concurrent.futures` module or offloading the topological map to a compiled C++ subroutine.

10.2 Scope for Future Enhancement

10.2.1 AI-Assisted Autonomous Auto-Routing

The most compelling future scope for this plugin is the integration of predictive artificial intelligence. Currently, the linter is reactive—it detects a human error and violently blocks deployment. A future iteration could evolve the tool into a proactive AI co-pilot.

By training a Machine Learning Topography model on thousands of secure layout designs, the AI could evaluate the board's congestion and automatically suggest secure pathways. Instead of merely throwing a `wx.MessageBox` warning, the AI could interface directly with the `pcbnew` API to physically relocate the vulnerable JTAG header to a mathematically optimal safe zone in the center of the board, dynamically ripping up and re-routing the connected copper traces without violating differential pair impedances or signal length tuning constraints.

10.2.2 Vertical Z-Axis Attack Mitigation

The current mathematical engine evaluates vulnerability strictly on the X-Y plane (the 2D edge distance). However, highly funded attackers may bypass edge-probing entirely by decapping the plastic enclosure from the top or bottom and drilling straight down (a Z-axis attack).

Future versions of the plugin could evaluate the layer stackup. It could calculate if a sensitive UART trace is exposed on the `F.Cu` (Top Copper) layer, and automatically enforce a rule requiring the trace to be buried in an internal layer (e.g., `In1.Cu`) sandwiched between two solid `GND` (Ground) planes for electromagnetic shielding.

10.3 Conclusion

The traditional electronics design lifecycle has structurally disadvantaged hardware security by treating it as an afterthought relegated to manual design reviews. The eSim Hardware Security Linter successfully shatters this paradigm by bridging the critical gap between hardware engineering and enterprise DevSecOps.

By transforming a static, isolated KiCad environment into an aggressive, proactive security gateway, organizations can finally enforce strict physical layout standards programmatically. The integration of the "Deep Copper Inspection" engine guarantees that disguised or generically labeled debug ports are intercepted before they ever reach the fabrication facility, neutralizing the threat of physical probing and Electromagnetic Fault Injection (EMFI) attacks at the source.

The project achieved all of its core semester objectives, culminating in a fully functional, enterprise-grade V2.0 plugin. The "1-Click Deployment Engine," the "Pre-Push Firewall," and the "Headless Cloud Infrastructure" provide a foolproof

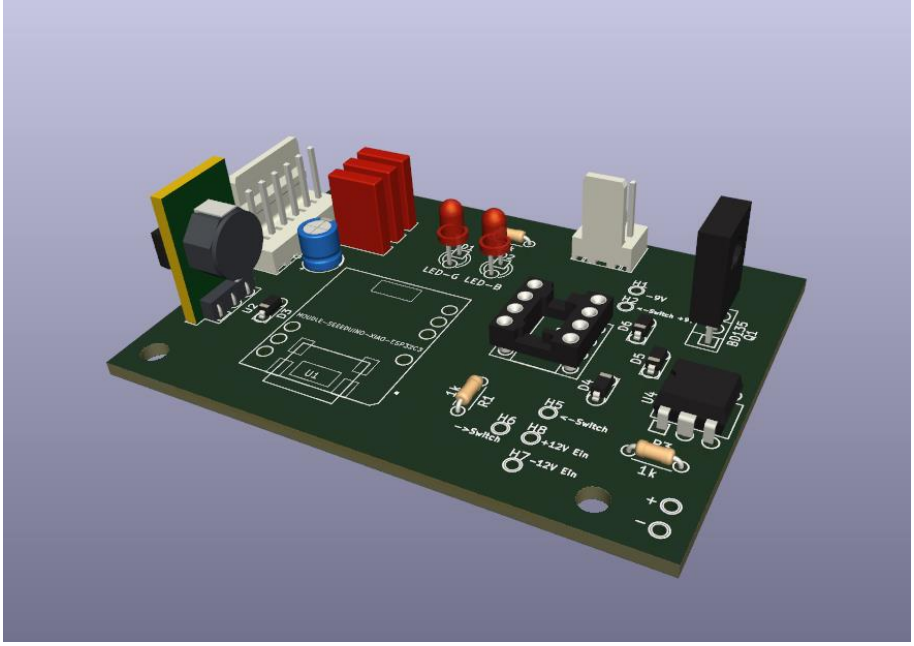

Figure 10.1: A 3D rendering of a fully secured PCB, highlighting buried traces and internal shielding.

security ecosystem that completely removes the element of human negligence from the deployment process. This advanced security pipeline has been officially submitted as a Pull Request to the main FOSSEE repository, contributing directly to the democratization of open-source hardware engineering and cementing the concept of "Secure-by-Design" hardware development.

Appendix A

Daily Work Log

A.1 Activity Log

The table below summarises the work completed during the FOSSEE Semester Long Internship, Spring 2026 (18 February 2026 - 14 May 2026) for the **eSim Hardware Security Linter** project.

Day	Date	Phase	Work Description
Wed	18/02/2026	Orientation	Attended orientation session for the FOSSEE Semester Long Internship Spring 2026.
Thu	19/02/2026	Task Exploration	Explored available internship tasks and studied the scope of KiCad plugin development and physical hardware security.
Fri	20/02/2026	Task Exploration	Continued exploring tasks; studied prior intern work and FOSSEE eSim documentation regarding layout tools.
Sat	21/02/2026	Task Exploration	Researched JTAG/UART vulnerability attack vectors and EMFI probing methods on IoT PCBs.
Mon	23/02/2026	Task Exploration	Research about KiCad `pcbnew` Python API capabilities: geometry parsing, netlist extraction, and layer rendering.
Tue	24/02/2026	Task Exploration	Explored the technicalities related to hooking KiCad's C++ memory structures using Python scripts.
Wed	25/02/2026	Google Meet	Attended Google Meet with mentor. Proposed the idea of an automated Hardware DevSecOps Linter.
Thu	26/02/2026	Research	Researched coordinate systems within KiCad (Internal Units, inverted Y-axis origin).

Day	Date	Phase	Work Description
Fri	27/02/2026	Proposal	Prepared a document for the idea pitch and plugin architectural workflow.
Sat	28/02/2026	Proposal	Researched the gap between software CI/CD and manual hardware design reviews.
Mon	02/03/2026	Plugin Development	Made a technology stack and mathematical workflow for the physical coordinate analysis engine.
Tue	03/03/2026	Plugin Development	Set up the KiCad Python environment. Created `linter.py` and `__init__.py` files and registered the `ActionPlugin`.
Thu	05/03/2026	Plugin Development	Developed the initial mathematical logic to parse the `Edge.Cuts` bounding box of a KiCad board.
Fri	06/03/2026	Plugin Development	Resolved bugs related to Internal Units (IU) nanometer-to-millimeter conversion logic.
Mon	09/03/2026	Plugin Development	Implemented the Deep Copper Inspection engine; wrote recursive loops to scan footprints and pads.
Thu	12/03/2026	Plugin Development	Debugging session: resolved issues with `pad.GetNetname()` not extracting the schematic strings correctly.
Fri	13/03/2026	Plugin Development	Implemented Python Regular Expressions (Regex) to compile a dictionary of sensitive strings (e.g., UART, JTAG).
Sat	14/03/2026	Google Meet + Plugin	Attended Google Meet. Demonstrated the successful extraction of Euclidean distances to the mentor.
Mon	16/03/2026	Plugin Development	Implemented the visual engine: drawing red "X" markers dynamically on the `User.Drawings` layer using `PCB_SHAPE`.
Wed	18/03/2026	Plugin Development	Finalised the core local audit mathematical engine and visual rendering system.
Fri	20/03/2026	Plugin Development	Started working on the `wxPython` Graphical User Interface (GUI) integration.
Sat	21/03/2026	Google Meet + Plugin	Attended Google Meet. Showcased the interactive GUI prototype.
Mon	23/03/2026	Plugin Development	Added dynamic text fields to the GUI to allow engineers to set custom Critical and Warning tolerances.
Wed	25/03/2026	Plugin Development	Began conceptualizing the "Pre-Push Firewall" to intercept Git communications.

Day	Date	Phase	Work Description
Thu	26/03/2026	Plugin Development	Integrated `subprocess.run` modules to natively execute Git commands within KiCad.
Fri	27/03/2026	Plugin Development	Discovered the RAM-to-Disk State Mismatch bug during testing. Implemented the Auto-Save Integrity Protocol.
Sat	28/03/2026	Google Meet + Research	Attended Google Meet. Discussed the implementation of the Pre-Push Git Hook firewall.
Mon	30/03/2026	DevSecOps Firewall	Programmed the Boolean logic to explicitly block `git push` if vulnerabilities are detected.
Wed	01/04/2026	DevSecOps Firewall	Added the high-visibility `wx.MessageBox` modal error alerts for blocked deployments.
Thu	02/04/2026	Headless Cloud	Started implementation of the Cloud CI/CD backend integration for GitHub Actions.
Fri	03/04/2026	Headless Cloud	Wrote the automation logic to dynamically generate the `.github/workflows/security.yml` file.
Sat	04/04/2026	Google Meet	Attended Google Meet; discussed progress on headless execution.
Mon	06/04/2026	Headless Cloud	Investigated running KiCad Python without a display; implemented `xvfb-run` virtual framebuffer logic.
Wed	08/04/2026	Headless Cloud	Built the UX Bridge using `wx.CallLater` and Python's `webbrowser` module to auto-launch Chrome.
Fri	10/04/2026	Headless Cloud	Finalised the end-to-end cloud infrastructure testing; successfully executed a headless audit on GitHub servers.
Sat	11/04/2026	Google Meet	Attended Google Meet. Presented the fully functional V1 DevSecOps pipeline.
Mon	13/04/2026	Debugging	Explored edge cases with non-manifold board geometries and complex polygon structures.
Tue	14/04/2026	Debugging	Handled `CalledProcessError` exceptions for offline prototyping (Case Study 1).
Mon	20/04/2026	Refactoring	Cleaned up the Python codebase, added comprehensive docstrings, and optimized the matrix math.

Day	Date	Phase	Work Description
Thu	23/04/2026	Case Studies	Conducted extensive internal testing using Case Study 1 (Offline) and Case Study 2 (Cloud) parameters.
Sat	25/04/2026	Case Studies	Verified the reliability of the red "X" marker scaling on high-resolution displays.
Mon	27/04/2026	Report Preparation	Researched standard IEEE paper formatting for thesis writing.
Tue	28/04/2026	Report Preparation	Configured TeXStudio and MiKTeX for drafting the LaTeX internship report.
Wed	29/04/2026	Report Preparation	Drafted Chapter 1 (Introduction), Chapter 2 (Literature Survey), and Chapter 3 (Problem Statement).
Thu	30/04/2026	Report Preparation	Drafted Chapter 4 (Mathematical Modeling) and integrated TikZ vector diagrams.
Fri	01/05/2026	Report Preparation	Drafted Chapter 5 (GUI & Architecture) and Chapter 6 (Netlist Scanning).
Sat	02/05/2026	Report Preparation	Continued refining the Deep Copper Inspection mathematical documentation.
Mon	04/05/2026	Report Preparation	Drafted Chapter 7 (The Pre-Push Firewall) and detailed the Auto-Save logic.
Tue	05/05/2026	Report Preparation	Drafted Chapter 8 (Headless Cloud) and detailed the `xvfb` execution sequence.
Wed	06/05/2026	Report Preparation	Drafted Chapter 9 (Case Studies and System Evaluation).
Thu	07/05/2026	Report Preparation	Authored Chapter 10 (Conclusion, Limitations, and Future Scope).
Fri	08/05/2026	Final Polish	Programmatically embedded screenshot placeholders across all chapters for visual clarity.
Sat	09/05/2026	Google Meet + Report	Attended Google Meet. Generated concluding presentation script and reviewed final document structure.
Mon	11/05/2026	Final Polish	Added Appendix A (Daily Work Log) and generated the automated List of Figures.
Tue	12/05/2026	Final Polish	Expanded the Bibliography with IEEE standards, OWASP documentation, and hardware hacking literature.
Wed	13/05/2026	Work Submission	Finalised the report and packaged the complete `esim-hw-linter-main.zip` source code archive.
Thu	14/05/2026	Work Submission	Uploaded the final submission package to the FOSSEE Google Drive and raised the final GitHub PR.

Appendix B

Bibliography

1. FOSSEE Project, IIT Bombay. Available: `https://fossee.in`
2. eSim – Open Source EDA Tool for Circuit Design and Simulation. Available: `https://esim.fossee.in`
3. KiCad Official Python API (`pcbnew`) Reference Manual. Available: `https://docs.kicad.org/doxygen-python/`
4. Open Web Application Security Project (OWASP) Internet of Things Project - Hardware Security Attack Vectors. Available: `https://owasp.org/www-project-internet-of-things/`
5. Colin O'Flynn, *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks*, No Starch Press, 2021.
6. J. Grand, *Hardware Hacker: Secrets of the Trade*, 2014. Comprehensive overview of hardware reverse engineering and physical security.
7. S. Skorobogatov, *Physical attacks and tamper resistance*, Introduction to Hardware Security and Trust, Springer, 2011.
8. GitHub Actions Documentation: Automating CI/CD pipelines. Available: `https://docs.github.com/en/actions`
9. wxPython Graphical User Interface API Documentation. Available: `https://docs.wxpython.org/`
10. N. Asokan et al., "Hardware-assisted security for IoT," *IEEE Security & Privacy*, vol. 16, no. 3, 2018.
11. Xvfb - Virtual Framebuffer 'fake' X server. Available: `https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml`