Bachelor's Thesis Computing Science

Radboud University Nijmegen

Automation of Data Flow Analysis of Cryptographic Implementations

---

Author:
Catalin Sabau
s1070935

First assessor:
Dr. Ileana Buhan

Second assessor:
Prof. Dr. Lejla Batina

Daily supervisor:
Dr. Durba Chatterjee

January 30, 2025

Abstract: Side-channel attacks are one of the most predominant and practical attacks on cryptographic implementations, particularly targeting embedded devices. With the surge of low-cost and lightweight platforms such as RISC and ARM, software implementations have taken precedence in several embedded devices. This makes it imperative to evaluate the security of software implementations on such platforms and understand the root cause of any identified side-channel leaks. This thesis addresses this aspect by means of an interactive visualization framework that allows designers to visualize the execution of cryptographic implementations, data interactions and processing in architectural registers at the level of assembly instructions.

The framework is built on top of an architectural simulator for RISC-V that generates instruction level execution traces specifying the state of architectural registers throughout the execution. Our visualization framework takes the execution traces as input along with the leakage information, in the form of a TVLA test result and generates an interactive visualization depicting the data interactions. The framework supports several interactive features that enable user-friendly and efficient root-cause analysis. The tool, implemented using Python libraries and Plotly with interactive callbacks, offers adaptability for various cryptographic implementations targeting RISC-V platforms.

Keywords: Automation, Side-Channel Analysis, RISC-V, Pre-silicon, Data Flow Analysis, Cryptographic Security, Cryptographic Algorithm

Contents

Chapter 1

Introduction

Side channel attacks are a class of attacks that exploits weaknesses in the implementation of cryptographic algorithms. Implementing cryptographic algorithms typically follows a standard process: collecting traces of execution and applying statistical tests to detect potential side-channel leaks. However, while this approach identifies leaks, it does not reveal their root causes or provide strategies for mitigation. To address this gap, simulators are employed to correlate power-based leaks with the data processed within the processor during code execution. One such simulator, focusing on architectural-level leaks for RISC systems, is presented in [1].

This simulator includes two key components: a side-channel analysis module that detects leaks in simulated traces using statistical methods, and a data interaction module critical for identifying the underlying causes of leaks. However, the process of analyzing data flow—particularly tracking intermediate data through registers across multiple assembly instructions—becomes increasingly complex with tabular representations, especially for complete cryptographic implementations.

To overcome these challenges, this thesis introduces an automated visualization framework for data flow analysis. This framework systematically maps and visually tracks the movement of data within cryptographic implementations, enabling developers, researchers, and security evaluators to identify potential vulnerabilities more effectively. Notably, the proposed framework is designed to function independently of specific cryptographic algorithms, offering a versatile and interactive tool for understanding and mitigating data flow leaks in diverse implementations. By automating this process, the proposed framework enables the identification of key points of potential data leakage, thus providing a foundation for comprehensive security evaluations. This automation not only reduces the time and effort required for analysis but also enhances the accuracy and scalability of the process, making it applicable to a wide range of cryptographic software implementations.

Motivation

The primary motivation for this research arises from the need to enhance the security of cryptographic software implementations. Cryptographic algorithms, despite being designed for robust security, remain vulnerable to side-channel attacks[2]. These attacks exploit unintended information leakage, such as variations in execution timing, power consumption(the only type we cover in this paper), or electromagnetic emissions, to infer sensitive information about the cryptographic keys or plaintext data.

By visualizing the data flow within cryptographic implementations, this research facilitates side-channel analysis by allowing researchers to trace and monitor intermediate data states during the execution of cryptographic operations. The ability to track these states systematically enables the identification of potential vulnerabilities and side-channel leaks, contributing to the development of more resilient cryptographic software. Moreover, the visualization aids in understanding the complex interplay of data within cryptographic processes, making the framework a valuable tool for security researchers, evaluators and developers in the field.

Major Contributions

The core contributions of the thesis are :

- **Automated Framework**: Development of an automated framework for data flow analysis, which streamlines the process of identifying and visualizing data movement in cryptographic implementations.
- **Interactive Features**: We add several features to aid in the data flow analyses, such as searching by the index for precision, searching for Program Counter(PC) to observe the evolution of the instructions at different indexes but with the same PC and figure-related features which will help you in your analyses.
- **Enhanced Visualization through Color-Coding**: Introduction of a proper color-coding scheme within the visualizations to improve clarity and interpretability of the data flow, making it easier to distinguish between different states and flows.
- **Comprehensive Legend Integration**: Inclusion of a detailed legend to provide a contextual understanding of the visualized data, ensuring that users can easily interpret and analyze the presented information.

These contributions collectively advance the state of the art by addressing both the technical and usability aspects of data flow analysis in cryptographic software. The proposed framework not only facilitates more efficient

and accurate security evaluations but also enhances the accessibility and usability of the analysis for researchers and practitioners alike.

Outline of the Thesis

The thesis is organized as follows:

- **Chapter 1: Introduction**
  This chapter introduces the thesis topic, outlines the research questions, and sets the context and motivation for the work.
- **Chapter 2: Background**
  This chapter provides the necessary background information required for understanding the thesis.
- **Chapter 3: Related Work**
  This chapter offers a review of the related work, highlighting what has been done in the field.
- **Chapter 4: Interactive Visualization**
  This chapter discusses several features of interactive visualization.
- **Chapter 5: Visualization Code**
  This chapter presents a discussion about the code implementation, including files, different approaches, improvements made, and the challenges encountered.
- **Chapter 6: Conclusion**
  The final chapter summarizes the findings, discusses their implications, and suggests directions for future work.

Abbreviations

The following list describes the abbreviations used in this thesis.

AES – Advanced Encryption Standard
API – Application Programming Interface
ARM – Advanced RISC Machine
HD – Hamming Distance
HW – Hamming Weight
ID – Identity
ISA – Instruction Set Architecture
LUT – Look-Up Table
PC – Program Counter
RISC – Reduced Instruction Set Computing
RISC-V – Open standard RISC five
RV32I – 32-bit RISC-V base integer instruction set
RV64I – 64-bit RISC-V base integer instruction set
S-box – Substitution box used in cryptography
TVLA – Test Vector Leakage Assessment
WebGL – Web Graphics Library
XOR – Exclusive OR logical operation

Chapter 2

Background

This chapter provides an overview of the concepts required to understand this work. We provide a brief overview of Advanced Encryption Standard (AES) that is used to demonstrate the tool, RISC-V architecture, which is the target platform in our analysis, and Python-based libraries such as Plotly and Dash, which are used to implement the visualization framework.

2.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a widely recognized symmetric-key cryptographic cipher adopted as a standard by the National Institute of Standards and Technology (NIST). This cipher is based on the Rijndael algorithm [3] and processes an N-bit plaintext into an N-bit ciphertext using an N-bit key. AES supports key and plaintext sizes of 128, 192, or 256 bits, with the number of transformation rounds determined by the key size: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. These variations offer increasing levels of security and complexity, making AES suitable for a wide range of applications.

Each round of the AES-128 algorithm transforms the input state, represented as a 16-byte (128-bit) matrix S(4x4 blocks of 128 / 16 = 8 bits each), through a sequence of operations designed to ensure both confusion and diffusion. As we can see in Figure 2.1 these operations include [2]:

- **SubBytes()** — Substitutes each byte in the state matrix with a corresponding value from the S-box, a nonlinear substitution table, thereby introducing non-linearity and enhancing confusion. The S-Box transforms 8-bit input into 8-bit secret data using a precomputed lookup table (LUT).
- **ShiftRows()** — Cyclically shifts the rows of the state matrix by varying offsets: the first row remains unchanged, while subsequent rows are

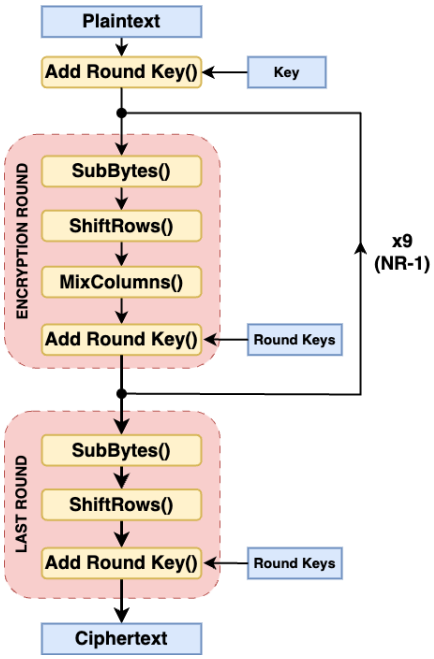


Figure 2.1: Schematic representation of AES-128. Adapted from [2].

shifted by one, two, and three positions, respectively, ensuring diffusion.

- **MixColumns()** — Treats each column of the state matrix as a polynomial and multiplies it by a fixed polynomial matrix, enhancing diffusion within columns.
- **AddRoundKey()** — Performs a bitwise XOR between the state matrix and the corresponding round key, ensuring encryption is heavily dependent on the key.

The complete sequence of AES-128 operations is visually depicted in Figure 2.2, which provides a matrix-based visualization of the encryption steps based on Figure 2.1.

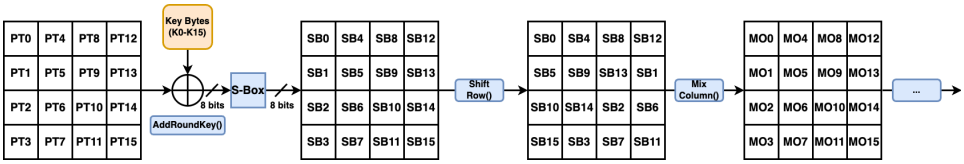


Figure 2.2: Schematic representation of AES-128 steps. Adapted from [1].

2.2 RISC-V

RISC-V is an open, free, and extensible Instruction Set Architecture (ISA). Unlike proprietary ISAs such as x86 or ARM, RISC-V is released under permissive open licenses. At its core, RISC-V embodies the principles of Reduced Instruction Set Computing (RISC). It features a relatively small set of simple instructions that can be executed quickly. By maintaining an orthogonal and minimal ISA, RISC-V simplifies hardware implementation, enabling efficient pipelines, reduced power consumption, and simplified control logic. It adheres to a LOAD/STORE architecture, meaning that operations cannot be performed directly on memory. Instead, data must first be moved to registers. This design ensures that any data-dependent activity is visible in the register state [4] [1].

Instruction Types in RISC-V

RISC-V supports a variety of fundamental operations that enable efficient computation. Key instruction types [4] include:

- **ARITHMETIC**: The ARITHMETIC operations encompass basic mathematical computations such as addition, subtraction, multiplication, and division between registers. These are fundamental to numerical processing and algorithm implementation.
- **LOAD**: The LOAD instruction transfers data from memory into a register. It is a crucial operation in the LOAD/STORE architecture, ensuring that data is available for processing in registers.
- **STORE**: The STORE instruction writes data from a register to memory, allowing results to be saved or shared with other parts of a program.
- **SHIFT**: The SHIFT instructions, such as logical and arithmetic shifts, modify the bitwise representation of data, useful in various computational scenarios.
- **BRANCH**: The BRANCH instructions enable conditional or unconditional changes in program flow, facilitating decision-making and loops within programs.
- **UBRANCH**: The UNCONDITIONAL BRANCH instructions alter the flow of execution without requiring any condition evaluation. They are vital for directing program control explicitly and implementing control flow structures.

These operations form the foundation of RISC-V's functionality, enabling efficient data manipulation, memory access, and control flow management.

Reg	Alias	Description	Preserved Across Calls
x0	zero	Hard-wired zero	Immutable
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	Unallocatable
x4	tp	Thread pointer	Unallocatable
x5	t0	Temporary/alt. link register	No
x6-7	t1-2	Temporary registers	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10-11	a0-1	Function arg./return value	No
x12-17	a2-7	Function arguments	No
x18-27	s2-11	Saved registers	Yes
x28-31	t3-6	Temporary registers	No

Table 2.1: Register table of RISC-V. Adapted from [4].

Register Set and Calling Conventions

The base integer register file of RISC-V comprises 32 registers, each capable of holding a word-sized value (commonly, the 32-bit instruction set in the RV32I variant or the 64-bit instruction set in the RV64I variant is used). These registers are named `x0` through `x31`, each serving a specific purpose as outlined in Table 2.1. Notably, `x0` is hardwired to zero and cannot be modified, which simplifies operations such as nullifying register values without requiring a separate constant.

The registers in the RISC-V ISA are categorized into three primary groups: temporary (caller-saved) registers, saved (callee-saved) registers, and special-purpose registers. Temporary registers are used for intermediate calculations and do not retain their values across function calls; the calling function (caller) is responsible for saving these values if necessary. Conversely, saved registers preserve their contents across function calls, as the called function (callee) is required to maintain their values. Special-purpose registers include the stack pointer (`x2`), global pointer (`x3`), and thread pointer (`x4`), which are reserved for compiler or runtime usage and should not be altered in most application-level code.

2.3 Side-Channel Attacks

Side-channel attacks exploit unintended physical or logical channels to extract sensitive information from a system, often bypassing the mathematical robustness of cryptographic algorithms [5]. These attacks leverage observable characteristics such as power consumption, electromagnetic emissions, timing variations, or even acoustic signals, which can reveal internal states or the secret key. A critical aspect of defending against side-channel attacks is minimizing these unintended leakages through careful implementation and robust design [6].

The security of cryptographic algorithms is not only dependent on their mathematical strength but also on their implementation. Even robust algorithms like AES can become vulnerable if the implementation introduces side-channel leakages. There are several side-channel attacks based on the exploited physical information, such as timing-based attacks, power-based attacks, and cache attacks; we focus on power-based attacks in this paper. Power-based side-channel attacks are detected using statistical methods. We describe one such test based on hypothesis testing in the following section.

Test Vector Leakage Assessment

Test Vector Leakage Assessment (TVLA) [7] has emerged as a widely used technique to identify leakage points in cryptographic implementations. TVLA uses Welch's t-test to detect statistical deviation of two distributions, one corresponding to a set of fixed inputs and another corresponding to a set of randomly chosen data. Each set comprises a set of power traces. The t-test outputs a t-score, which, when greater than 4.5 or less than -4.5, indicates a leak. While TVLA is effective in detecting leakage as we can see in Figure 2.3, it lacks the ability to pinpoint its root causes, making further analysis and mitigation difficult [6] [1].

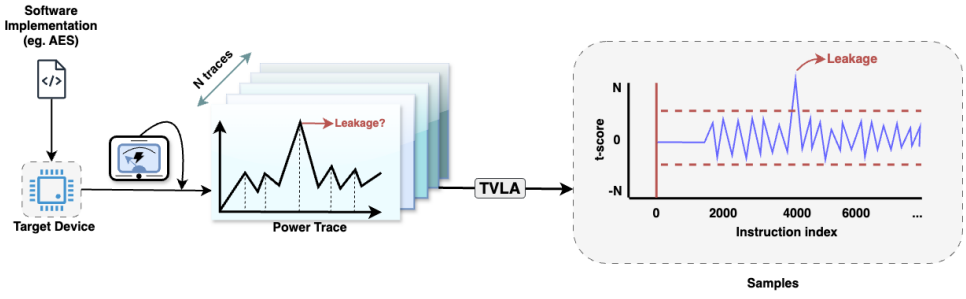


Figure 2.3: Schematic representation of a leakage model

2.4 Architecture Level Simulator for RISC-V

ARCHER [1] is a dedicated architecture-level tool designed for side-channel analysis in RISC-V processors. Its primary purpose is to identify the root causes of side-channel leaks in cryptographic implementations at the architectural level, enabling targeted and effective mitigation strategies. Below, we outline the key features and functionality of ARCHER:

2.4.1 Archer Key Features

- **Execution Trace and Power Simulation:**
  - ARCHER processes binary files of target implementations to generate detailed execution traces at the instruction level.
  - Using a leakage model, the tool transforms execution traces into simulated power traces, reflecting hypothetical power consumption (Figure 2.3).
  - Supported leakage models include:
    - Hamming Weight (HW) a value-based leakage model
    - Hamming Distance (HD) a transition-based leakage model
    - Identity (ID) a value-based leakage model
- **Purpose and Methodology:**
  - ARCHER identifies the root causes of side-channel leaks by correlating execution behavior with power side-channel leakage results.
  - By focusing on the architectural level, it isolates leaks from microarchitectural effects, providing insights into implementation-level vulnerabilities.
- **Visualizations:**
  - ARCHER incorporates some limited visualization tools that highlight leaking instructions and their causes.
  - These visualizations, combined with statistical results from TVLA, offer an intuitive interface for analyzing execution flow and data dependencies.
- **Support for Cryptographic Algorithms:**
  - The tool supports pre-silicon analysis of various cryptographic algorithms, including AES and ASCON.
  - It detects vulnerabilities in unprotected implementations and aids developers in enhancing software resilience.

2.4.2 Motivation for Interactive Visualizations

The interactive visualization capabilities of ARCHER play a crucial role in analyzing the complex interactions between instructions and data in cryptographic algorithms. These visualizations:

- Enable pinpointing of specific instructions responsible for leaks.
- Highlight patterns in intermediate value propagation and register usage.
- Simplify the process of communicating findings and deriving actionable insights.

By combining detailed trace analysis, leakage modeling, and interactive feedback, ARCHER bridges the gap between detecting and mitigating side-channel leaks at the architectural level, fostering a deeper understanding of cryptographic security on RISC-V platforms.

2.5 Python Graphing Libraries: Plotly and Dash

Python is a versatile programming language in data visualization due to its rich ecosystem of libraries [8] [9]. Among the most popular graphing libraries are Plotly and Dash, which are widely used for creating interactive, visually appealing, and customizable visualizations.

Plotly

Plotly [10] is an open-source graphing library that enables users to create interactive and publication-quality visualizations. It supports a wide range of chart types, including scatter plots, line graphs, bar charts, pie charts, heatmaps, 3D plots, and more. One of Plotly's standout features is its interactivity, which allows users to zoom, pan, hover, and export visualizations seamlessly.

Plotly is designed to work across various platforms, making it ideal for sharing insights in web applications or embedding them in reports and dashboards. It integrates smoothly with popular data analysis libraries like `pandas` and `NumPy`, enabling users to create visualizations directly from dataframes.

Some of the commonly used methods and functions in Plotly include:

- **plotly.graph_objects:** A lower-level module that provides fine-grained control over chart customization. Key functions include:
  - `go.Figure()`: Constructs figures for custom layouts and traces.
  - `go.Scatter()`: Plots scatter data with advanced options.

  - `go.Layout()`: Configures layout properties such as titles, axes, and annotations.

- `add_trace()`: Adds multiple data series to a single figure.
- `update_layout()`: Modifies the layout of a chart dynamically.

Dash

Dash [11] is a Python framework built on top of Plotly, Flask, and React.js, designed for creating interactive web applications for data visualization and analysis. Unlike Plotly, which focuses on individual graphs, Dash provides a full-fledged platform to develop dashboards and applications with interconnected components and interactive features.

Dash applications are structured as a combination of three main components:

1. **Layout:** Defines the structure of the application using Dash HTML and core components such as dropdowns, sliders, and graphs.
   - Commonly used functions:
     - `dash.html.Div()`: Creates containers for grouping components.
     - `dash.dcc.Graph()`: Embeds Plotly figures into the app.
     - `dash.dcc.Store()`: Stores user input and other types of data.
2. **Callbacks:** Enable interactivity by connecting inputs (e.g., dropdown selections) to outputs (e.g., graph updates) using Python functions.
   - `@app.callback()`: A decorator that defines the logic for interactivity.
3. **Server:** Executes the application and handles user requests.
   - `app.run_server()`: Launches the Dash application.

Dash's integration with Plotly makes it an ideal choice for developing data-driven dashboards that combine static and dynamic visualizations with user input. Its modular approach simplifies the creation of complex applications, allowing to build interactive tools for analysis.

Plotly and Dash together offer powerful tools for creating engaging visualizations and interactive applications as we will see in the following chapters. While Plotly focuses on creating standalone visualizations, Dash extends this capability to build dynamic and user-friendly dashboards. Their intuitive APIs and extensive functionality make them indispensable for professionals in fields such as data science, business analytics, and academic research.

Chapter 3

Related Work

Although no existing work directly automates data flow analysis for cryptographic implementation with a focus on side-channel analysis [10] [12], this research incorporates concepts from established visualization frameworks, particularly Plotly which is a versatile visualization library known for its ability to create interactive and customizable data representations. Examples from [10] showcase a variety of advanced visualization techniques, which have inspired the design of the visual components in this project.

Further inspiration is drawn from data science projects utilizing Plotly, such as those detailed at [12]. These projects emphasize clarity, interactivity, and accessibility in data visualization, principles that are critical for making complex datasets more interpretable. While these projects focus on general-purpose visualization, their underlying principles have been adapted to suit the specific needs of cryptographic data flow analysis.

By building on the foundational concepts of tools like Plotly and adapting them to an entirely new context, this work offers a unique framework for exploring and mitigating side-channel vulnerabilities in cryptographic software.

Novelty of our Approach

While the visualization concepts used in this project build on the capabilities of Plotly, the implementation represents a novel application tailored to the unique demands of analyzing cryptographic implementation. Unlike generic visualization tools, this work introduces domain-specific innovations, such as:

- **Automated Data Flow Analysis**: A fully automated process for identifying and visualizing intermediate data points within cryptographic computations.
- **Specialized Visualization Techniques**: Customized graphical representations designed specifically to highlight data flows relevant to

side-channel vulnerabilities.

- **Enhanced Usability**: Intuitive color schemes, interactive elements, and detailed legends to aid researchers in interpreting complex cryptographic processes.

These enhancements distinguish this work from existing frameworks and projects, as no prior approach has integrated such visualization techniques with automated data flow analysis in the cryptographic domain.

Chapter 4

Interactive Visualizations

This chapter presents an interactive data flow visualization framework for software implementations of cryptographic algorithms, which forms the core focus of the thesis. To demonstrate the features of the visualization framework, we use a C implementation of unmasked AES-128[3] executed on PicoRV32 (a RISC-V core)[13]. We analyze the implementation using two leakage models, namely HD and HW. The tool is implemented in Python3 using `plotly` package. The possible visualizations of the framework are depicted in Figure 4.1. The tool takes as input one mandatory file(trace reference) and combines it with various optional files(TVLA leakage model and Cryptographic algorithm markers), resulting in a few possible visualizations. The details of the inputs are provided in the next section 4.1.

4.1 Files needed for visualizations

In order to build the visualization, you need the following files:

4.1.1 Trace Reference/Execution Trace

The dataset file provides a detailed breakdown of assembly instructions executed by the program. Each row represents an instruction execution with various associated details, including PC, instruction name and type, and operand values. Additionally, it includes states of all 32 registers, enabling the analysis of system behavior at the instruction level during the execution.

Visualization: Figure 4.2 shows the entire set of instructions executed by the implementation of the target device before any leakage model is applied.

4.1.2 TVLA Leakage Analysis

The dataset file provides results from a TVLA for identifying potential information leakage during cryptographic operations. Each row represents

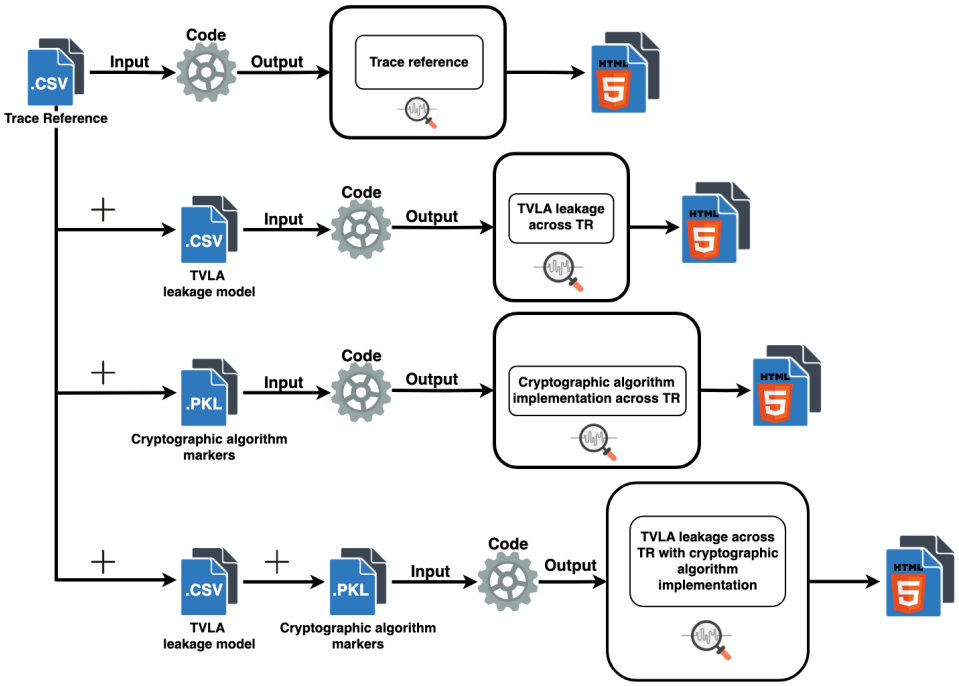


Figure 4.1: Toolflow of the Visualization Framework

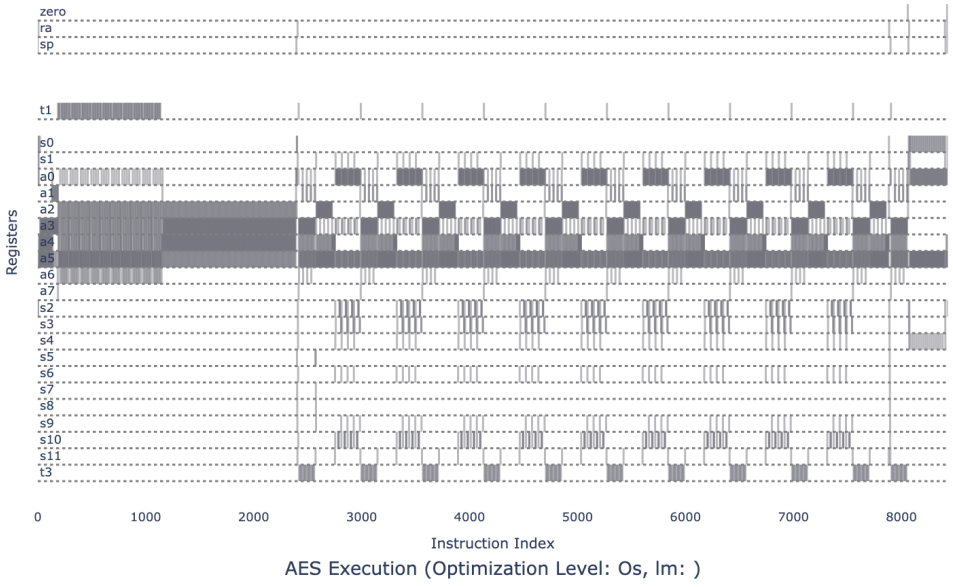


Figure 4.2: Execution Trace Visualization

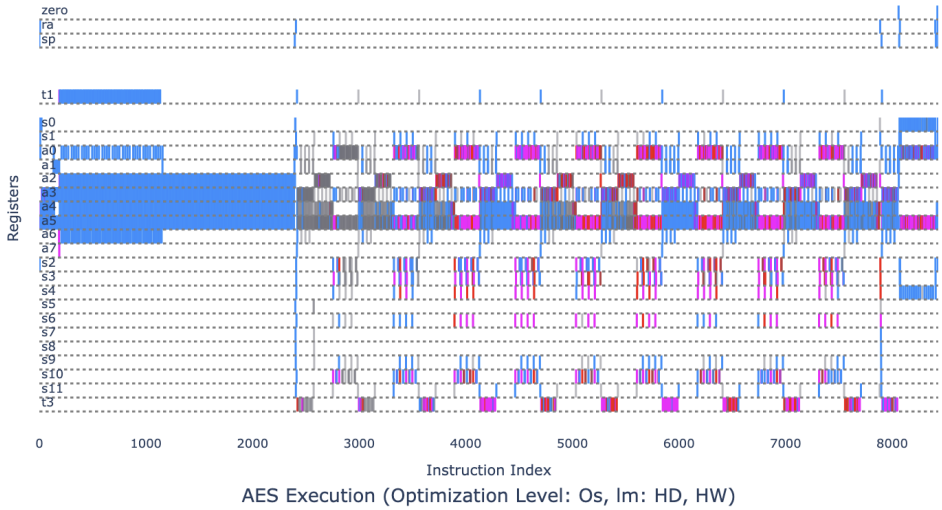


Figure 4.3: Visualization of TVLA leakage across instruction indices

a specific instruction with its associated T-score, indicating whether the instruction shows leakage depending on the input(nTraces) where all plaintext bytes remain the same except for one byte or multiple, which is/are randomly varied. A T-score exceeding the defined threshold signifies a potential leakage and is marked with `1` otherwise with `0`.

Visualisation: Figure 4.3 highlights the leaky instructions using red and blue colors. Red indicates leakage as per HD leakage model, whereas blue indicates leaks corresponding to HW leakage model.

4.1.3 AES Implementation Markers

This dataset provides detailed marker indexes and register values required for analyzing and implementing cryptographic algorithm encryption over the current trace reference. It categorizes information into specific registers, enabling targeted analysis of instruction sequences essential for cryptographic operations. For unprotected AES implementation, the intermediate data(data generated during the code execution [1]) includes plaintext bytes (PT), SubBytes output (SB), RoundKey bytes (RK), MixColumn outputs (MC_OUT), and key bytes (K). These data are stored as a dictionary in a .pkl file. The keys of the dictionary are the architectural registers (`a1, a2, ` $\cdots$), and each key is associated with a list that includes the instruction indexes, where an intermediate occurs in that register.

Visualisation: Figure 4.4 depicts the intermediate data using various markers, each for a different intermediate value. This figure provides insights into the register usage pattern for a given implementation, how long

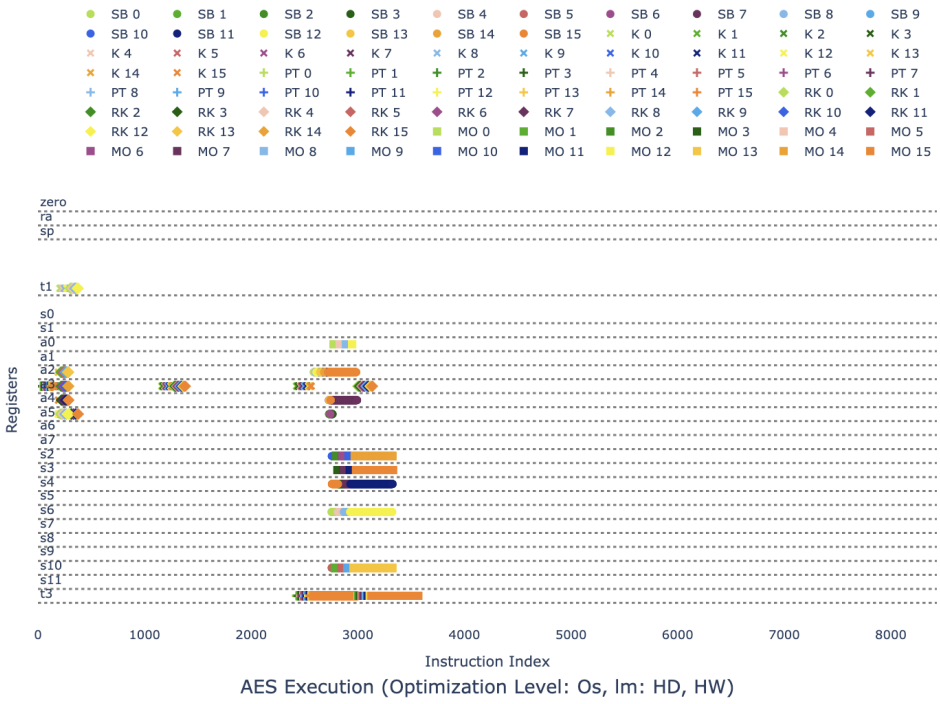


Figure 4.4: Visualization of AES marker indexes and register utilization

the values remain in the register, and which registers are used by which operations of the cryptographic algorithm.

By combining everything, you obtain Figure 4.5:

4.2 Supported Operations and Features

The visualization framework supports several key operations designed to enhance user interaction and data exploration. The inspiration for the features was taken when reading [1] and discussing with the project owners what are they looking for to achieve with this tool. The layout of the interactive visualizations along with the features is depicted in Figure 4.6. We tabulate the features in Table 4.1 and describe each feature as follows:

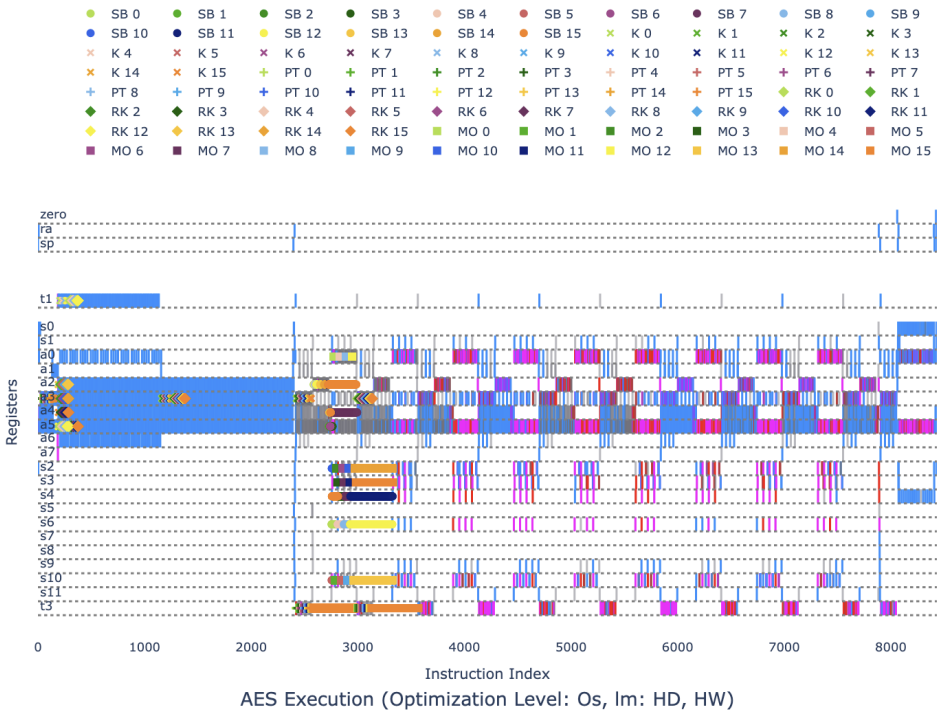


Figure 4.5: Visualization of unmasked AES

Feature	**Description**	**How to Access**
Index Range Selection (Sec. 4.2.1)	Allows users to select a specific range of indexes.	Range slider
Index Opacity Selection (Sec. 4.2.2)	Adjusts the transparency of the indexes for clarity.	Index Opacity slider
Markers Opacity Selection (Sec. 4.2.3)	Adjusts the transparency of the markers for clarity.	Marker Opacity slider
Search by Index (Sec. 4.2.4)	Find a specific index.	Search bar (left container)
Search by PC (Sec. 4.2.5)	Find specific index(es) based on PC.	Search bar (right container)
Screenshot (Sec. 4.2.6)	Captures the current visualization as an HTML file.	"Screenshot" button below the image
Legend (Sec. 4.2.7)	Toggle visibility of the markers for clarity.	Above the figure
Image-Related Features (Sec. 4.2.8)	Includes the following tools: pan, zoom, zoom in, zoom out, zoom scroll, autoscale, reset axes, and box select.	Toolbar (right corner of the figure) or mouse interactions

Table 4.1: Summary of Interactive features supported in the Visualization Framework.

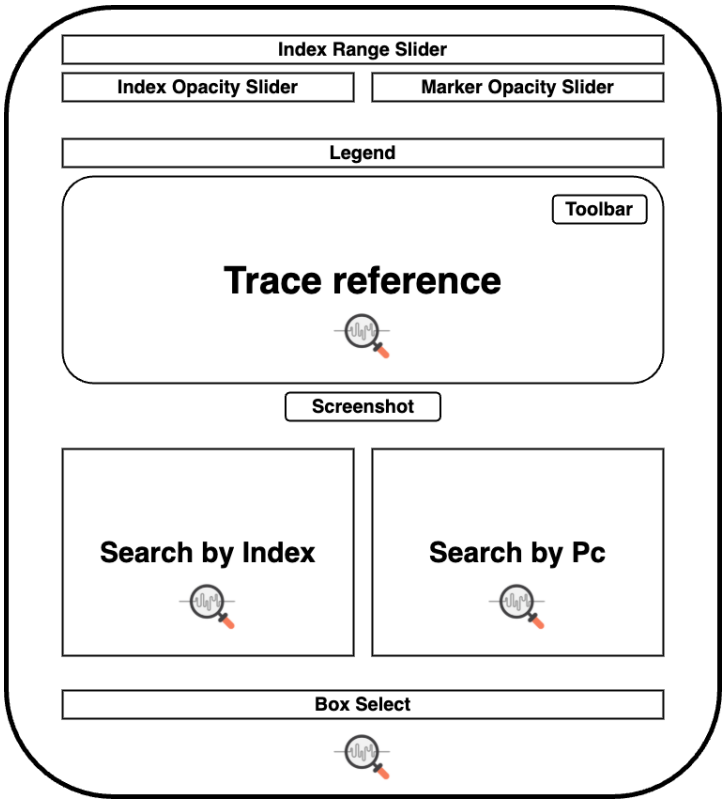


Figure 4.6: Interactive Visualization Layout

4.2.1 Index Range Selection

Description: The visualization framework allows users to select a specific range of instructions. This feature is particularly useful for focusing on subsets of instructions within the visualization, enabling targeted analysis or highlighting specific sections of interest.

How to use: Users can adjust the range by dragging one end of the slider at a time to define the desired range. (Figure 4.7)

Where to find it: The "Range Slider" is located at the very top and is the longest slider in the interface. (Figure 4.6)

Figure 4.7: Range slider used for selecting a specific range of indexes.

4.2.2 Indexes Opacity Selection

Description: To enhance visualization clarity, users can adjust the transparency of instructions. This feature is especially useful for improving visibility when overlapping elements are present in the visualization.

How to use: Users can modify the instruction opacity using the "Index Opacity" slider. (Figure 4.8)

Where to find it: The "Index Opacity" slider is located at the very top, on the left side. (Figure 4.6)

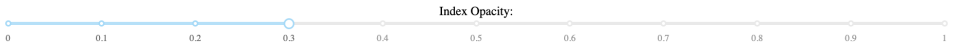


Figure 4.8: Index Opacity slider for adjusting transparency of indexes.

4.2.3 Markers Opacity Selection

Description: Similar to the "Index Opacity" feature, the framework allows users to adjust the transparency of markers for improved clarity. The "Marker Opacity" slider provides flexibility in visualizing dense or overlapping data points effectively.

How to use: Users can modify the marker opacity using the "Marker Opacity" slider. (Figure 4.9)

Where to find it: The "Marker Opacity" slider is located at the very top, on the right side. (Figure 4.6)

Figure 4.9: Marker Opacity slider for adjusting transparency of markers.

4.2.4 Search by Index

Description: The "Search by Index" feature allows users to locate specific instructions efficiently by knowing the index. A search bar located in the left container facilitates this functionality, enabling precise and quick navigation within the dataset if the zoom checkbox is selected.

How to use: Users can simply type the desired index instruction into the search bar to visualize it. The selected index instruction will be highlighted with three different colors: one for the two predecessors, one for the current index, and one for the two successors. Additionally, users can enable the 'Zoom' checkbox to quickly locate the index instruction within the visualization. (Figure 4.10, Table 4.2)

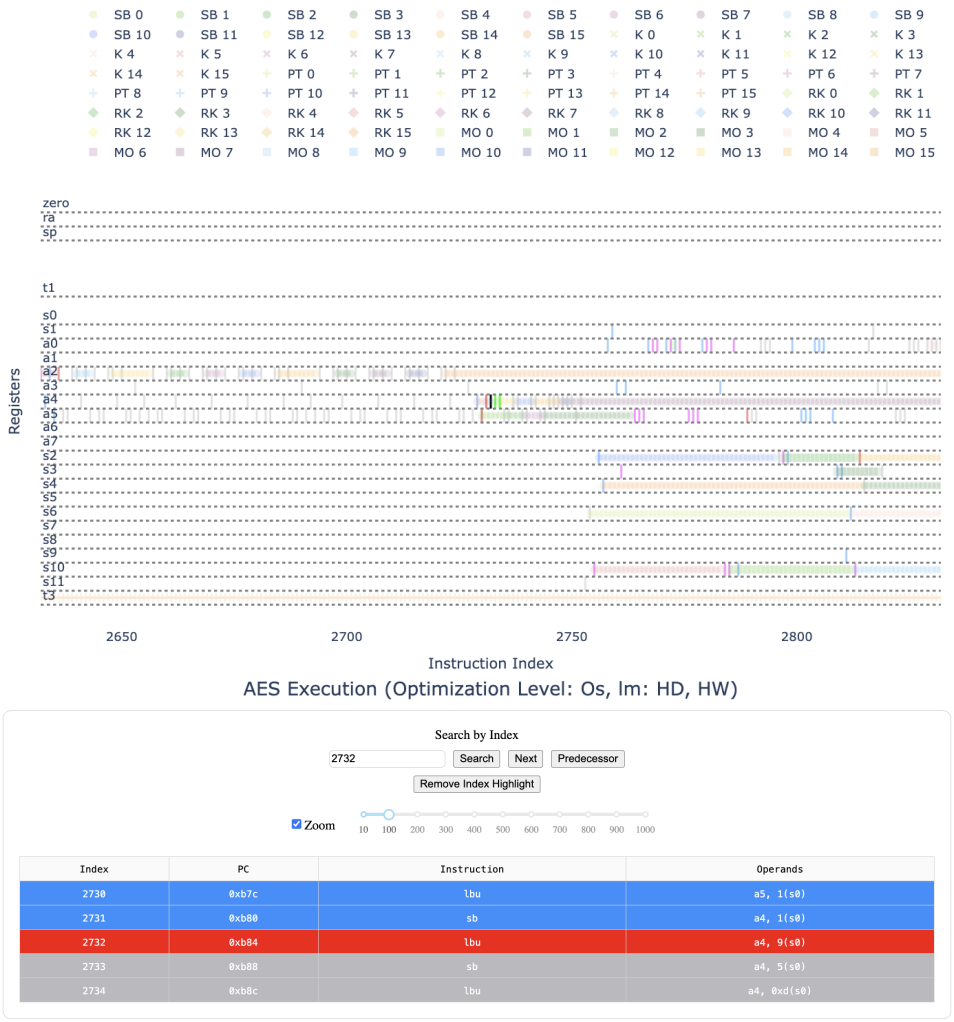


Figure 4.10: Search by Index

Where to find it: The "Search by Index" functionalities are found in the left container below the figure. (Figure 4.6)

Note: "Search by Index" can be combined with the "Search by PC" functionalities.

4.2.5 Search by PC

Description: This feature allows users to search for instruction indexes based on the PC. A dedicated search bar supports the functionality of searching for multiple PCs using comma-separated values, making it a convenient method for program-related data analysis.

Functionality	Description
"Search" Button (Figure 4.10)	Initiates the search for the specified index instruction and displays the instructions indexes accordingly.
"Next" Button (Figure 4.10)	Moves the search to the next instruction index to the right.
"Predecessor" Button (Figure 4.10)	Moves the search to the previous instruction index to the left.
"Remove Index Highlight" Button (Figure 4.10)	Removes the current search, erases the highlights applied to the traces, and restores them to their original color.
Data Table (Figure 4.10)	Displays the searched instructions and provides additional information about them, such as index, PC, instruction, operands. The rows are highlighted to indicate the corresponding leakage model of the instruction.

Table 4.2: Buttons and their functionality in the visualization framework.

How to use: Users can type the desired PC instruction(s) into the search bar to locate and visualize all matching instruction indexes. The indexes corresponding to the searched PCs will be highlighted in a color distinct from the "Search by Index" section. Additionally, users have the option to highlight all matches or selectively highlight specific ones. (Figure 4.6, Table 4.3)

Where to find it: The "Search by PC" functionalities are found in the right container below the figure. (Figure 4.6)

Note: "Search by PC" can be combined with the "Search by Index" functionalities.

4.2.6 Screenshot

Description: The framework offers a feature to capture the current visualization and save it as an HTML file. This functionality provides a convenient method to preserve and share the current state of the visualization.

How to use: Simply press the "Screenshot" button in Figure 4.6, and the current visualization will be saved to the path you specified.

Where to find it: The "Screenshot" button is located directly below the figure within the interface. (Figure 4.6)

Note 1: The autoscale and reset axes buttons functionalities are inversed.

Note 2: The user can still access the full figure if needed by zooming out or using pan.

Note 3: Box Select/Lasso Select will not work in the saved file.

4.2.7 Legend

Description: The Legend feature allows users to toggle the visibility of

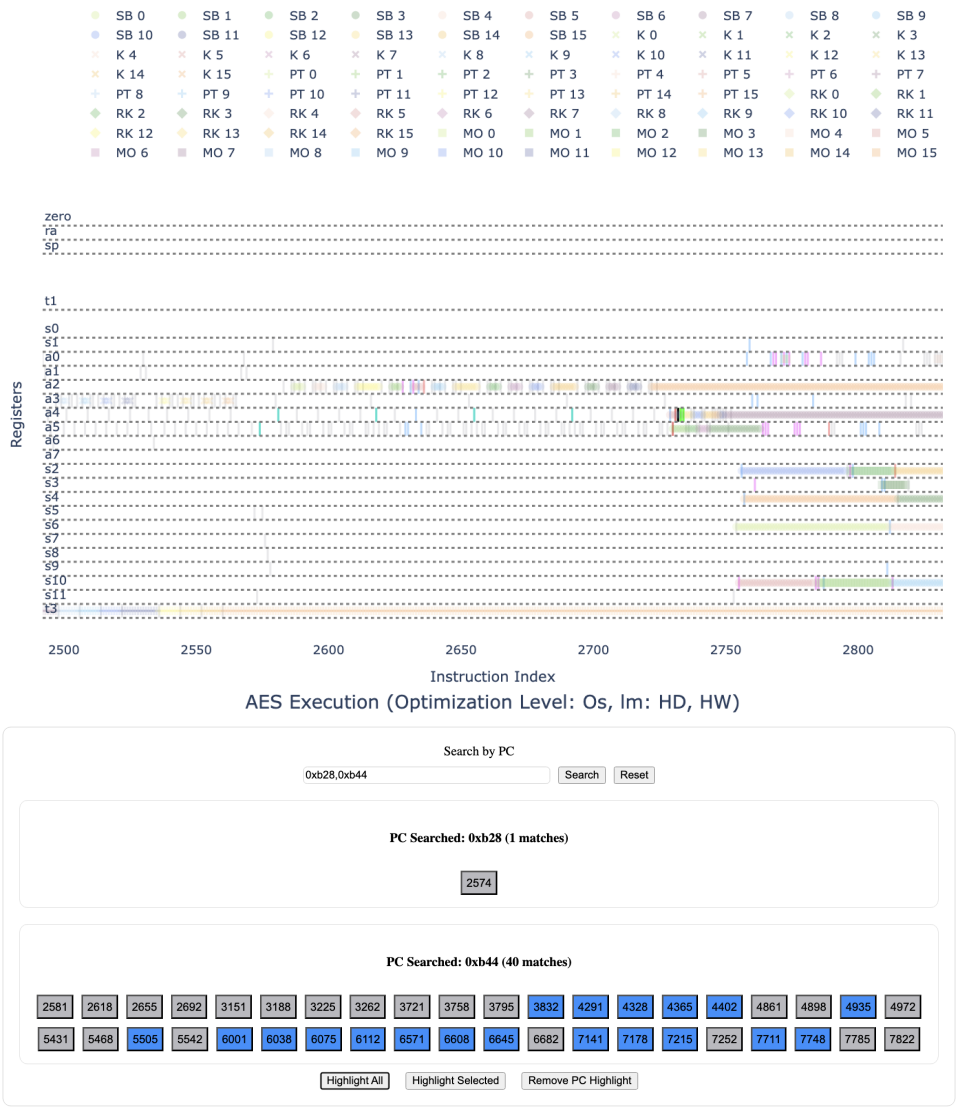


Figure 4.11: Search by PC

markers within the visualization. This functionality is particularly useful for reducing visual clutter and focusing on specific layers.

How to use: Users can click once on a marker label to toggle its visibility on or off. Additionally, performing a double click on a marker label hides all other markers, leaving only the selected one visible. (Figure 4.12)

Where to find it: The "Legend" is located above the figure. (Figure 4.6)

Functionalities	Description
"Search" Button (Figure 4.11)	Initiates the search for the specified PC instruction and displays the instruction indexes accordingly.
"Reset" Button (Figure 4.11)	Removes the current search and the current highlight.
"Highlight All" Button (Figure 4.11)	Highlights all the instruction indexes in the current search.
"Highlight Selected" Button (Figure 4.11)	Highlights only the selected instruction indexes in the current search.
"Remove PC Highlight" Button (Figure 4.11)	Removes the current highlight applied to the traces, and restores them to their original color.
"Index" Button(s) (Figure 4.11)	Displays the searched instruction indexes and provides additional information by being highlighted to indicate the corresponding leakage model of the instruction.
Opacity "Index" Button(s) (Figure 4.11)	Those which are selected are darker in color compared to a faded color of those not selected.

Table 4.3: Buttons and their functionality in the visualization framework.

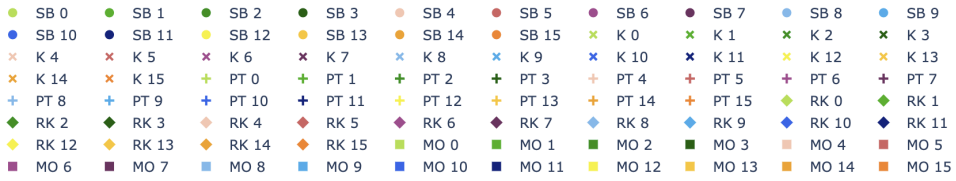


Figure 4.12: Legend toggle for controlling marker visibility.

4.2.8 Image-Interactive Features

Description: The visualization framework includes a comprehensive suite of interactive image tools, such as pan, zoom, zoom in, zoom out, zoom scroll, autoscale, reset axes, and box select. These tools enhance the interactivity and versatility of the visualization, allowing users to explore the data in detail.

Where to find it: These tools are accessible through the toolbar located at the top-right corner of the Figure 4.6.

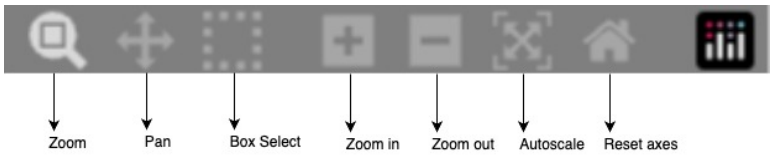


Figure 4.13: Toolbar tools for image manipulation

Pan

Description: The "Pan" feature allows users to move the visualization horizontally or vertically to explore different areas of the data set without altering the zoom level. This is particularly helpful for examining large visualizations or data distributed over a wide range.

How to use it: Activate the "Pan" tool from the toolbar Figure 4.13. Click and hold the left mouse button while dragging the visualization to move it in the desired direction.

Zoom

Description: The "Zoom" feature enables users to magnify a specific area of the visualization, making it easier to focus on certain sections of data points.

How to use it: Select the "Zoom" tool from the toolbar Figure 4.13. Click and drag to create a rectangular region around the area you wish to zoom into.

Zoom In

Description: The "Zoom In" feature increases the magnification of the visualization incrementally, providing a closer view of the data.

How to use it: Click the "Zoom In" button on the toolbar Figure 4.13 repeatedly to increase the magnification step by step.

Zoom Out

Description: The "Zoom Out" feature decreases the magnification of the visualization incrementally, allowing users to view a broader range of data.

How to use it: Click the "Zoom Out" button on the toolbar Figure 4.13 repeatedly to reduce the magnification step by step.

Zoom Scroll

Description: "Zoom Scroll" provides users with a quick way to zoom in and out using the scroll wheel on their mouse, offering smooth magnification control. Works regardless of the selection from the toolbar.

How to use it: Hover your mouse over the visualization and scroll up to zoom in or scroll down to zoom out.

Autoscale

Description: The "Autoscale" feature automatically adjusts the visualization to fit all data points within the selected range of the index slider,

ensuring that no part of the data is left behind by undoing any modifications made through panning, zooming, scrolling or box select.

How to use it: Click the "Autoscale" button on the toolbar Figure 4.13 to reset the visualization to display all data points in the current selected range.

Reset Axes

Description: The "Reset Axes" feature restores the axes of the visualization to their original state, undoing any modifications made through panning, zooming, scrolling or box select. It differs from the autoscale feature as it restores all data points, regardless of the currently selected range.

How to use it: Click the "Reset Axes" button on the toolbar Figure 4.13 to reset the visualization to its default axes configuration.

Box Select

Description: "Box Select" enables users to select a specific subset of data points by drawing a rectangular region on the area of interest. Figure 4.14, 4.15

How to use it: Click the "Box Select" button on the toolbar Figure 4.13. Click and drag to draw a rectangular region over the desired register(s) line(s) covering how many data points you want. This feature highlights the selected data while fading the rest and generates a table below the page containing various information, such as the index, PC, instruction, operands, register, and marker (if a marker exists on top of the instruction).

Note: Use the "Copy to Clipboard" button if you want to copy the current selection shown in the table.

4.3 Interactions between features

The visualization framework offers various features that work collaboratively to enhance the clarity and usability of the visualization. Each feature is designed to assist and complement others, allowing users to perform targeted analyses, improve visualization precision, and emphasize specific data points.

It is important to note that the "Screenshot" feature operates independently of the other features and captures the current state of the visualization regardless of the applied settings or adjustments. This allows users to preserve and share their insights effectively. Similarly, the Toolbar (Figure 4.13) provides a set of tools (e.g., zoom, pan, and box select) that enhance the interactive experience but remain independent of specific features in the matrix. Together, these elements ensure flexibility and efficiency in data exploration.

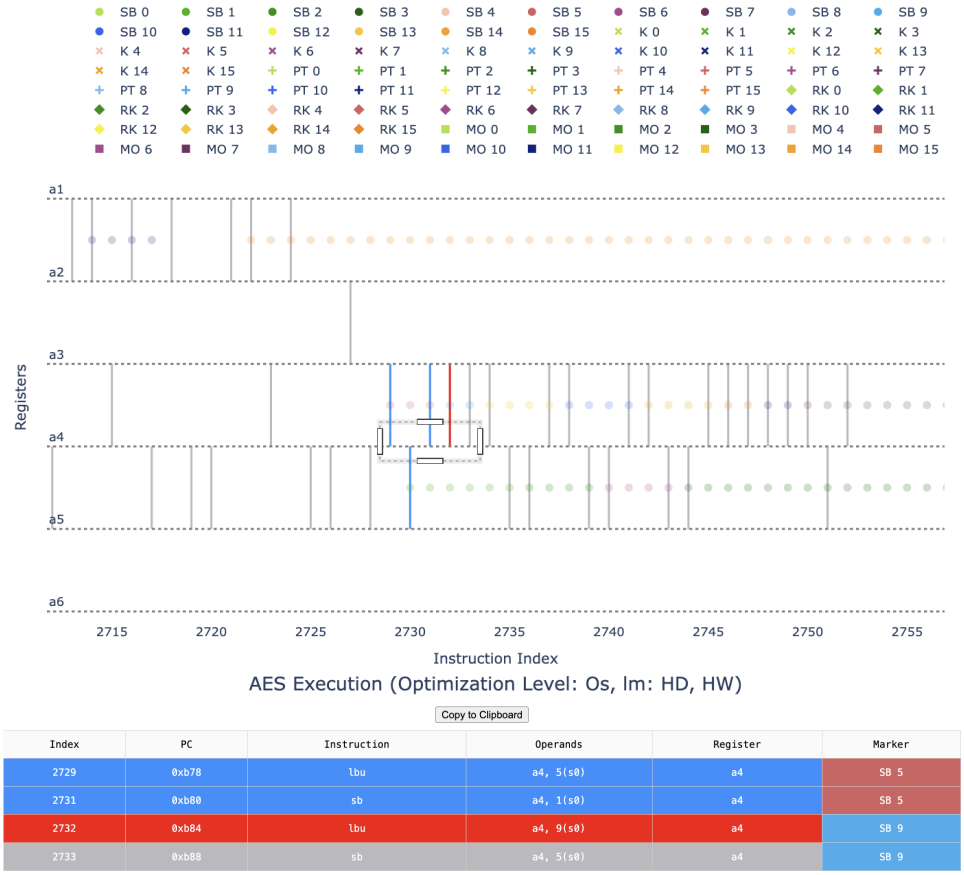
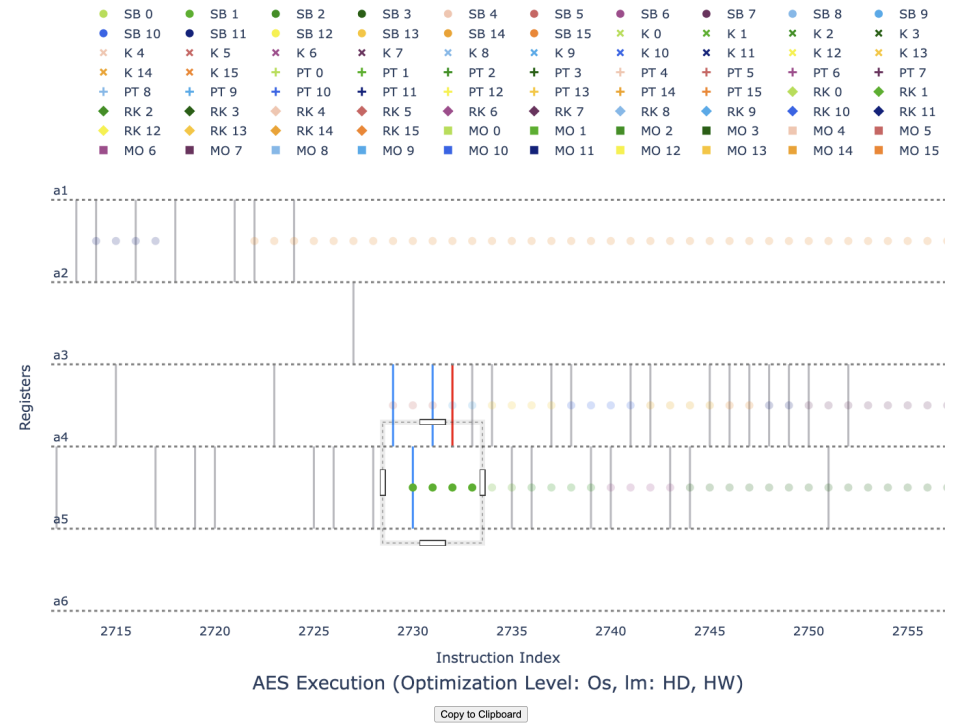


Figure 4.14: Box Select on one register line

Table 4.4 provides a detailed interaction matrix showing how the primary features of the framework assist and complement each other. Some combinations of features work as follows: For instance, using the slider for index range selection followed by a search by index helps narrow the search space to a relevant subset, improving search efficiency and focus. Another example is combining index opacity selection with a search by PC, which enhances visibility by highlighting the selected PC indexes while reducing the opacity of other indexes. These examples demonstrate how features complement each other to enhance usability and precision.



Index	PC	Instruction	Operands	Register	Marker
2729	0xb78	lbu	a4, 5(s0)	a4	SB 5
2730	0xb7c	lbu	a5, 1(s0)	a5	SB 1
2731	0xb80	sb	a4, 1(s0)	a4	SB 5
2732	0xb84	lbu	a4, 9(s0)	a4	SB 9
2733	0xb88	sb	a4, 5(s0)	a4	SB 9

Figure 4.15: Box Select on multiple register lines

Features	Index Range Selection	Index Opacity Selection	Markers Opacity Selection	Search by Index	Search by PC
Index Range Selection	-	Clearer visualization through trace opacity control	Clearer visualization through marker opacity control	Perform search precision	Perform display precision
Index Opacity Selection	Clearer visualization through trace opacity control	-	Enhances marker clarity when overlapping with a trace and vice versa	Highlights searched indexes for better visibility and also by reducing the trace opacity	Highlights searched PCs for better visibility and also by reducing the trace opacity
Markers Opacity Selection	Clearer visualization through trace opacity control	Enhances marker clarity when overlapping with trace and vice versa	-	Highlights searched indexes for better visibility and also by reducing the marker opacity	Highlights searched PCs for better visibility and also by reducing the marker opacity
Search by Index	Perform search precision	Highlights searched indexes for better visibility and also by reducing the trace opacity	Highlights searched indexes for better visibility and also by reducing the marker opacity	-	Combines with PC-based search to refine results
Search by PC	Perform display precision	Highlights searched PCs for better visibility and also by reducing the trace opacity	Highlights searched PCs for better visibility and also by reducing the marker opacity	Combines with index-based search to refine results	-

Table 4.4: Feature Interaction Matrix showing how features assist and complement each other

Chapter 5

Visualization code

This chapter presents an overview of the implementation details of the visualization framework. The visualization code supports common file formats such as `.csv` and `.pkl`, enabling flexibility in data input and compatibility with diverse computational scenarios. Extensions to additional formats like `.parquet` are possible, although preprocessing steps are required due to the limitations in modifying such formats. To aid in understanding how the code operates, a code tree (Figure 5.1) is provided, illustrating the structure and data flow of the framework.

The framework is implemented in Python3 using the Plotly package and supports interactive visualizations through Plotly. This chapter discusses the optimization strategies employed, including upgrading to Python 3.12, switching to Bokeh, utilizing the `pyarrow` engine for efficient file handling, and parallelizing computational tasks to enhance scalability. Despite these advancements, challenges remain in rendering large-scale visualizations and addressing performance bottlenecks. These observations highlight the need for further exploration and development to improve the efficiency and scalability of the visualization framework.

5.1 Supported Files

The supported files are of the `.csv` and `.pkl` format with their visualisations presented in the previous chapter representing a variety of data inputs, each serving specific purposes in computational and analytical workflows.

The code can be extended to use other types of files for example `parquet` but it would require to have all the data modifications done before hand as you cannot modify anymore as in csv and it requires a close inpection to the code which handles the visualization part as it differs from pandas and numpy.

1. **Execution trace**
   A structured log(Figure 5.2) of executed instructions and register val-

ues for debugging and analysis.
File: `trace_reference.csv`

	PC	Machine	Ins	Type	Operands	zero	ra	sp	gp	tp	...	s8	s9	s10	s11	t3	t4	t5	t6	Index	RD
0	0xfdc	130101e9	addi	ARITHMETIC	sp, sp, -0x170	0x0	0x1090	0x7ff3cec0	0x0	0x0	...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0	sp
1	0xfe0	23202117	sw	STORE	s2, 0x160(sp)	0x0	0x1090	0x7ff3cd50	0x0	0x0	...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	1	s2
2	0xfe4	13090500	mv	ARITHMETIC	s2, a0	0x0	0x1090	0x7ff3cd50	0x0	0x0	...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	2	s2
3	0xfe8	13054000	addi	ARITHMETIC	a0, zero, 4	0x0	0x1090	0x7ff3cd50	0x0	0x0	...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	3	a0
4	0xfec	23248116	sw	STORE	s0, 0x168(sp)	0x0	0x1090	0x7ff3cd50	0x0	0x0	...	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	4	s0

5 rows × 39 columns

Figure 5.2: Trace reference data frame

Used for:

- Visualizing executed assembly instructions.
- Analyzing the values of specific registers (e.g., `sp`, `ra`, `zero`).
- Tracking intermediate data generated during execution.
- Identifying and annotating potential anomalies or inefficiencies in execution.

Advantages of using the CSV type are:

- Clear tabular structure for efficient analysis and modifications.
- Lightweight and software-independent, ensuring broad compatibility.
- Easy to import and process using data analysis tools such as Python, R, or MATLAB.
- Easily extensible for integrating additional metadata.

Challenges/Limitations:

- Performance issues with very large datasets due to memory limitations.

2. **TVLA information for a given cryptographic implementation.** A tabular representation(Figure 5.3) of leakage analysis to assess side-channel vulnerabilities.
File: `TVLA_leakage_model.csv`

	Instruction Index	T-Score
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0

Figure 5.3: TVLA data frame

Used for:

- Identifying instructions with significant leakage.
- Debugging cryptographic implementations to minimize side-channel vulnerabilities.
- Visualizing leakage distribution across the instruction set.
- Prioritizing instructions for optimization or mitigation efforts.

Advantages of using this CSV format include:

- Clear tabular structure for efficient analysis and modifications.
- Lightweight and software-independent, ensuring broad compatibility.
- Easy to import and process using data analysis tools such as Python, R, or MATLAB.
- Easily extensible for integrating additional metadata.

Challenges:

- Performance issues with very large datasets due to memory limitations.

3. **Intermediate data generated during execution.**
A structured mapping (Listing 5.1) of markers to a cryptographic algorithm's operations for debugging and analysis.
File: `Cryptographic_algorithm_markers.pkl`

Listing 5.1: Mapping of markers to cryptographic operations

```
{
    a2: [2662, 2663, 2664, 2665],
    a3: [],
    a5: [2737, 2738, 2739, 2740],
    s2: [2873, 2874, 2875, ..., 2912]
}
```

Used for:

- Mapping marker indexes to specific cryptographic operations(eg. AES) for debugging and optimization.
- Analyzing register usage during algorithmic encryption(eg. AES) to ensure efficiency.
- Validating the correctness of instruction sequences in a cryptographic implementations(eg. AES).
- Providing insights into potential areas for performance improvement or side-channel mitigation.

Advantages of using this dataset as a PKL:

- Clear structure for efficient analysis.
- Compatibility with statistical tools for leakage evaluation.

Challenges:

- Requires familiarity with the cryptographic implementation(eg. AES) details for effective analysis.
- Not easy to access, read and modify compared to the CSV files.

5.2 Support for Various Cryptographic Implementations

This section describes the adaptability of the visualization framework in supporting diverse cryptographic implementations. Provided that the files discussed in Section 5.1 retain the format, the framework can be efficiently tailored to accommodate different cryptographic algorithms with minimal modifications.

5.2.1 Configuring Cryptographic Markers

The framework facilitates the specification of the number of cryptographic markers required for different types by adjusting the parameter `N` within the `build_figure()` function (Figure 5.1). For example, depending on the marker type, the number of markers required for visualization can be configured as follows:

```
if(condition):
    N = 10
else:
    N = 16
```

```
# Retrieve dot_color
dot_colors = get_dot_colors(N)
```

The function `get_dot_colors` retrieves the corresponding dot colors based on the specified value of `N`, thereby ensuring flexibility in the visual representation.

5.2.2 Defining Color Configurations

The function `get_dot_colors(N)` generates a list of colors from a predefined set, as illustrated below:

```
def get_dot_colors(N):
    base_colors = ['#afde40', '#38b000', '#008f00', '#005f00',
                   '#f6c5af', '#d36060', '#A8488F', '#70305F',
                   '#79BAEC', '#38ACEC', '#2B65EC', '#0F227E',
                   '#F7F016', '#fcc419', '#f59f00', '#F88017']

    dot_colors = []
    for i in range(N):
        dot_colors.append(base_colors[i])

    return dot_colors
```

This implementation allows users to customize the color scheme according to the number of markers required, ensuring a visually coherent representation.

5.2.3 Customizing Marker Symbols

The framework also provides the capability to modify marker symbols to align with various cryptographic implementations. Marker symbols are defined within the dictionary `marker_symbol_dict`, as shown below for AES-128:

```
marker_symbol_dict = {
    'SB': 'circle',         (SBox bytes)
    'K': 'x-thin',          (Keys Bytes)
    'PT': 'cross-thin',     (Plaintext Bytes)
    'RK': 'diamond',        (Round Keys Bytes)
    'MO': 'square',         (Mix Column Out Bytes)

    # Uncomment the two lines below for a masked AES version
    # 'MI': 'triangle-up', (Mix Column In Bytes)
    # 'M': 'star'          (Mask Bytes)
}
```

By updating the entries in this dictionary, users can seamlessly adapt the framework to accommodate different cryptographic algorithms, such as ASCON and AES, which are supported by the Archer, specifically targeting RISC-V architectures.

5.2.4 Automating the Configuration of Cryptographic Markers and Color Configurations

To enhance the robustness and scalability of the visualization framework, the process of configuring cryptographic markers and defining color configurations can be automated. This automation eliminates the need for manual intervention in specifying the number of markers and ensures seamless adaptation to varying cryptographic implementations.

5.2.4.1 Dynamic Marker Configuration

The automated process dynamically determines the number of cryptographic markers by analyzing the dataset files provided. A function, `get_highest_N_from_files()`, retrieves the maximum number of markers required by parsing filenames in the target directory, as shown below:

```
def get_highest_N_from_files(path_marker, implementation, opt_level):
    """
    Determines the highest possible number of markers in a folder.
    """
    pattern = re.compile(rf"specify file pattern.pkl")
    highest_n = 0

    # Scan through all files in the folder
    for filename in os.listdir(path_marker):
        match = pattern.match(filename)
        if match:
            # Extract the number from the filename
            num = int(match.group(1))
            highest_n = max(highest_n, num)

    # Since index is 0-based, add 1 to get the count
    return highest_n + 1
```

This function automatically determines the highest marker count based on the available files, eliminating the need for manual configuration. The retrieved value of `N` is then used to configure the cryptographic markers, ensuring flexibility across diverse implementations.

5.2.4.2 Dynamic Color Generation

To ensure that the visualization framework remains robust even when the number of markers exceeds a predefined limit, the process of defining color configurations can also be optimized. Instead of relying on a fixed list of colors, the function `get_dot_colors(N)` now generates an arbitrary number of visually distinct colors using the HSL color space:

```
import colorsys

def get_dot_colors(N):
    """
    Generates N visually distinct colors using the HSL color space.
    """
    dot_colors = []
    for i in range(N):
        hue = i / N  # Distribute hues evenly
        # High saturation and brightness
        rgb = colorsys.hsv_to_rgb(hue, 1, 0.9)
        hex_color = "#{:02x}{:02x}{:02x}".format(
            int(rgb[0] * 255), int(rgb[1] * 255), int(rgb[2] * 255)
        )
        dot_colors.append(hex_color.upper())

    return dot_colors
```

This implementation guarantees that the system can generate a sufficient number of distinct colors to accommodate any required number of markers. However, certain markers next to each other may appear visually similar to the naked eye. Despite this, the approach significantly enhances the framework's adaptability and ensures a coherent and consistent visual representation.

5.2.4.3 Automated Workflow Integration

The entire automated process is integrated into the workflow as follows:

```
# Retrieve the highest value of N from the dataset
N = get_highest_N_from_files(path_marker, implementation, opt_level)

# Generate dynamic colors for the markers
dot_colors = get_dot_colors(N)
```

This automated approach allows users to simply provide the necessary files, without the need for manual adjustments to marker counts or color

configurations. The system elegantly adapts to the files, ensuring robust operation across diverse cryptographic implementations, including ASCON and AES for RISC-V architectures.

5.3 Support for different implementations

In this section, we describe the modifications required to adapt the framework to other interactive visualization packages.

5.3.1 Switching to Plotly Express (PX)

The framework currently utilizes Plotly's `go.Figure` with `Scattergl`, which is a high-performance rendering mode optimized for large datasets. At its core, Plotly Express (PX) also employs `Scattergl` for rendering scatter plots, and thus, it is possible to seamlessly switch between these approaches. For example:

```
# Using Plotly Express (PX)
# Create a dummy DataFrame to initialize an empty PX figure
empty_df = pd.DataFrame({'x': [], 'y': []})
fig = px.scatter(empty_df, x='x', y='y')
fig.add_scatter()

# Using Plotly Graph Objects (GO)
fig = go.Figure()
fig.add_trace(go.Scattergl())
```

While both approaches support the creation of interactive figures with `Scattergl`, it is important to note the following differences and considerations:

- **Interactive Features:** When switching from `go.Figure` to `px.scatter`, the code managing interactive features may require modification. This is because while it is permissible to pass a `PX` figure into a `GO` workflow, the reverse—passing a `GO` figure into a `PX` workflow—is not supported.
- **Performance:** Based on observations, no significant performance differences were noted between `go.Figure` and `px.scatter` when rendering figures of similar complexity and size. Both approaches load figures at comparable speeds, as they utilize the same underlying rendering engine (`Scattergl`).
- **Convenience:** `PX` offers a higher-level abstraction, simplifying the creation of common chart types with minimal code. However, for advanced customization and control over individual traces or layout properties, `GO` provides greater flexibility.

Ultimately, the choice between `PX` and `GO` depends on the specific requirements of the project. For workflows that prioritize simplicity and rapid prototyping, `PX` may be more suitable. On the other hand, for scenarios that demand extensive customization or integration with existing `GO`-based components, retaining the use of `go.Figure` is recommended.

5.3.2 Switching to Bokeh

Transitioning from Plotly [10] to Bokeh [14] represents a more complex and challenging task compared to making adjustments within the Plotly ecosystem, such as switching from `go.Figure` to `px.scatter`. While Plotly and Bokeh both provide powerful tools for creating interactive visualizations, their underlying architectures, data handling mechanisms, and APIs differ significantly. As a result, the migration process involves more than simply modifying a few lines of code.

5.3.2.1 Differences Between Plotly and Bokeh

The complexity of transitioning to Bokeh arises from the following key differences:

- **Data Handling:** Plotly integrates seamlessly with `pandas` DataFrames, allowing for straightforward data manipulation and visualization with minimal configuration. Bokeh, on the other hand, relies on its own `ColumnDataSource` object to manage and pass data to plots. This means that data prepared for Plotly must be converted into the format required by Bokeh, often requiring additional preprocessing steps.
- **Rendering Models:** Plotly is a client-side rendering library, where interactive features are handled directly in the browser using JavaScript. Bokeh, while also supporting client-side rendering, excels in server-side rendering for building dashboards and applications. This difference may necessitate rethinking the application architecture if server-side interactivity is desired.
- **APIs and Syntax:** Plotly provides a high-level API with expressive and declarative commands, making it simple to create complex visualizations. Bokeh, in contrast, offers a more programmatic approach, requiring users to explicitly define plot elements such as axes, grids, and legends. This makes Bokeh highly customizable but introduces additional complexity.

5.3.2.2 Challenges in Migration

The migration process from Plotly to Bokeh involves several steps that increase its complexity:

- **Refactoring Code:** The direct substitution of Plotly functions with their Bokeh equivalents is not feasible due to the differences in API design. For example, creating a scatter plot in Plotly typically involves one line of code with `px.scatter`, whereas in Bokeh, users must define a `Figure` object, configure the `ColumnDataSource`, and explicitly specify plot elements.
- **Interactive Features:** Interactive capabilities, such as tooltips, zooming, and panning, are implemented differently in Bokeh. While Plotly provides these features out of the box, Bokeh requires explicit configuration of tools and callbacks, often involving more lines of code.
- **Customization and Layout:** Plotly simplifies the customization of layouts, annotations, and subplot configurations. In Bokeh, layout customization often involves defining multiple components (e.g., plots, widgets, and tabs) and combining them into layouts using functions like `gridplot()` or `row()`.

5.3.2.3 Recommendations

While switching to Bokeh may offer benefits such as advanced server-side interactivity and better integration with Python-based dashboards (e.g., using `Panel` or `Flask`), careful consideration should be given to whether these advantages justify the additional effort. For projects with existing Plotly-based visualizations, the following steps can help mitigate migration challenges:

1. **Evaluate Project Requirements:** Assess whether Bokeh's features align with the project's needs, especially in terms of interactivity, scalability, and integration with other tools.
2. **Incremental Migration:** Begin by recreating simpler visualizations in Bokeh to familiarize with its workflow and gradually transition more complex plots such as the ones in the previous chapter.
3. **Prepare Data Appropriately:** Convert data to Bokeh's `ColumnDataSource` format early in the process to simplify plot creation and reduce errors.

While the migration from Plotly to Bokeh is undoubtedly more complex than switching between Plotly modules, a structured and well-planned approach can help streamline the process and leverage the strengths of Bokeh effectively.

5.3.3 Deploying the Application to a Server

To enhance accessibility and reduce dependency on local execution environments, the entire codebase and associated files can be deployed to a

dedicated server such as Render[15] using DashTools[16]. This server would handle the execution of the code and store all necessary resources required for the application. By hosting the visualization component on the server, including all interactive elements such as buttons and functionalities, an HTML link can be generated and shared with users all over the world.

This approach enables users to access the visualization directly through their web browsers without requiring the code to run locally. Additionally, centralizing the application on a server ensures consistency, facilitates updates, and improves scalability, as multiple users can access the hosted application concurrently. Such a deployment paradigm is particularly beneficial for collaborative projects, educational purposes, or cases where computational resources may be limited on user devices.

5.3.4 Common Issues

In this section, we identify commonly encountered issues related to a specific context and analyze their causes and impacts. Alongside each issue, we present alternative solutions, evaluating their feasibility and effectiveness in addressing the challenges.

No Data Points Displayed When Loading

A common issue encountered during data visualization, particularly on systems utilizing Apple Silicon (e.g., Mac M1), is the failure to render data points correctly when handling a large number of traces. In this specific case, an attempt to load 40,000 traces resulted in only the registration lines being plotted, with no accompanying traces.

This issue appears to stem from the system's inability to efficiently handle the graphical processing requirements associated with rendering a high volume of 'Scattergl' traces, a Plotly module optimized for WebGL. WebGL is generally favored for its performance with large datasets; however, certain hardware or driver configurations, as exemplified by the Mac M1, can introduce compatibility challenges if it runs locally.

To address this problem, the user can either choose to render fewer traces or the rendering mode can be switched from 'Scattergl' to 'Scatter', as shown below:

```
fig.add_trace(go.Scattergl())
```

is replaced with:

```
fig.add_trace(go.Scatter())
```

The 'Scatter' mode does not leverage WebGL, but its use significantly improves compatibility in environments with constrained WebGL performance.

While this adjustment circumvents the immediate issue, it is worth noting that the ‘Scatter‘ rendering mode may exhibit decreased performance for extremely large datasets compared to ‘Scattergl‘.

Future work could investigate optimizations for rendering high-density datasets on systems with Apple Silicon, such as batching trace additions or leveraging more specialized WebGL configurations to mitigate compatibility issues.

Rendering Performance

Image lagging during data visualization occurs when rendering large datasets, leading to slow interactions and delayed responsiveness. The choice between Plotly's `Scattergl` (WebGL-based) and `Scatter` (CPU-based) modes is crucial in managing performance trade-offs.

The `Scattergl` mode, leveraging GPU acceleration, is ideal for large datasets due to its speed and efficiency. However, it may cause lag on systems with limited GPU capabilities, such as Apple Silicon, or when browser support for WebGL is suboptimal. In contrast, the `Scatter` mode, which processes graphics on the CPU, provides better compatibility and stability but may slow down with very large datasets.

5.4 Optimizations

Optimizations done correctly can significantly enhance the performance of the code, particularly in areas involving data processing and computationally intensive operations. For instance, transitioning from Python 3.9 to Python 3.12 can result in a reduction of code runtime by an average of 25% at least as we can see in Table 5.1 where 90% of the time shown is used for handling the figure(create the frame,fill the frame and then pass it) and the rest for parsing the data. However, this improvement is contingent upon the compatibility of the project's libraries, features, and implementations with the newer Python version. It is important to note that this transition does not lead to faster image rendering.

Python Version	**Optimization Level**	**Time**
3.9	`-OO` (40 000 traces)	avg. 5.4s
3.9	`-Os` (10 000 traces)	avg. 1.3s
3.12	`-OO` (40 000 traces)	avg. 4s
3.12	`-Os` (10 000 traces)	avg. 0.9s

Table 5.1: Average Run Time per Process on MAC M1

One such optimization is the use of the ‘pyarrow‘ engine in conjunction

with ‘pandas‘ for reading files more efficiently.

By replacing the standard file reading method:

```
df = pd.read_csv(execution_trace_file)
```

with the optimized method:

```
df = pd.read_csv(execution_trace_file, engine="pyarrow")
```

Substantial improvements in execution time can be achieved. It is important to note that the ‘pyarrow‘ engine is recommended exclusively due to compatibility with other backends and works well with parquet files if you choose to utilize those instead of the standard csv files. The performance improvements observed in Table 5.2 are as follows:

Python Version	Optimization Level	Standard Method(s)	Optimized Method(s)	nTimes
3.9	`-OO`(40 000 traces)	0.0625554425	0.0089676454	5-7
3.9	`-Os`(10 000 traces)	0.0210622650	0.0047182962	5-7
3.12	`-OO`(40 000 traces)	0.0598298221	0.0082747142	5-7
3.12	`-Os`(10 000 traces)	0.0199393375	0.0046910308	5-7

Table 5.2: Performance Comparison of File Reading Methods on MAC M1

Another area for optimization involves parallelizing computationally intensive sections of the code. For example, instead of processing each trace in a figure sequentially, traces can be grouped into batches and processed in parallel. This approach reduces the overhead associated with managing a large number of threads or processes, thereby improving efficiency.
The traces are divided into batches, with each batch containing multiple traces. These batches are then distributed across threads or processes using parallel processing libraries such as Python's `concurrent.futures`. Each thread or process processes a single batch of traces concurrently, as shown below:

```
from concurrent.futures import ThreadPoolExecutor
import numpy as np  # For batching

# Define the worker function
def process_batch(batch):
    for trace in batch:
        if (
            hasattr(trace, "mode")
            and hasattr(trace, "x")
            and len(trace.x) > 0
            and trace.customdata is not None
        ):
```

```
        .
        .
        .

# Divide traces into batches
batch_size = 100  # Adjust batch size as needed
batches = np.array_split(fig.data, len(fig.data) // batch_size)

# Process batches in parallel
with ThreadPoolExecutor() as executor:
    executor.map(process_batch, batches)
```

By processing batches of traces concurrently, the computational workload is distributed more effectively, resulting in significant reductions in execution time for figures with a large number of traces(not to be confused with rendering time, which stays the same regardless). This optimization not only enhances performance but also improves the scalability of the tool for handling complex and large-scale visualizations.

5.5 Limitations of the Tool

One notable limitation of the tool lies in its computational efficiency. While certain optimizations can enable faster execution of computations, the overall performance largely depends on the specific objectives and use cases for which the tool is employed.

A significant bottleneck arises from the overhead associated with passing figures between processes or components, which accounts for more than 90% of the total execution time. Additionally, when using tools such as Plotly to render figures containing a large number of traces (e.g., 10,000 traces), the rendering process itself introduces significant delays in showing the figure. Plotly's architecture, which involves transferring data and rendering instructions between the back end and the front end, struggles to handle such complex visualizations efficiently. This is particularly problematic in scenarios requiring interactive plotting with a lot of traces or frequent updates.

Efforts were made to address this bottleneck, including approaches such as processing figures in batches—e.g., dividing the workload into 10 batches of 1,000 figures each (for figures with 10,000 traces). However, this strategy has not demonstrated any substantial performance improvements. Similarly, attempts to partition the figure into smaller subsets for more granular processing have not yielded meaningful gains in efficiency rather than in code complexity.

With these observations, we want to highlight the need for further research into alternative methods or architectural solutions that could mitigate

the performance costs associated with figure handling and data transfer between the back-end and the front-end. Such advancements are essential for improving the overall scalability and effectiveness of the tool.

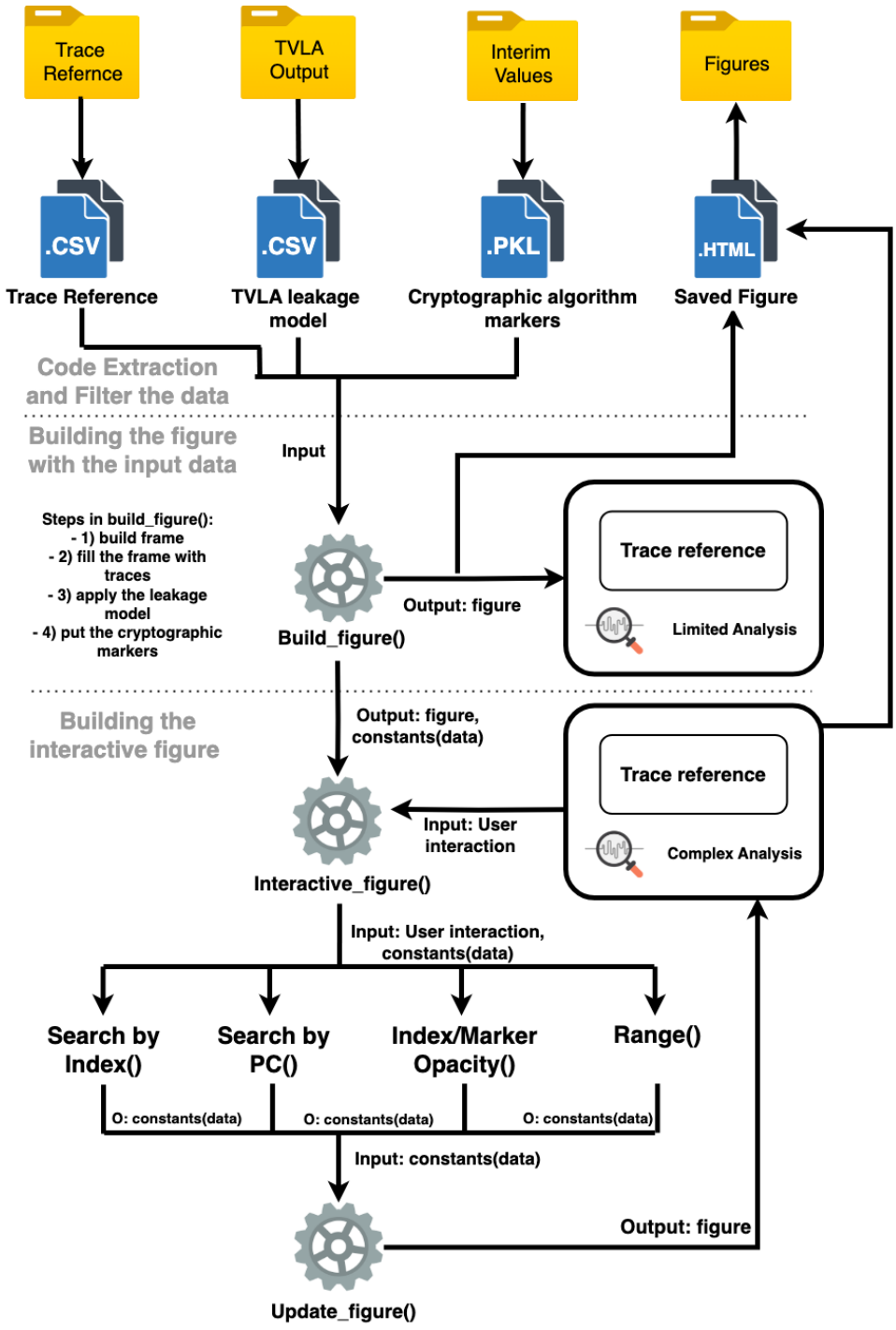


Figure 5.1: Code Structure of the Visualization Framework with required inputs and file types

Chapter 6

Conclusions

The thesis proposes a novel architecture-level interactive visualization framework for systematically analyzing side-channel vulnerabilities in RISC-V processors. The framework incorporates several user-friendly features that enable security evaluators and developers to evaluate cryptographic implementations and identify architectural vulnerabilities introduced by unintended data interactions. Features such as "Zoom", "Search by PC", and "Search by Index" equip developers to explore the entire execution efficiently and pinpoint the exact assembly instructions causing the leakage, thereby enabling the leakage analysis of large cryptographic implementations.

We demonstrate the working of the framework using a C implementation of AES-128 [17]. However, the tool can be readily used for any cryptographic implementation, given the input file structure remains intact, therefore allowing enhanced scalability and ease in portability. This framework can be integrated with the data-flow analysis component of ARCHER [1], forming an integral component in leakage root-cause analysis.

The use of Python-based tools, particularly Plotly, facilitates interactive and visually insightful representations, making the process more intuitive and informative. While Bokeh was considered during development, its refinement requirements made Plotly a more suitable choice for this framework. By offering an adaptable and extensible platform, the framework serves as a valuable resource for developing secure cryptographic implementations and also as a fundamental building block for automated visualization tools.

Bibliography

[1] A. Adhikary, A. J. B. Becerra, L. Batina, I. Buhan, D. Chatterjee, S. V. Hoek, and E. S. Gonzalez, "Archer: Architecture-level simulator for side-channel analysis in risc-v processors," *Cryptology ePrint Archive*, 2024. [Online]. Available: https://eprint.iacr.org/2024/1866

[2] F. E. Potestad-Ordóñez, E. Tena-Sánchez, A. J. Acosta-Jiménez, C. J. Jiménez-Fernández, and R. Chaves, "Hardware countermeasures benchmarking against fault attacks," *Applied Sciences*, vol. 12, no. 5, p. 2443, 2022. [Online]. Available: https://doi.org/10.3390/app12052443

[3] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, ser. Information Security and Cryptography. Springer, 2002.

[4] E. Borin, *An Introduction to Assembly Programming with RISC-V*, 2021. [Online]. Available: https://riscv-programming.org/book/riscv-book.html

[5] F.-X. Standaert, "Introduction to side-channel attacks," in *Secure Integrated Circuits and Systems*. Springer, 2010.

[6] E. O. Stefan Mangard and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[7] B. Gilbert Goodwill, J. Jaffe, P. Rohatgi *et al.*, "A testing methodology for side-channel resistance validation," in *NIST Non-Invasive Attack Testing Workshop*, vol. 7, 2011, pp. 115–136. [Online]. Available: https://www.rambus.com/wp-content/uploads/2015/08/a-testing-methodology-for-side-channel-resistance-validation.pdf

[8] O. Embarak, *Data Analysis and Visualization Using Python: Analyze Data to Create Visualizations for BI Systems*. Apress, 2018.

[9] A. Lavanya, S. Sindhuja, L. Gaurav, and W. Ali, "A comprehensive review of data visualization tools: Features, strengths, and weaknesses," *International Journal of Computer Engineering in*

Research Trends, vol. 10, no. 1, pp. 10–20, 2023. [Online]. Available: https://www.ijcert.org/index.php/ijcert/article/view/825/736

[10] P. T. Inc. Plotly: Collaborative data science. [Online]. Available: https://plot.ly

[11] ——. Dash layout. [Online]. Available: https://dash.plotly.com/

[12] Kaggle. [Online]. Available: https://www.kaggle.com/

[13] YosysHQ, "Picorv32 - a size-optimized risc-v cpu," url-https://github.com/YosysHQ/picorv32.

[14] Bokeh. [Online]. Available: https://bokeh.org/

[15] Render. [Online]. Available: https://render.com/

[16] A. Hossack. Dashtools. [Online]. Available: https://github.com/andrew-hossack/dash-tools

[17] kokke, "tiny-aes-c," https://github.com/kokke/tiny-AES-c.

Appendix A

Appendix

In this section, we present in more technical detail about some concepts we used throughout the thesis which can help with the data flow analysis.

A.1 Binary to RISC-V Instruction Transition Example

In this section, we provide a detailed breakdown of how RISC-V assembly instructions are translated into their corresponding binary representations. The RISC-V ISA follows a fixed encoding format, where each instruction is represented by a 32-bit(RV32I) binary value, divided into specific fields that denote the operation type, source and destination registers, and other control parameters.

Each instruction format—such as R-type, I-type, S-type, B-type, U-type, and J-type—follows a specific bit allocation pattern that allows the processor to decode and execute operations efficiently. Understanding these formats is crucial for analyzing low-level program execution, debugging, and optimizing performance.

The following subsections demonstrate the binary encoding of various RISC-V instruction types, breaking them down field by field to provide a clear understanding of how the instruction's components map to their respective binary values.

ARITHMETIC Instruction

Example: `add x3, x1, x2`
What it means: x3 = x1 + x2.
Binary Representation: `0000000 00010 00001 000 00011 0110011`
The RISC-V **R-type** instruction format is structured as follows:

[funct7][rs2][rs1][funct3][rd][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-25	funct7	0000 000	7	Operation variant for ARITHMETIC
24-20	rs2	0 0010	5	Second source register
19-15	rs1	0000 1	5	First source register
14-12	funct3	000	3	Operation type
11-7	rd	0001 1	5	Destination register
6-0	opcode	011 0011	7	ARITHMETIC instruction

Table A.1: Binary Breakdown of the RISC-V ARITHMETIC Instruction

LOAD Instruction

Example: `lw x3, 0(x1)`
What it means: x3 = MEM[x1 + 0]
Binary Representation: `000000000000 00001 010 00011 0000011`
The RISC-V **I-type** instruction format is structured as follows:

[imm[11:0]][rs1][funct3][rd][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-20	imm[11:0]	0000 0000 0000	12	Immediate value
19-15	rs1	0000 1	5	Base register
14-12	funct3	010	3	LOAD type
11-7	rd	0001 1	5	Destination register
6-0	opcode	000 0011	7	LOAD instruction

Table A.2: Binary Breakdown of the RISC-V LOAD Instruction

STORE Instruction

Example: `sw x3, 0(x1)`
What it means: MEM[x1 + 0] = x3
Binary Representation: `0000000 00011 00001 010 00000 0100011`
The RISC-V **S-type** instruction format is structured as follows:

[imm[11:5]][rs2][rs1][funct3][imm[4:0]][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-25	imm[11:5]	0000 000	7	Immediate upper bits
24-20	rs2	0 0011	5	Source register
19-15	rs1	0000 1	5	Base register
14-12	funct3	010	3	STORE type
11-7	imm[4:0]	0000 0	5	Immediate lower bits
6-0	opcode	010 0011	7	STORE instruction

Table A.3: Binary Breakdown of the RISC-V STORE Instruction

SHIFT Instruction

Example: `sll x3, x1, x2`
What it means: x3 = x1 << x2 (bits in x1 are shifted to the left by x2)
Binary Representation: `0000000 00010 00001 001 00011 0110011`
The RISC-V **R-type** instruction format is structured as follows:

[funct7][rs2][rs1][funct3][rd][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-25	funct7	0000 000	7	SHIFT type
24-20	rs2	0 0010	5	SHIFT amount register
19-15	rs1	0000 1	5	Base register
14-12	funct3	001	3	SHIFT left logical
11-7	rd	0001 1	5	Destination register
6-0	opcode	011 0011	7	SHIFT instruction

Table A.4: Binary Breakdown of the RISC-V SHIFT Instruction

BRANCH Instruction

Example: `beq x1, x2, 8`
What it means: BRANCH to PC + 8 if x1 == x2
Binary Representation: `0001000 00010 00001 000 01000 1100011`
The RISC-V **B-type** instruction format is structured as follows:

[imm[12—10:5]][rs2][rs1][funct3][imm[4:1—11]][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-25	imm[12—10:5]	0001 000	7	Immediate value upper bits
24-20	rs2	0 0010	5	Second register
19-15	rs1	0000 1	5	First register
14-12	funct3	000	3	BRANCH condition
11-7	imm[4:1—11]	0100 0	5	Immediate value lower bits
6-0	opcode	110 0011	7	BRANCH instruction

Table A.5: Binary Breakdown of the RISC-V BRANCH Instruction

UNBRANCH Instruction

Example: `jal x3, 16`
What it means: Jump to PC + 16 and store return address in x3
Binary Representation: `00000000000000010000 00011 1101111`

The RISC-V **J-type** instruction format is structured as follows:

[imm[20—10:1—11—19:12]][rd][opcode]

Index of Bits	Field Name	Field Value	Number of Bits	Meaning
31-12	imm[20—10:1—11—19:12]	0000 0000 0000 0001 0000	20	Immediate value for jump
11-7	rd	0001 1	5	Destination register
6-0	opcode	110 1111	7	Unconditional BRANCH (JAL)

Table A.6: Binary Breakdown of the RISC-V UNBRANCH Instruction

A.2 Decoding RISC-V Instructions

In this section, we illustrate how to decode some of the RISC-V binary instructions by breaking them down into their respective fields. Each example provides an insight into how different instruction formats can be interpreted from their binary representation.

Decoding example of an I-type Instruction

Example: `lw x3, 0(x1)`
What it means: x3 = MEM[x1 + 0]
Binary Representation:

$$\underbrace{0000\ 0000\ 0000}_{\text{imm[11:0]}}\ \underbrace{0000\ 1}_{\text{rs1}}\ \underbrace{010}_{\text{funct3}}\ \underbrace{0001\ 1}_{\text{rd}}\ \underbrace{000\ 0011}_{\text{opcode}}$$

Immediate Breakdown:

$$\underbrace{000000000000}_{\text{imm[11:0]}}$$

Field name	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Bit Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary Representation	a	b	c	d	e	f	g	h	i	j	k	l																				
Rearranged Immediate	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b	c	d	e	f	g	h	i	j	k	l
Size	20 bits																				12 bits											

Table A.7: I-type Immediate Field Breakdown

The Table A.8 demonstrates how an I-type instruction is decoded and how each bit contributes to forming the final instruction. Similar techniques can be applied to other RISC-V instruction formats.

Decoding example of an S-type Instruction

Example: `sw x3, 0(x1)`
What it means: MEM[x1 + 0] = x3
Binary Representation:

$$\underbrace{0000\ 000}_{\text{imm[11:5]}}\ \underbrace{0\ 0011}_{\text{rs2}}\ \underbrace{0000\ 1}_{\text{rs1}}\ \underbrace{010}_{\text{funct3}}\ \underbrace{0000\ 0}_{\text{imm[4:0]}}\ \underbrace{010\ 0011}_{\text{opcode}}$$

Immediate Breakdown:

$$\underbrace{0000000}_{\text{imm[11:5]}}\ \underbrace{00000}_{\text{imm[4:0]}}$$

Field name	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Bit Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary Representation	a	b	c	d	e	f	g														u	v	w	x	y							
Rearranged Immediate	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b	c	d	e	f	g	u	v	w	x	y
Size	20 bits																				7 bits					5 bits						

Table A.8: S-type Immediate Field Breakdown

The Table A.8 demonstrates how an S-type instruction is decoded and how each bit contributes to forming the final instruction. Similar techniques can be applied to other RISC-V instruction formats.

Decoding example of a B-type Instruction

Example: `beq x1, x2, 8`
What it means: BRANCH to PC + 8 if x1 == x2
Binary Representation:

$$\underbrace{0001\ 000}_{\text{imm[12—10:5]}}\ \underbrace{0\ 0010}_{\text{rs2}}\ \underbrace{0000\ 1}_{\text{rs1}}\ \underbrace{000}_{\text{funct3}}\ \underbrace{0100\ 0}_{\text{imm[4:1—11]}}\ \underbrace{110\ 0011}_{\text{opcode}}$$

Immediate Breakdown:

$$\underbrace{0}_{\text{imm[12]}}\ \underbrace{001000}_{\text{imm[10:5]}}\ \underbrace{0001}_{\text{imm[4:1]}}\ \underbrace{0}_{\text{imm[11]}}\ \underbrace{0}_{\text{fixed end}}$$

Field name	imm[12—10:5]							rs2					rs1					funct3			imm[4:1—11]					opcode						
Bit Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary Representation	a	b	c	d	e	f	g														u	v	w	x	y							
Rearranged Immediate	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	y	b	c	d	e	f	g	u	v	w	x	0
Size	19 bits																			1 bit	1 bit	6 bits				4 bits						1 bit

Table A.9: B-type Immediate Field Breakdown

The Table A.9 demonstrates how a B-type instruction is decoded and how each bit contributes to forming the final instruction. Similar techniques can be applied to other RISC-V instruction formats.

Decoding example of a J-type Instruction

Example: `jal x3, 16`
What it means: Jump to PC + 16 and store return address in x3
Binary Representation:

$$\underbrace{0000\ 0000\ 0000\ 0001\ 0000}_{\text{imm[20—10:1—11—19:12]}}\ \underbrace{0001}_{\text{rd}}\ \underbrace{1\ 110\ 1111}_{\text{opcode}}$$

Immediate Breakdown:

$$\underbrace{0}_{\text{imm[20]}}\ \underbrace{0000000100}_{\text{imm[10:1]}}\ \underbrace{0}_{\text{imm[11]}}\ \underbrace{00000000}_{\text{imm[19:12]}}\ \underbrace{0}_{\text{fixed end}}$$

Field name	imm[20—10:1—11—19:12]																			rd					opcode						
Bit Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary Representation	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t												
Rearranged Immediate	a	a	a	a	a	a	a	a	a	a	a	a	m	n	o	p	q	r	s	t	l	b	c	d	e	f	g	h	i	j	k	0
Size	11 bits											1 bit	8 bits								1 bit	10 bits										1 bit

Table A.10: J-type Immediate Field Breakdown

The Table A.10 demonstrates how a J-type instruction is decoded and how each bit contributes to forming the final instruction. Similar techniques can be applied to other RISC-V instruction formats.

A.3 Transformation Between RISC-V Instructions and Hexadecimal Representation

The transformation between RISC-V assembly instructions and their hexadecimal representation involves encoding and decoding processes based on the RISC-V instruction set architecture (ISA). RISC-V employs a fixed-length instruction format, typically 32 bits(RV32I), which simplifies the conversion process. Understanding this transformation is crucial for low-level debugging, compiler design, and embedded system development using the RISC-V ISA.

Encoding RISC-V Instructions to Hexadecimal

The process of converting RISC-V assembly instructions to hexadecimal can be structured in the following steps:

1. Identify the type and format of the instruction (R-type, I-type, S-type, etc.).

2. Find the binary representation of the command by determining the values of opcode, registers, immediate values, and function codes.

3. Section the binary representation into groups of 4 bits.

4. Calculate the hexadecimal value of each group.

5. Combine the hexadecimal values to form the final hexadecimal representation.

Example: “add x5, x6, x7“ (an R-type instruction):
Format: [funct7][rs2][rs1][funct3][rd][opcode]

- Opcode: 0110011

- Funct3: 000

- Funct7: 0000000

- Destination register (rd): x5 = 00101

- Source register 1 (rs1): x6 = 00110

- Source register 2 (rs2): x7 = 00111

The assembled binary instruction:
0000000 00111 00110 000 00101 0110011 = 0x007302B3
The conversion follows this breakdown:

```
Binary Value:   0000000 00111 00110 000 00101 0110011
Grouped:        0000 | 0000 | 0111 | 0011 | 0000 | 0010 | 1011 | 0011
Hexadecimal:    0    | 0    | 7    | 3    | 0    | 2    | B    | 3
Final Hex:      0x007302B3
```

Decoding Hexadecimal to RISC-V Instructions

The reverse process involves translating a hexadecimal value back into a human-readable RISC-V instruction by following these steps:

1. Section the binary string into one character for each section.

2. Decode each section to determine its binary value(4 digits).

3. Identify the instruction format based on the opcode.

4. Map the binary values to assembly components (registers, operation codes, etc.).