Zero-Knowledge Proofs for Secure and Private Voting Systems.

Individual Component - Real-Time Dynamic Taint Tracking with Web-Based Visualization.

Project ID – IT21318252

Final Project Thesis

Student ID: IT21318252

Name: Rafeek A.M

Supervisor: Mr. Kavinga Yapa

B.Sc. (Hons) Degree in Information Technology Specializing in Cyber Security

Department of Information technology

Sri Lanka Institute of Information technology

April 2025

Table of Contents

Introduction

The computer security of data processing machinery has gained paramount importance as one moves along the road of digitalism. Inputs by users constitute a major security factor as the interactivities and acts upon various systems become timely, dynamic, interactive, and primarily online. Perhaps the most unrecognized threats occur not by system break-ins but by wrongful input injections and indirect data flows, which may either modify a program's behavior surreptitiously or embarrass it in leaking sensitive information [1].

The thesis is addressing the problem of spot tracking by focusing on dynamic taint analysis during runtime to determine if untrusted or suspicious data flows through a system without proper validation or sanitization. Traditionally, taint-tracking mechanisms have been implemented at the language or compiler levels, but their application to systems at the application level, especially web-facing and real-time ones, is a very nascent research area [2].

To overcome this problem, we propose a system that couples metadata-based taint tracking with a real-time visualization dashboard. The system monitors user inputs from a command-line interface, traces flows of that data through critical stages in a given CFG, and outputs taint tracking results into a Flask-based web dashboard. Each input is judged for maliciousness based on patterns; behaviors define them respectively as either valid or malicious. This allows a clear inspection and the provision of immediate security feedback for system analysts.

Unlike many typical taint tracking implementations that occasionally place emphasis on storing full payloads and analyzing them later, our methodology focuses more on scalable, lightweight tracking [3]. In addition, suspicious behavior is flagged by our system beforehand, so the normal reactive analysis of logs and crashes remains rare. It also renders visualization of the control path of the program while the data is being processed-aiding observability and response time.

To address these issues, this thesis presents the design and implementation of a taint-tracking-based system tailored for secure vote handling and anomaly detection. Leveraging techniques such as control flow graph (CFG) visualization, metadata-based taint tracking, and a web-based GUI, the system allows for both real-time monitoring and post-analysis of input validation events. The implementation focuses on enhancing usability and auditability while preserving robust detection of potentially malicious patterns.

This introductory chapter provides the foundation for research and system design covered in the thesis. It provides the justification for taint tracking in modern software systems and describes the novel combination of CLI-based input consumption, backend taint tracking, and frontend visualization. In the

following chapters, we cover related work, explore the design and implementation of our proposed system, provide experimental results, and analyze its impact and scalability in real-world environments.

Literature Review

Effective input validation together with tracking malicious data remain essential security measures for modern cybersecurity within systems that accept user input. This chapter presents a detailed examination of the core methods and advanced development in taint analysis, input sanitization, control flow modeling, and security dashboards which establish the theoretical framework for the thesis system development.

Taint Analysis: Concepts and Classifications

Taint analysis involves techniques for tracing how untrusted input flows through a system. All data that originates from an untrusted source is marked as "tainted," and any time it flows to an operation that is considered sensitive, the analysis is recorded. There are two types of taint analysis:

- **Static Taint Analysis:** Static taint analysis is performed at compile-time, and involves analyzing potential data flows based on control and data flow graphs. A program is never actually executed in this case. Static taint analysis covers a broad scope, but can often suffer from high false positive rates, primarily due to over-approximation.
- **Dynamic Taint Analysis:** Static taint analysis is performed at compile-time, and involves analyzing potential data flows based on control and data flow graphs. A program is never actually executed in this case. Static taint analysis covers a broad scope, but can often suffer from high false positive rates, primarily due to over-approximation [4].

TaintDroid and Panorama improved on this notion of dynamic taint analysis by developing strong tracking capabilities for mobile and desktop respectively [3] [5]. However, because of the system-level integration needed and the consumption of resources (CPU and Memory), both systems are not ideal for light-weight or modular systems.

Metadata-Based Taint Tracking

Traditional taint tracking often involves the use of shadow memory or variable annotation which can cause the use of additional memory and make the system more complex, while metadata-based taint tracking overcomes some of these limitations by keeping the taint state externally, making it more flexible and more efficient. Clause et al. showed that this isn't just a theoretical concept, when they introduced Dytan - a dynamic tainting framework that separates data flow analysis from the execution environment [2].

In our system, we want to achieve this by having a highly scalable and modular architecture, where the external metadata is how we track the input taint state. This allows us to integrate taint tracking into web applications much more easily, improves maintainability of the application, and allows us to change the taint tracking logic as we need without having to change the core application logic.

Validation and Sanitization Techniques

Input validation is a key step in counteracting injection attacks through Cross-Site Scripting (XSS) vulnerabilities and SQL injection attacks. Validation methods are typically used in the following forms::

- **Blacklist Filtering:** Where we drop any inputs matching the well-known dangerous patterns (e.g., <script>, DROP TABLE).
- **Whitelist Filtering:** Only accepts inputs that fit some strict criteria defined to be valid (e.g., alphanumeric only)

Although blacklist filtering is more flexible, it is more likely to be evaded. While whitelist filtering is perhaps more secure, it may come at the cost of usability. Newer systems, therefore, often utilize hybrid systems that leverage context-aware filtering along with pattern matching[6].

Shashidhar described the effectiveness of mixing machine learning models with taint analysis to classify XSS attacks, demonstrating that dynamic taint tracking plus intelligent filtering combined can lead to much better detection of attacks. Our application makes use of a pattern matching method utilizing improved regular expressions to sanitize user inputs in real-time [7].

Control Flow Graphs in Security Monitoring

Control Flow Graphs (CFGs) are used to visualize the path of execution within a program. In threats such as user input, CFGs can be used to:

- Identify disallowed transitions, and aberrant anomalies.
- Track the propagation of user-provided input to sensitive operations.
- Provide visibility to security analysts/consultants when auditing software.

Kruegel et al. mentioned that CFGs also assist in decompiling obfuscated binaries with the aim of discovering security holes or weaknesses [8]. Our application simplifies the concept to one of a four stage static CFG—Input, Validation, Processing, and Storage—to monitor the progress of systematic taint knowledge as it learned across these four stages. As web applications tend to be structured this way, abstraction works quite well for the purpose of facilitating real-time tracking of control over taint knowledge to mitigate anomalous activity.

Zhu and Yan suggested a taint analysis-based approach for detecting web vulnerabilities using CFGs and context-based reasoning [9]. They acknowledged the utility of visualization for behavioral analysis, which is fundamental to our system's use of matplotlib to visualize streaming CFG snapshots for each input.

Real-Time Dashboards for Security Feedback

Security dashboards support the important role of interpreting low-level detection events to human-readable summary information (dashboard):

- Centralized visibility on taint tracking information.
- Quick access to identify abnormal inputs.
- Encouragement for system analysts to turn insights into action through visualization.

While enterprise-scale monitoring platforms (e.g. Splunk and ELK stack) will dominate the discussion in this space, the downside is that they are heavy on setup and require integrated systems. To provide a lightweight consumer-grade monitoring, our system components a Flask-based web dashboard that provides logging of user inputs, a taint status display, and CFG visualizations tied to user inputs to maintain observability without overwhelming users.

Summary

This chapter surveyed key areas of research underpinning our system: dynamic and metadata-based taint analysis, input validation, CFG-based visualization, and real-time dashboards. While previous studies have laid robust theoretical and technical foundations, our work differentiates itself through its minimalist, modular architecture and its novel use of CLI-to-dashboard integration for tracking data flow in real-time user-facing systems.

Methodology

This chapter offers a detailed description of the methodology that was followed for the development of the proposed dynamic taint tracking system. The purpose of this research was to design and implement a system to detect, classify and visualize tainted (or potentially malicious) user inputs in real time. The proposed design for the taint tracking system incorporates a combination of: metadata-based taint analysis, control flow modeling of the tainted user input, and web-based visual analytics through a simple, lightweight dashboard interface.

The discussed methods and analysis were divided into the different phases that contributed to the system as a whole: User Input, Taint Detection and Data Propagation Analysis, Classification of User Inputs, Control Flow Analysis to Visualize Metadata Changes, and Dashboarding. A relevant feature of this approach is in its modular system design which ultimately decouples the underlying data tracking service from the visualization layer of the system. The key advantage here is the ability to react and respond in real-time while also allowing for scaling and maintaining the reliability of the dynamic taint tracking system.

The taint tracking knowledge is fundamentally based on the metadata annotation, or tagging, of every piece of user input in a separate metadata store. Rather than adding taint status in the Application code's variables or memory spaces, the taint status was managed with separate metadata tracking. This offered a lightweight, scalable approach to taint propagation tracking mechanisms with minimal overhead and disruption to the application logic.

Additionally, a simplified Control Flow Graph (CFG) captured in four main steps of processing – Input, Validation, Processing, and Storage – represents the flow for each user input. More than a visual model for tracking data flow, CFGs highlight the different state values for potential anomalies within the validation process. When an input is submitted through a Command-Line Interface (CLI), the system will ingest the input, log it, assess tainting properties and will provide a visual rendering of data flow on the CFG to represent the ingestion process. The logs are ingested again on a Flask-based web dashboard for

real-time operational support and the CFG visual can be shared for later viewing when a user interacts with the Web application.

There are two basic categories of inputs: valid and malicious. Valid or malicious inputs contribute to the overall system response and visualization in the dashboard.

The general methodology is focus on a few key research principles:

- real-time user feedback and system analyst.
- observability for taint status and control flow behavior.
- visibility and traceability of system adaptations based on logging and visualizations.
- non-invasive system adaptations by using metadata and modularity to separate taint logic from core functionality.

In the following pages, we will examine all system aspects including, but not limited to, architecture, implementation design, data flow, validation processes, control flow design, and user interfaces. Each aspect uses supporting diagrams and flowcharts to illustrate internal logic, decision making, and communication between system components.

This chapter establishes the technical underpinnings upon which the system evaluation, results, and proposed improvements are developed. Finally, this chapter serves as an indication of how this work addresses certain limitations associated with traditional taint tracking systems, featured in Chapter 2, by providing a lightweight, flexible, and interactive way of monitoring user input and identifying vulnerabilities in user-facing systems.

System Design and Architectural Overview

The proposed system utilizes a CLI-to-Dashboard architecture, which is a custom built and intentional architecture of command-line interactions and modern, web-based visualization tools. This architecture enables fluid interaction between backend input processing and front-end data visualization. The taint tracking engine is central to this system design, as this component is responsible for dynamically identifying, tagging, and characterizing user-supplied inputs, based on their syntactic and semantic shapes. Once the taint status of all the inputs is known, the system produces a visual representation of the data flow, using a Control Flow Graph (CFG) derived from the taint status, while rendering the visual in a web dashboard powered by flask.

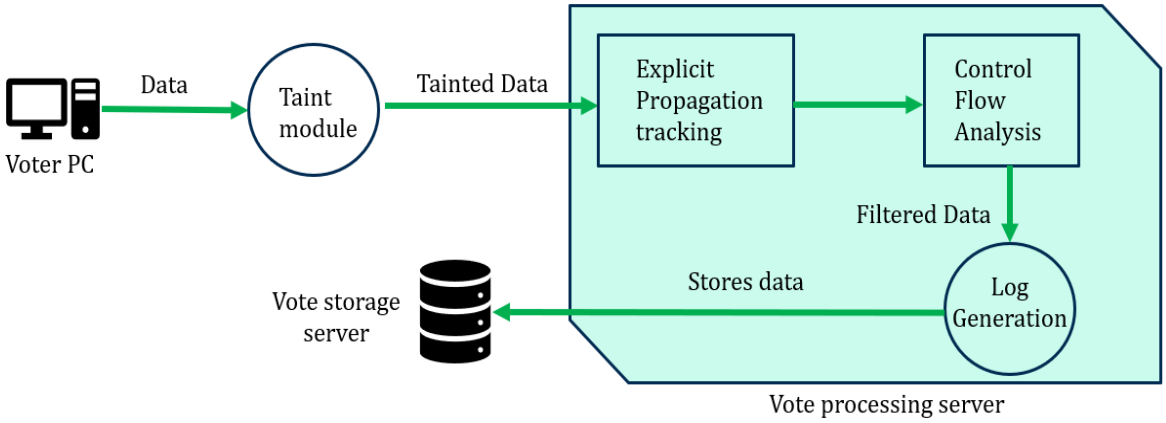

Figure 1

The system works in a modular fashion, emphasizing a separation of roles and responsibilities for taint detection, input classification, data logging, visualization, and user engagement layers. This modularity provides a number of important advantages:

- **Real-time Responsiveness** Inputs are processed in real-time, flagged in an instant, and logs are readily available for review.
- **Extensibility**: Standalone modules, such as the validation engine or visual renderer, could be updated or optimized without affecting other elements.
- **Minimal Overhead**: The proposed system is not instrumenting the actual application logic. Taint tracking is done externally with metadata, which minimizes performance interference from taint tracking.

This architectural strategy is consistent with the overarching objectives of applied cybersecurity research, which seeks to provide useful and scalable strategies that could be adopted into the operational workflow, while requiring no deep system-level integration.

The architecture, as a design strategy, is consistent with an applied research methodology by integrating theoretical models of dynamic taint analysis, metadata-based tagging, and control flow modeling into a workable system that can be evaluated and used in authentic environments. More specifically::

- Dynamic Taint Analysis is used for the evaluation of user inputs at runtime, focusing on the detections of user inputs that are associated with known attack vectors (e.g., XSS, SQLi) or implicit taint (e.g., sensitive keywords like "admin", "token", ...).

- Metadata-based tagging is used instead of shadow memory or in-band tagging, which supported a lightweight process by enabling us to tag every user input as it occurs without modifying the application variables.
- Control Flow Tracking occurs through an abridged CFG to show the pathway of each input through the critical application phases, which were Input Reception, Validation, Processing, and Storage.

Using this methodology, the system is used to support end-to-end monitoring of the user interaction, from ingestion through a CLI to visual presentation on a dashboard. The logs are persistently stored and classified as either Valid or Malicious based on how the user input is evaluated and propagated.

The Flask dashboard is more than just a static visualization – it's an interactive monitoring interface where analysts can review user input history, check the taint classification, and, with the click of a button, generate a CFG visualization for any log entry. This enables analysis, forensic auditing, and educational demonstration of taint behavior in program execution paths.

This system is designed to successfully translate research findings into a working prototype, aiding in security analysis of real applications while also demonstrating certain important concepts that are established in taint tracking literature. This means that there exists combinations and compromises between theoretical precision and engineering capabilities that result in a research artifact (or piece of research output) as well as a monitoring product that can be deployed.

Metadata-Based Taint Tracking

A primary characteristic of the proposed system is the use of metadata-based taint tracking, which offers great efficiency and modularity, making the taint analysis engine simpler to implement. The common taint tracking approaches, especially in low-level systems or run-time instrumentation in a language, pollute the program's memory or variables directly by embedding taint markers with the data itself. These embedded methods will often use structures like shadow memory, where an entry that tracks the taint status mirrors each byte or memory address containing taint. Embedded taint tracking is useful at this fine granularity, but it can limit a system in terms of memory usage, performance overhead, and implementation complexity for user-facing or real-time systems.

This system overcomes the limitations of the embedded methods described above by utilizing a non-intrusive metadata-based approach to separate data from its taint status. Instead of applying taint labels directly to memory objects or variables, the taint status will be separated from the data and tracked in a

metadata-based structure, similar to a dictionary. When a user submits input through the command line interface (CLI) the input is immediately hashed to derive a unique identifier called the data_id. The hashed value acts as an identifier in the metadata dictionary containing taint-related information, including the source of taint and its current status.

There are a few major technical and architectural advantages of this design for modular systems for modern software systems:

Lightweight, Memory-Efficient Tracking

The system can also eliminate the complexity of shadow memory and does not need to instrument every variable or function in the application because it is able to externalize taint information. The system is less complex internally, making it easier to use and more efficient; it's especially beneficial for scenarios with limited memory and processing power. When a taint metadata entry is created, it will be for something that we identified as needing to be tracked, and benign, untraceable values will not take up space in the taint metadata, as will non-critical values.

This also satisfies our desire for minimal trust by default — all user inputs are untrusted by default until proven trusted, so taints all User Inputs until it's proven it should not be. Unlike many low-level instrumentation programs such as Valgrind or TaintDroid, that often rely on hooks based on our platform or kernel modifications, we can understand a taint as metadata in our user space. The taint tracing is the only component that operates on the application owner's behalf.

Decoupled Taint Engine for Modularity and Extensibility

The architecture separates application core functionality and taint tracking functionality. The taint engine is hermetically sealed so it will only operate at specific time intervals and only accesses application data through a defined interface (mainly, the taint metadata dictionary). This feature will also ease future upgrades, such as integrating:

- Additional taint sources (e.g., file uploads, API calls)
- Taint sinks (e.g., database writes, system commands)
- More significant propagation rules (e.g., tainting functions, conditionals)

In addition, the metadata, as it is being stored in an abstracted, centralized layout, can be easily persisted over sessions, audited for historical purposes, or exported externally to use with other tools (e.g., SIEMs).

Real-Time Integration with Dashboard and Logging Subsystems

The system's metadata-based structure enables real-time introspection of the system state. Since each taint record is associated with a unique id and is stored in a global dictionary, it is trivially easy to expose this state to external entities like a Flask-based web dashboard or a JSON logging engine. Each input taint state can be retrieved and displayed on demand, and system administrators can walk through each stage of processing (Input → Validation → Processing → Storage) to see where tainted inputs flow.

These dynamic Control Flow Graphs (CFGs), that can be constructed in real-time based on metadata status, also improve the system's interpretability. The dashboard did not simply render raw execution traces or log lines, instead it mapped the system's metadata into less abstract visuals to represent the progression of each input through the stages, complete with color-coded taint status. These visual abstractions distinguish the information system from traditional taint-tracking systems, and help non-technical users (e.g., auditors, security analysts) comprehend the taint tracking system.

Lifecycle Management and Taint Status Transitions

The taint lifecycle for the system is simple, and follows a conservative model:

1. **Initial Tainting:** All inputs are considered untrusted when ingested, thus they are marked as tainted.
2. **Validation Check:** The input goes through a sanitation and pattern-matching process, where known malicious patterns (e.g., SQL injection, JavaScript tags) or special characters are inserted and detected.
3. **Clearing Taint:** If the input passes the sanitation/validation, the taint is removed from the metadata dictionary, and now trusted.
4. **Rejecting:** If the input has explicit taint keywords it will still be tainted. Malicious inputs are noted and persisted with comprehensive logs including dates and timestamps, rejection reasons, and status codes assigned (e.g., "Malicious").

This lifecycle assures taint information is accurate (and timely) and only bounded to the current state of the input in question (we protect against stale taint propagation and false negatives).

Control Flow Graph Construction

Our Control Flow Graph (CFG) in the system is not simply a static representation of program logic, it is a dynamic, security-oriented representation that visualizes untrusted inputs and its propagation through relevant stages of the application. We elaborate on how to build and use the CFG below, and refer to important code parts, if applicable.

CFG Design Philosophy

Conventional CFGs in program analysis capture all possible execution paths including the analysis of loops, conditionals, and function calls. However, when it comes to taint tracking in the context of security monitoring, it is unnecessary to include all possible paths (and therefore an abundance of execution paths), nor is it efficient because there is simply too much data to analyze..

Our CFG abstracts the lower-level detail of CFGs by presenting only four high-level stages that warrant attention whenever taint is expected to be propagated at these high-level stages:

1. Receiving input
2. Verifying input
3. Processing input
4. Storing input

What we end up with is an abstract representation that has notable advantages:

- Clarity – Security analysts only see the taint flow patterns that are relevant to them without any low-level code paths.
- Performance – They do not need to instrument every fork/loop of the program.
- Desire – These representations will provide early visual warnings for malicious input.

Step-by-Step CFG Construction

Node Creation (Per Input)

Each time a user submits input via the CLI, the system:

1. Creates a unique data_id (hash of the input for metadata).
2. Builds the entry node of the CFG (Input Reception) with properties:

```
cfg_node = {
   "id": data_id,
   "stage": "Input",
   "taint_status": "Tainted",  # Default assumption
   "content": user_input,
   "timestamp": datetime.now()
}
```

3. The similarly passes the input to the validation engine, that performs regex and keyword checks:

```
def validate_input(input_str):
   if re.search(r"<script>|DROP TABLE", input_str):
      return "Malicious"
   else:
      return "Valid"
```

```
# Control Flow Graph (CFG) Setup
cfg = nx.DiGraph()
cfg.add_node("Input", description="User input is received")
cfg.add_node("Validation", description="Input is validated")
cfg.add_node("Processing", description="Validated input is processed")
cfg.add_node("Storage", description="Processed data is stored")
```

Figure 2

- If Malicious, then the CFG will change status to "Blocked" edge from there on.
- If Valid, then taint will be cleared and the CFG will proceed.

Edge Formation (Taint-Aware Transitions)

```
cfg.add_edge("Input", "Validation", condition="Input received")
cfg.add_edge("Validation", "Processing", condition="Input validated")
cfg.add_edge("Processing", "Storage", condition="Input processed")
```

Figure 3

Edges between nodes encode security decisions. For example:

- Input → Validation
    - *If malicious*:

      cfg_edge = {

        "source": "Input",

        "target": "Validation",

        "action": "Blocked",

        "reason": "XSS detected",

        "color": "red"

      }
    - *If valid*:

      cfg_edge["action"] = "Sanitized"

      cfg_edge["color"] = "green"
- Validation → Processing (only for valid inputs)
    - Taint can propagate when there is input influencing other variables (concatenation).

Real-Time Visualization (NetworkX + Matplotlib)

The CFG is generated dynamically by going through the following processes:

1. Networkx for graph construction:

   import networkx as nx

```
G = nx.DiGraph()
G.add_node("Input", status="Tainted")
G.add_node("Validation", status="Malicious")
G.add_edge("Input", "Validation", label="Blocked", color="red")
```

- Matplotlib for interactive plotting use:

```
pos = nx.spring_layout(G)  # Layout algorithm
edge_colors = [G[u][v]['color'] for u,v in G.edges()]
nx.draw(G, pos, with_labels=True, edge_color=edge_colors)
plt.show()  # Renders in Flask dashboard
```

- Node color: Red (malicious), green (clean).
- Edge labels of the reason an input was blocked or allowed.

Key Technical Innovations

Lightweight Metadata-Driven CFGs

Unlike traditional CFGs that require full program instrumentation, our approach:

- Only tracks taint-relevant stages (Input → Validation → Processing → Storage).
- Stores metadata separately (no shadow memory or code modification).
- Generates graphs on-demand (no persistent overhead).

Code Example (Metadata Store):

```
taint_metadata = {
   "abc123": {  # data_id
      "stage": "Validation",
      "status": "Malicious",
      "reason": "SQLi attempt"
```

```
    }
}
```

Interactive Forensics in Dashboard

The Flask dashboard gives the analyst the following options::

- Click on the nodes to see raw input and taint reasons.
- Compare multiple CFGs to see attack patterns.
- Export graphs for reporting.

Flask Endpoint for CFG Retrieval:

```python
# Route to Generate CFG for a Specific Log Entry
@app.route("/generate_cfg", methods=["POST"])
def generate_cfg():
    """Generate CFG for a specific log entry."""
    log_index = int(request.form.get("log_index"))
    logs = []
    with open(LOG_FILE, "r") as logfile:
        logs = [json.loads(line) for line in logfile]

    if 0 <= log_index < len(logs):
        generate_cfg_for_log(logs[log_index])
        return send_file(os.path.join(STATIC_DIR, "cfg.png"), mimetype="image/png")
    else:
        return "Invalid log index", 400
```

Figure 4

Practical Example: SQL Injection Attempt

Input: "SELECT * FROM users; DROP TABLE logs--"

1. **Input Node**
    - Created with status="Tainted".
2. **Validation Node**

- Regex detects DROP TABLE → status="Malicious".

3. **CFG Output**
    - A **two-node graph** with a red "Blocked" edge labeled "SQLi detected".

Rendered CFG:

[Input (Tainted)] --(Blocked: SQLi)--> [Validation (Malicious)]

Why This Approach Works

This new approach to CFG construction has many benefits when compared to traditional CFG construction. It is especially effective in the security-focused taint tracking space. Traditional CFGs are almost always presented as instruction level CFGs which tend to produce complex graphs that are difficult to interpret. In contrast, the security-friendly CFG presented here is a simplified model that represents stages of execution, and is only concerned with the important taint propagation stages (Input, Validation, Processing, Storage). With this simplified graph, it is much easier to see what is happening since the high detail clutter is reduced. Traditional CFGs can easily have hundreds of nodes and edges for a single execution path while the security-friendly model uses at most four graphs in a maximum of four of those same stages so is immediately interpretable for security analysts.

Efficiency is another especially good aspect of our approach. Traditional CFGs typically require extensive instrumentation to work correctly, which tends to have a very high runtime overhead, as all branches and loops have to be tracked. Our metadata driven CFG does not require a lot of deep code instrumentation, as with using simple taint-tracking issues that occur entirely external to the code the performance impact is minimal. Because of this improved efficiency we can run our analysis in real time without being too concerned with introducing lag into the system.

More importantly, our CFG is built for security instead of a general program analysis purpose. The classic CFG builds mappings of every single possible execution (control-flow), but our CFG maps the only CFG relevant to security and explains how the tainted inputs will be exploited through vulnerable elements of an application. In a practical context, `our CFG removes control-flow detail that will slow security analysts down by focusing on finding attacks (e.g. SQLi and XSS attempts). We leverage the unique properties of our CFG to strike a balance between simplicity, performance, and aid of security-specific task(s) of the analyst; the result shows that we have changed the CFG from a complex program-analysis tool to a practical analyst-facing utility for real-world threat detection.

Input Validation and Logging

Good input validation for an application receiving and processing user-generated content is central to any effective security measure. Accept any user-generated input without constraints and the possibilities for attack vectors are many, from minor inconveniences to critical vulnerabilities including code injection, SQL injection, and cross site scripting (XSS). This section discusses the input validation process used in the proposed taint tracking system and justifies the design, discussing the use of regex-based pattern matching, implicit taint detection, and in-situ logging as part of an overall strategy to achieve traceability, accountability, and transparency.

Motivation for Rigorous Input Validation

User inputs are the primary point of interaction with software systems, and they also represent the most significant potential for unpredictability into these systems. Malicious attackers take advantage of this unpredictability by intentionally designing inputs that are syntactically valid but semantically harmful to evade naive validation mechanisms (ex. may be harmless to a badly designed parser, but will execute arbitrary code - with catastrophic effects in a web application - when parsed in a browser).

Researchers and security practitioners have long argued for proper input validation. Today, OWASP (Open Web Application Security Project) ranks injection flaws and improper input validation as issues number one and two respectively in the top 10 issues affecting modern applications. A taint tracking framework that is system-level must have a comprehensive input validation layer that filters, tags, or modifies inputs according to definitions or expected behavior.

Whitelist-Based and Regex-Driven Sanitization Model

The input validation module of our framework uses a hybrid approach, using a whitelist as the main strategy, but allowing for regex-style pattern matching to enforce constraints on acceptable input.

Whitelist models, by contrast to blacklist models, assume that user input should specifically include only the forms previously approved and everything else poses some risk of being suspect. This can be a much more conservative policy that can substantially decrease the attack surface, but will deny legitimate user-generated data if the policy is not appropriately designed.

To address this challenge, we defined an incrementally layered validation process that reviewed the input for the following items:

- Empty Inputs: Any input that was completely blank or whitespace was rejected and logged as invalid.
- Malicious Keywords: All input was scanned for SQL keywords (SELECT, DROP, INSERT, etc.), script blocks (), and common tokens of injections like --, #, or / *
- Special Characters: A strict regex pattern was created, which flagged any input that contained a non-alphanumeric character (excluding spaces and colons), as these can be these are often utilized for obfuscation or to bypass a filter or specific condition.

A regex that is representative of what we use in our validation engine is:

```
21
22  # Whitelist pattern (only allows these characters)
23  WHITELIST_PATTERN = re.compile(r'^[a-zA-Z0-9 :]+$')
24
```

Figure 5

This technique allows common XSS and SQL injection payloads to be detected with case insensitive scanning and careful contextual string boundaries, which provides a very high degree of fidelity.

Comprehensive Logging of Input Events

```
# Dashboard Route to Display Logs
@app.route("/")
def dashboard():
    """Render the dashboard with logged invalid attempts."""
    logs = []
    try:
        with open(LOG_FILE, "r") as logfile:
            for line in logfile:
                logs.append(json.loads(line))
    except FileNotFoundError:
        print("Log file not found. No logs to display.")
    except Exception as e:
        print(f"Error reading log file: {e}")

    return render_template("dashboard.html", logs=logs)
```

Figure 6

Logging is an essential part of providing observability, auditability, and forensic traceability in any security-sensitive application. For our application, every input event, whether validated or identified as

malicious, is saved to a central log file using structured JSON lines so that all user inputs are recorded. The log keeps track of:

- the raw user input
- validation result (Valid, Malicious)
- a reason why
- a time stamp for chronology scene observations

An example log entry might appear as:

```
{
  "timestamp": "2025-04-10 14:32:55",
  "input": "<script>alert(1)</script>",
  "is_malicious": true,
  "reason": "Possible XSS attack detected using script tag",
  "status": "Malicious"
}
```

The log files will be stored in a flat text JSON lines (logs/all_inputs.json) where each line is the representation of one user input log. This is append-only, so the entire provenance remains in-tact, and it allows for external tools to parse the incoming logs and provide further research and investigation.

Real-Time Reflection on Dashboard

As each log event gets created and saved on the backend, we immediately synchronize that event to the Flask based web dashboard for security analyst access to gain a near real-time representation of input behaviors:

- Visual summary of all inputs submitted.
- Has clickable entries that will visualize Control Flow Graphs (CFG).
- Has taint indicator colours (red is malicious, yellow is flagged, green is valid).

This integration allows analysts to visualize where tainted data spreads in the system, analyze patterns of behavior, and proactively identify and respond to emerging threats.

Benefits and Limitations

The input validation and logging solution has many advantages:

- security: blocks many common injection threats, and flags unusual input early in the process chain.
- transparency: generates an audit trail for all user submissions.
- modular: can be reconfigured to target different applications or include additional detection rules.
- user awareness: the human-in-the-loop dimension is augmented through feedback loops via the dashboard.

However, there are drawbacks, such as:

- only providing feedback on patterns that have previously been identified (often missing situationally novel or obfuscated threats).
- lack of deeper context (e.g. semantic understanding of input).

Web-Based Interaction and System Interface

Although these issues are quite common in related fields, they are significant considerations in the contemporary context of web applications and user-facing platforms, where there is pressure on to be intuitive and apparatus is generate feedback as instantaneously as possible, particularly for systems under security-sensitive operations (e.g. taint analysis).This project is more advanced than typical tools that furnish results through a CLI, processing static input because user input is included at the time of system interaction, using the Flask framework to manage user submissions, monitor data flow, and generate dynamic visual feedback.

This section further describes the architecture, design, and purposes of the web-based interaction module that will serve as the core of the taint tracking system.

Motivation for Web-Based Input and Visualization

The decision to shift from a CLI-based input model, to a complete browser-enforced interface was driven by a number of user-experience and structural reasons, including:

- Access and Interoperability: Browser-based interfaces are platform agnostic and will not task the user with any kind of local setup as CLI-based tools might.
- Instantaneous Returns: Users may provide input and see the JSONObject of the taint analysis immediately through the model of a visual cue displayed alongside the textual interaction within the same interface.
- Integration-Ready: Web interfaces are better suited for realistically deploying toolsets, especially in enterprise or academic operational contexts, since they can integrate with existing web-services, authentication modules, data-monitors.
- Richer User Interaction: By combining a submission of input with an ongoing dynamic rendering of the graphs of the Control Flow Graph (CFG) and insights from the logging, we can illustrate the interactions of input events within the system.

The interactive dashboard becomes the epicenter of both ingestion of users input and visualization of security data. Together it completes the feedback loop between users and the taint tracking engine.

Input Submission via Web Forms

The interaction model is purely a basic HTML form that is embedded in the dashboard user interface. Users are asked to submit individual inputs (such as votes, strings, and form data). Flask uses the built-in POST route to relay the individual inputs and successfully does this by following normal methods for handling form input..

Upon receiving the input, it is sent to a backend DataFlow object that:

1. Hashes the input in order to produce a deterministic identifier.
2. Sets the new input to tainted by default.
3. Uses pattern-matching rules to validate or taint the input.
4. Writes the output to a structured and useful JSON lines log file.
5. Propagates immediate visualization updates corresponding to the path of the input's CFG.

This model supports modularity and scalability. Inputs are treated as discreet units of analysis and the logic in the back-end is entirely separate from the presentation in the front-end.

Real-Time Dashboard Architecture

The dashboard is created as a real-time web interface that dynamically changes with each user interaction. It is built on Flask, HTML5, JavaScript, and jQuery and has the following features:

- Log Renderings: Each user input is rendered as a clickable log entry, with timestamp, validation result, and taint status.
- Interactive CFG Visualization: Clicking on a log entry sends a request to a Flask route, which reconstructs the CFG traversal that was completed and uses matplotlib to create a visualization of it. The graph is served back to the interface as a PNG image.
- Taint Status Updating: A background route (/taint_status) exposes the current tainted metadata dictionary for users to inspect active taints and their sources.
- Log Management: Users are able to clear logs and reset the system state using special dashboard controls, without having to restart the server or manually edit files on the backend.

By using AJAX calls through the interface, users are able to have a seamless and responsive experience, even while taint states are changing in real time, since the experience is kept as dynamic as possible by avoiding full page reloads.

Data Flow and Control Integration

The system makes use of a set Control Flow Graph (CFG) with four nodes: Input → Validation → Processing → Storage. Each user's inputs passed through these nodes and was visualized based on:

- Whether it passed scaritization, failed.
- Whether it was tainted implicitly or explicitly
- Whether taint was remove during validation

And the overall pathway was visually the graph was colored as:

- Red nodes if explicitly tainted or failed validation

- Green nodes if sanitization was successful
- Grey nodes if stages were not traversed.

The CFG is replicated per input providing a unique visualization document that is linked to each user submission as context. This mapping process allows for a significant increase in transparency and assists in behavior audit tracking.

Security Considerations

A web interface for the input process and analysis is risky. However, with the design considerations incorporated into the system, these risks are mitigated:

- Input values are validated immediately after submission, and values are sanitized even before any backend processing, analysis, or visualization takes place.
- No inputs are executed or interpreted on the server; malicious payloads are logged and safely visualized (e.g., <script>, SQL fragments).
- Separation of concerns allows the ingestion, processing, and visualization of data to be independent, thereby minimizing the possibilities for cross-contamination between these modules.
- Immutability of logs, along with metadata-based tracking, gives the ability for forensic audit without relying on runtime memory inspections.

This architectural design aims to strike a balance between usability and strong assurances of security, thus repudiating any chance of the dashboard being hazardous during any research, training, or embedding into a real system.

Benefits of Web-Centric Design

Some measurable benefits occur when switching from CLI to web UI:

- Usability: People interact with a graphical environment instead of plain inputs.
- Visual feedback: CFG diagrams and taint logs give an instant view and do not require deep technical understanding.
- Remote operation: One can submit inputs from any device connected to the server for analysis, cultivating a collaborative environment.

- Scalability: Later, the system may be extended to support session management, authentication, or integration with existing SIEM tools.

Additionally, this style of application development is more conformant with modern software development, where RESTful API, browser dashboard, and real-time logging reign over secure system architecture.

With this change to the web-based input and visualization architecture, the usability, interactivity, and extensibility of the taint tracking system have improved leaps and bounds; inputs are securely and efficiently processed via a web form whereas results are presented in an intuitively interactive dashboard, lending itself to logging, tracking, and visualization of the journey of each input through the system's CFG.

Implementation

Technologies Used

The implementation of the proposed taint tracking and visualization platform integrates a carefully selected stack of programming languages, libraries, and frameworks. These were chosen to implement the realizable goals of the system design and methodology so far considered in terms of reliability, extendibility, and suitability with respect to real-time data analysis and web-based user interaction..

Python 3.x

Python is the backend logic of the taint analysis engine and control flow modeler implementation language for. Because of the language's simplicity, readability, and access to many libraries, it is well supported and suitable for rapid development and experimentation.

- Used for: Logic handling, taint tracking, input validation, and CFG traversal.
- Why chosen: Python offers clean syntax, dynamic typing, and powerful libraries for both backend computation and visualization.

Flask Framework

The Flask framework, being a lightweight WSGI framework, serves in the creation of a system dashboard and implementation of RESTful routes.

Used for: Creating a web-based user interface, routing inputs, serving real-time visualizations, and facilitating interaction along with backend components.

Features utilized:

- Template rendering using Jinja2.
- JSON APIs for asynchronous log interaction.
- Static file serving (e.g., serving generated CFG images).

Flask is just right in granting fine control over backend routes and also easy to tie up with some Python logic without the use of a big framework like Django..

NetworkX

NetworkX is the Python library for creating, manipulating, and studying complex networks.

- -Used for: Constructing and maintaining the Control Flow Graph (CFG), representing data flow between key phases—Input, Validation, Processing, and Storage.
- -Why was NetworkX chosen: NetworkX allows to efficiently model graph-based structures, and hence it integrates very well with Matplotlib. It also supports directed graphs, which is needed to simulate the data transitions within a controlled entity..

Matplotlib

Matplotlib renders the real-time graphical view of taint propagation through CFGs.

- Use: Annotate, generate PNG images of the CFG for each input traversal, color-coded according to the taint status (e.g., sanitized, tainted, implicitly tainted).
- Why chosen: Being a versatile and widely accepted data-visualization library, it supports programmatic generation of images and hence can offer pure logic on the server side..

HTML5, CSS3, and JavaScript (with jQuery)

The dashboard interface is built using a normal set of web technologies, along with jQuery to allow dynamic page behavior and interaction with the Flask backend.

- HTML5 for laying out the dashboard and the forms.
- CSS3 to style log panels and controls and maintain eye consistency throughout.
- JavaScript (jQuery) for AJAX calls to get the logs from the backend, trigger generation of CFG images, and update the interface dynamically.
- Why chosen: These technologies make for a lightweight and snappy frontend that does not rely on complicated frontend frameworks such as React or Angular.

JSON and JSON Lines Format

Logs are held in JSON Lines format, with each input stored as a separate JSON object on a new line.

- Used for: All input data, taint status, timestamps, and validation results.
- Why chosen: JSON Lines is human-readable and also easy for the computer to process line by line without having to load every single log into memory at once..

File System and OS Libraries

The program uses Python's os and io modules to maintain logs in directories, save CFG images dynamically, and ensure that running times of creating directories remain pleasant..

Hashing Functions

Python's built-in hash() function then provides the ID mechanism to generate unique IDs per input that are subsequently used to track taint status in the metadata store.

- Why chosen: Provides them a quick hash internal id's to references user input without storing sensitive user content.

Web Browser (Client-Side Environment)

Strictly speaking, this is not part of the code base, but a modern browser has to be used to access the dashboards. We have tested the system on Chrome, Firefox, and Edge, ensuring that it operates cross-platform.

Optional Future Enhancements

A few technologies that were not considered at this stage but will be considered later on for integration:

- SQLite or something cheap: Persist logs and query on them.
- WebSocket (Flask-SocketIO): To support push-based real-time updates instead of polling.
- Docker: Can be used for containerizing the system and thereby ease deployment.

The integration of all the above will yield a big yet lightweight setup for real-time taint tracking, interactive data visualization, and easy operation through a web interface. By depending on easily accessible open-source tools and modular design, the system is left open for further extensibility for research and to be integrated into larger cybersecurity platforms.

Flask-Based Dashboard

Serving as the front-end visualization and control layer of the taint tracking system, Flask Dashboard acts as a lightweight and responsive web-based bridging front end for the interaction between analysts of the system and the backend taint analysis engine. This component thus forms the crux in presenting low-level taint tracking data in meaningful interpretations through real-time display of logs, taint status summaries, and graphical representations of control flow.

Purpose and Role

The ultimate goal of the dashboard is the following:

- It offers real-time feedback and visibility into user input flow.
- Security analysts can review each interaction and its classification.
- It affords interactive visualization of taint propagation with control flow graphs (CFGs).
- Analysts can selectively access the system's responses to each user input in a log-driven interface.

This dashboard makes the system from just a backend analysis tool into a useful and observable platform for live monitoring, auditing, and response..

Flask Framework for Web Serving

A Python microframework, Flask has been chosen because it is simple and extensible and works closely with the taint analysis logic written in Python. The Flask app publishes two main types of contents:

- Rendered HTML templates, such as `dashboard.html`, via Jinja2 for logs and interface components.
- API endpoints for AJAX calls, such as:
    1) GET / for the main dashboard.
    2) POST /generate_cfg for requesting CFG images.
    3) GET /taint_status for summaries of real-time metadata.
    4) POST /clear_logs to reset the session and clear previous logs.

This separation aids in modularity and allows asynchronous interactions between the client browser and backend logic

Dashboard Layout and Components

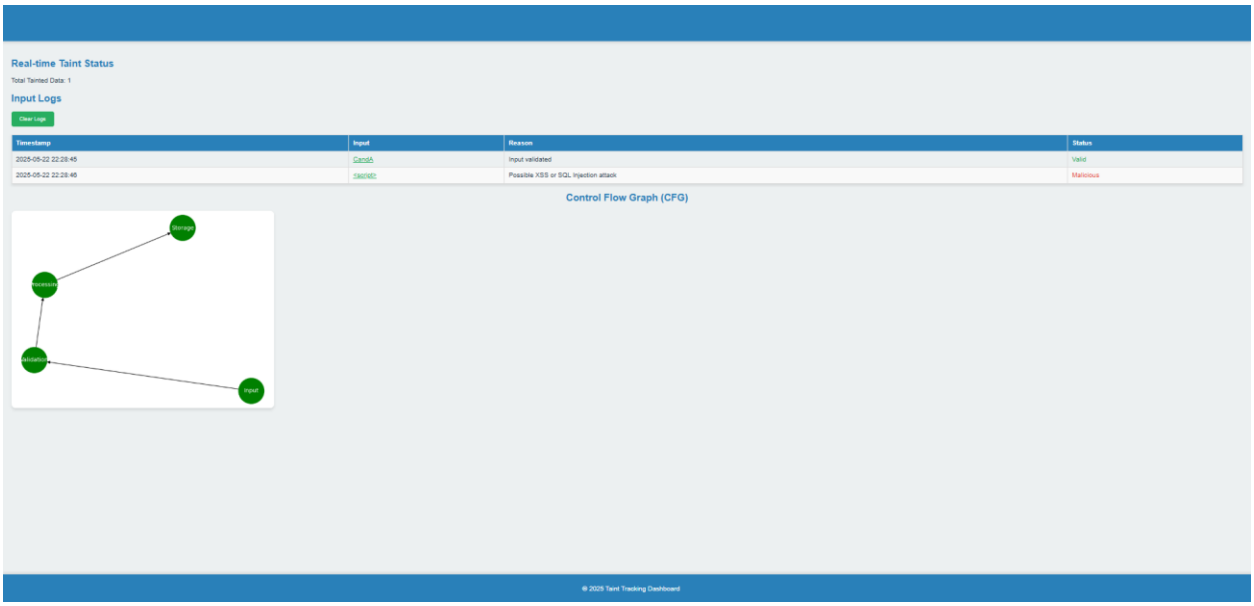

Figure 7

The dashboard consists of four main UI modules:

1. **Header Section:**
    - Displays the system name and timestamp

- Contains buttons for actions like Clear Logs.

2. **Input Logs Panel:**
   - Displays in a list format every input that has been processed by the system from the user.
   - The records exhibit input text, timestamp, classification (valid-malicious-flagged), and reason for such classification.
   - One can click on each record for further inspection and to trigger visualizations.
3. **Taint Metadata Summary Panel:**
   - Takes a summarized JSON-like representation of the tainted_metadata dictionary.
   - It lists how many inputs are still tainted and from where.
   - Updated via AJAX polling in real time.
4. **CFG Visualization Area:**
   - The moment a log record is clicked on, Flask renders a control flow graph image (cfg.png) depicting the pathway of that input through different stages of the program.
   - Nodes are color-coded as green for validated, red for tainted, yellow for flagged (such as implicit taints), and gray for skipped.

Real-Time Visualization Pipeline

Upon clicking a log:

1. A POST request is sent to /generate_cfg with the log index.
2. Flask retrieves the corresponding log entry from the JSON Lines log file.
3. Taint analysis engine rebuilds data flows for that input.
4. NetworkX and Matplotlib renders a new CFG, tagged with taint-related metadata.
5. The CFG is saved under static/cfg.png and sent on as the response.
6. The browser refreshes the <img> tag to show the latest visualization.

An interactive rendering of the same makes way for each user input to be independently contextualized. Henceforth, with visual aids definitely being provided for debugging purposes, forensic purposes, or educational purposes.

User Experience and Interface Design

To maintain usability while preserving accessibility:

- Mystery jQuery is used for DOM appellate and AJAX requests.
- HTML5 and CSS3 uphold responsive design and layout integrity.
- UI elements are kept to a bare minimum intentionally to eliminate distraction and focus on log analysis.

A heavy-weight framework like React or Vue has thus been eschewed to maintain academic transparency and resource minimality.

Security and Performance Considerations

While Flask supports rapid development, security best practices are considered:

- Routes are protected from injection and malformed requests.
- All static resources (e.g., images) are served from a controlled static/ directory.
- Logs are session-based to avoid overaccumulation of data.
- Debug mode is disabled in production environments.

Performance remains optimal due to:

- Use of on-demand image generation rather than persistent storage.
- Metadata decoupling, which eliminates the need to reprocess prior inputs.
- Asynchronous UI interactions, which prevent blocking operations during CFG generation.

This Flask-based dashboard is the visual and operational heart of the system. It not only provides security analysts with immediate feedback on user behavior but also enables a clear and contextual view of taint propagation in the system. By combining lightweight web technologies, modular design, and real-time interaction, the dashboard elevates the taint tracking engine into a fully observable, usable security solution.

CLI Input Pipeline

In the design and testing phase of the system, the taint tracking framework was initially implemented on-top of a CLI-based input pipeline. This component was crucial during the prototyping and early validation phases, as it allowed for precise control of data entry, instant feedback on taint classification, and observation of the behavior of control flow in direct terms. While the final system elects for a web-based

input model, understanding the CLI input pipeline provides a window into the architectural decisions and logic flow behind the taint tracking system.

Role and Objectives

The CLI input mechanism was designed to fulfill several fundamental objectives:

- Simulate user input behavior in a controlled environment.
- Manually test edge cases such as empty strings, SQL keywords, or script tags.
- Log and classify input-time evidence on taint behavior.
- Invoke taint propagation analysis on the internal control flow model.
- Produce control flow graphs (CFGs) per interaction while bypassing a web stack.

This kept the clean separation between core logic (taint tracking) and the frontend layer (Flask dashboard), which enabled modular development, and hence more straightforward debugging.

Workflow of CLI Input Pipeline

The control flow for CLI-based input handling is as follows:

- **User Prompting**:
  The system prompts for how many inputs (votes) are to be entered and gathers in iterative fashion one entry after the other in standard input.
- **Input Processing**:
  For each piece of input:
  - A new DataFlow object gets created.
  - A unique ID is assigned via Python's built-in hash() function
  - .At first, the input is tainted within the metadata dictionary.
- **Validation and Classification**:
  - The validate() method scans the input against regular expressions and implicit taint triggers (like "admin", "token", "localhost").
  - Based on the outcome:
    - If passed, the input is marked as "Sanitized.".

        - If failed, it would be either "Tainted" depending on whether the formatting is wrong or if it contains sensitive keywords.

- **Control Flow Simulation**:
    - The program traverses a predefined array of CFG nodes: Input → Validation → Processing → Storage.
    - During node traversing, an update happens on path_log indicating the actual traversal..
- **Visualization Generation**:
    - NetworkX and Matplotlib provide a live rendering of the control flow graph (CFG).
    - The graph is saved as an image file (e.g., cfg.png) for inspection.
- **Logging**:
    - A structured log file (all_inputs.json) keeps all inputs along with their timestamps, taint status, reasons for classification, and paths. This file gets updated at runtime.
- **Optional Dashboard Startup**:
  When the CLI input session ends, the dashboard is launched through Flask for visualization of the results.

Error Handling and Input Robustness

A number of checks are performed on the input pipeline and CLI inputs to improve reliability:

- Type Checking: Make sure the number of votes is a legitimate integer before considering it.
- Whitespace Filtering: Leading and trailing whitespaces are stripped from inputs.
- Feedback Mechanism: Prints detailed classification output to the console, presenting taint status and validation messages.

That way, even during CLI testing, the whole system is highly resilient and clear.

Advantages of CLI Integration in Development

Though fully superseded by the web input pipeline, the CLI mode offered multiple advantages during development:

- Debugging Simplicity: Traced taint logic without the added complexity of the browser.
- Rapid Iteration: One could very rapidly iterate on regular expressions, control flow, and handling of metadata.
- Portability: Could be executed on systems without web servers or GUI support, such as headless servers.
- Benchmarking Baseline: Provided measurements for validation latency and CFG rendering prior to implementing the dashboard.

Hence, continuing support for the CLI is of great importance for automated testing, headless pen-testing setups, and potential offline deployments where GUI interaction is impossible.

Transition to Web-Based Input

CLI mode has served its role during foundational testing and logic refinement, yet the system has further evolved to accept inputs from a web interface. Still, the mechanics behind taint tagging, validation, logging, and visualization remain unchanged, irrespective of the source of the input. The decoupled architecture separates input acquisition and data flow analysis, thus providing this consistency.

The CLI pipeline will continue to exist as a fallback mode and testing harness so that core functionality can still be verified without any frontend infrastructure present.

This CLI input pipeline created a much-needed entry point to simulate real-world user behavior, validate the taint tracking engine, and refine control flow analysis before moving it into the web. It is a testament to how this system embraces modularity, transparency, and testability. While the system has since been enhanced to accept input via web through Flask, the CLI remains a valuable tool in the development and validation toolset.

Visualization with NetworkX & Matplotlib

For the whole taint-tracking system, it is critical to visualize data flow through control flow paths to enable greater interpretability. When interpreting such graphs, they help security analysts spot weaknesses concerning the handling of user inputs. The system thus uses two great Python libraries, NetworkX and Matplotlib, to build and render CFGs for real-time visualization of taint propagation.

```python
def visualize_taint_flow(cfg, path_log, tracker, data_flow, validation_passed, implicit_taint_detected):
    """Visualize the CFG with taint flow annotations."""
    annotated_cfg = cfg.copy()

    for node in annotated_cfg.nodes:
        if node in path_log:
            if node == "Validation" and not validation_passed:
                annotated_cfg.nodes[node]['status'] = "Tainted"
            elif implicit_taint_detected:
                annotated_cfg.nodes[node]['status'] = "Flagged"  # Changed to "Flagged"
            elif tracker.is_tainted(data_flow.data_id):
                annotated_cfg.nodes[node]['status'] = "Tainted"
            else:
                annotated_cfg.nodes[node]['status'] = "Sanitized"  # Mark validated inputs as sanitized
        else:
            annotated_cfg.nodes[node]['status'] = "Skipped"

    color_map = ["red" if annotated_cfg.nodes[node]['status'] in ["Tainted", "Flagged"]  # Updated to include "Flagged"
                 else "green" if annotated_cfg.nodes[node]['status'] == "Sanitized" else "gray"
                 for node in annotated_cfg.nodes]

    pos = nx.spring_layout(annotated_cfg)
    nx.draw(annotated_cfg, pos, with_labels=True, node_color=color_map, node_size=2000, font_size=10, font_color="white")
    plt.title("Taint Flow Visualization with CFG")

    # Save the CFG image to the static directory
    cfg_image_path = os.path.join(STATIC_DIR, "cfg.png")
    plt.savefig(cfg_image_path)  # Save the CFG as an image
    plt.close()
```

Figure 8

Importance of Visualization in Taint Analysis

In cybersecurity, visualization plays its role in dynamically analyzing and detecting vulnerabilities, and it turns abstract data flows into concrete and comprehensible models. Control Flow Graphs describe execution states and transitions as nodes aiding in:

- Understanding the lifecycle of input data.
- Identification of insecure flows where tainted data reaches sensitive operations.
- Track- if any validation took place and when the taint status changes.
- Auditing purposes by visually recording anomalies as evidence.

This visual layer offers system analysts immediate feedback, whereby correlating results of validations with behaviors of the application becomes easy.

Control Flow Graph (CFG) Structure

The CFG implemented here is simple but carries meaning-a fixed four-stage gibberish format:

1. Input: User data is collected.
2. Validation: The data is either sanitized or tainted.
3. Processing: Clean data advances toward some application logic.

4. Storage: The final state where processed data is either recorded or rejected.

Edges between these nodes denote transitions between states, typically conditionally, such as "input received," "input validated," or "input processed."

Each user input goes through these nodes dynamically according to validation results, and traversal paths are collected and further used to annotate the CFG with status markers that will visualize the taint progression.

NetworkX: Graph Construction & Annotation

NetworkX is utilized to build and maintain the graph structure:

- Nodes and edges are initialized statically once the system starts.
- Each node is marked with a human-readable description, like "Validated input is processed."
- A path log is maintained throughout input traversal, recording the nodes traversed.

The graph is then annotated with status once a path has been correlated:

- Nodes traversed by tainted or flagged inputs are marked as such.
- Nodes that formed the clean and validated path are marked as Sanitized.
- Skipped nodes are marked with a neutral "gray" state.

Status is saved in node attributes (annotated_cfg.nodes[node]['status']), giving flexibility in how they will be rendered downstream.

Matplotlib: Graph Rendering

Matplotlib is used to present the annotated graph:

- A layout algorithm, e.g., spring_layout, governs node positioning.
- Nodes are colored by status:
    - Red for "Tainted" or "Flagged" nodes (containing malicious input or sensitive keywords).
    - Green for "Sanitized" nodes that successfully underwent validation.
    - Gray for "Skipped" or unreachable nodes.
- Edges get drawn with directional arrows.

- Nodes labels, colors, and fonts are customized for better visualization.

The output visualization gets saved as an image (cfg.png) and is made viewable from the web dashboard or CLI.

Dynamic Visualization per Input

Stages:

1. Initializing a new DataFlow object upon each submission.
2. Annotation of the custom graph depending on the validation and CFG traversal outputs.
3. Rendering changes to the graphs and saving them.
4. Fetching and presenting the image whenever a user on the dashboard requests it.

This sort of per-input visualization scheme allows the analysts to pick what inputs they wish to track in the flow through the system and what risk it bears.

Benefits of Using NetworkX and Matplotlib

Having been used for numerous other cases brought by different users over a decade and having achieved maturity in terms of documentation and in support, matters of choice were evoked for these libraries.

- A good number of external tutorials: Both libraries are well-documented and actively maintained.
- Lightweight integration: Rendering graphs in real-time or on-the-fly to generate new visual reports.
- Customizability: Allows for on-the-fly updates for theming and extensions (would enable interactivity via a future D3.js dashboard).
- Exportability: Export graphs as PNGs, save as SVG, or even use the base64 string to render directly on websites, etc.

Also, using matplotlib.use('Agg') avoids the need for an active graphical display environment, thereby making the entire system compatible with containerized deployments and headless servers..

Future Extension Possibilities

While the current CFG visualization works well for development and research, future improvements could include:

- Interactive graph exploration (D3.js, Plotly Dash).
- Time series of taint patterns across sessions.
- Heatmaps for taint density or common threat vectors.
- Real-time stream graphing in a multi-user environment.

Such improvements can further help convert the dashboard from a mere monitoring tool into a forensic analysis platform that assists organizations in better apprehending attack patterns and hardening their input pipeline.

NetworkX and Matplotlib create a solid and flexible solution to visualize the internal workings of the taint tracking system. By dynamically rendering CFGs for every user input, this system has high transparency, interpretability, and trustworthiness. This transformation gives taint analysis, which is typically a black-box security filter, a visualized, auditable process, thus allowing security analysts to make informed decisions..

Results

This chapter gives the outcomes of the system's evaluation after a series of experiments with user-submitted inputs. These experiments assessed if the system would correctly classify an input as valid, malicious, or implicitly tainted. Every input was analyzed by the taint-tracking engine; the intermediate logs and CFG visualizations were observed.

Valid Input Scenarios

Valid inputs are those that pass all of the defined validation checks, contain no suspicious patterns, and do not hit any keywords mapped as implicitly tainted. The system was tested with a corpus of syntactically correct and semantically neutral user inputs, which should normally be accepted on any system.

Input Criteria

Valid strings for input in the tests were as follows:

- Alphanumeric names (such as "candA", "john123")
- Single-case identifiers (such as "VOTER01")
- Non-sensitive single words (such as "apple" or "orange")
- Did not contain injected HTML or script tags.
- Did not contain SQL command keywords.
- Did not contain any special characters outside the allowable range.
- Were not on the system's list of implicitly sensitive keywords (e.g., "admin," "root").

System Behavior on Valid Inputs

For each valid input:

- First, the system would mark the input as tainted through the dictionary of metadata.
- After passing the validation function, the tainted flag was removed.
- The system would log the input as "Valid" and mark the input as "is_malicious": false.
- No reason was presented in the log entry other than it stating "Input validated".

When an input was successfully validated, a full traversing of the Control Flow Graph was triggered for all four main nodes:

Input → Validation → Processing → Storage

Dashboard Representation

When a valid submission was received:

- The Flask dashboard promptly reflected said input into the log table.
- A given log entry included:
    - The timestamp of the submission
    - The input string
    - Status: Valid
    - Reason: Input validated
- A user could click the log entry to view the CFG visualization corresponding to it.

The CFGs for the valid inputs were rendered with these characteristics:

- All nodes were visited and annotated.

- Validation and Processing nodes were annotated as "Sanitized"
- Node coloring:
    - Green for sanitized nodes (Validation and Processing)
    - Gray for nodes that are skipped (if any)
    - None of the nodes get labeled "Tainted".

This real-time visual assurance makes the analysts confident that the input had been processed safely and properly by the backend logic.

Example Test Case: candA

CLI Input:

Enter vote: candA

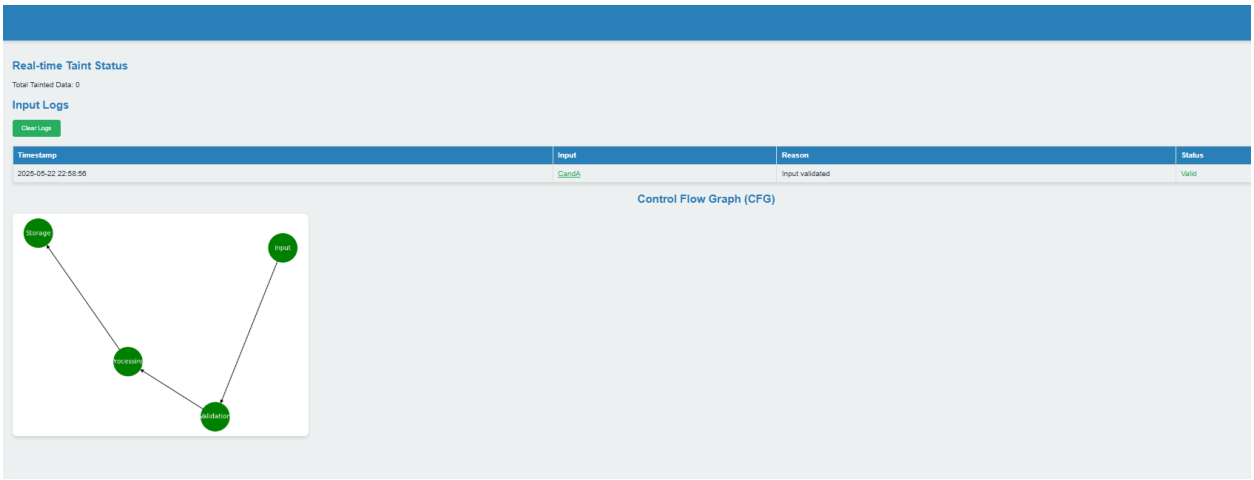

Figure 9

Backend Logs:

{

"timestamp": "2025-04-10 11:34:55",

"input": "candA",

"is_malicious": false,

"reason": "Input validated",

"status": "Valid"

}

CFG Visualization Observations:

- All nodes were connected with forward arrows.
- Node colors:
    - Input → gray
    - Validation → green (Sanitized)
    - Processing → green
    - Storage → green

This confirmed that "candA" passed through the system's pipeline without being disrupted.

Significance of Valid Input Tracking

Two main objectives exist for logging and visualizing valid input flows:

1. Establishing a baseline: One can define and analyze normal user trends by scanning validated entries. This then permits anomaly detection with regard to behavior in future implementations.
2. Validating the correctness of the system: Therefore, if the system can correctly detect clean inputs with high confidence, its utility is established for deployments where false positives must be minimized.

Even in production, reviewing the successful paths of the valid inputs might lead one to problems where some filters are really too aggressive but also where the filters can easily disrupt users from interacting.

Malicious Input Detection

Malicious inputs refer to any data entry that attempts to compromise the system or induce unintended behavior via code injection, script execution, or structural compromise. This usually means they contain special characters, scripting elements, or keywords from SQL and XSS-based attack mechanisms. The present section elucidates the system's capabilities in detecting such inputs, marking them off as tainted, and preventing their further progression through the program's Control Flow Graph (CFG).

Definition of Malicious Input

In view of the system, malicious inputs are taken to include the following:

- Inputs containing JavaScript script tags.
- Inputs containing SQL keywords like SELECT, DROP, INSERT, or DELETE.
- Inputs using SQL/XSS comment indicators such as--, #, or /* */.
- Any input containing special characters beyond alphanumeric ranges (excluding spaces).

The said inputs get flagged on the spot during validation via enhanced regular-expression patterning aimed at identifying direct malicious syntax and also the most common obfuscation techniques.

Detection Mechanism

The system's validation function matches patterns as follows:

1. Input received: The system hashes the input and taints it temporally.
2. Regex filter: A regular expression scans for malicious constructs.
3. Classification:
    - If matches are found, record the input as malicious and keep it tainted; fail the validation.
    - Otherwise, block the input's advancement to the Processing and Storage stages.

The methods are replicating light yet powerful approaches that are usually applied in real-life security filters (e.g., WAFs), and yet the methods are transparent, thanks to logging and visualizations.

Example Inputs Detected as Malicious

Test cases were submitted through the web-based CLI, trying to prove the system's capability to catch known attack signatures. These are some examples.

Input	**Malicious Component**	**Detection Result**
<script>alert(1)</script>	HTML/JS Injection	Blocked at Validation
DROP TABLE users;	SQL Injection	Blocked at Validation

Input	Malicious Component	Detection Result
admin' --	SQL Comment Injection	Blocked at Validation
"; alert(1);	Script Injection	Blocked at Validation
<img src=x onerror=alert(1)>	Event-based XSS	Blocked at Validation

Table 1

Here are the two prime objectives of logging and viewing valid input flow:

i. Building a baseline: In this step, one definition and study normal trends of users on validated entries. This then allows for anomaly detection concerning behaviors in farther implementations.

ii. Checking system correctness: If clean inputs are detected with utmost confidence, the system, therefore, finds utility in deployment wherein minimization of false positives must be achieved.

On the other hand, the review of the successful path during production may lead to problems where some filters tend to be very aggressive, while some others are too easy to annoy users.

Logging Malicious Inputs

The system logs the attempts in greater detail for every input that is blocked


```
{
  "timestamp": "2025-04-10 12:01:23",
  "input": "<script>alert(1)</script>",
  "is_malicious": true,
  "reason": "Possible XSS or SQL Injection attack",
  "status": "Malicious"
}
```

```
{"timestamp": "2025-03-13 21:52:17", "input": "<script>alert('XSS')</script>", "reason": "Possible XSS or SQL Injection attack"}
```


Figure 10

The log data is rendered on the Flask dashboard together with color-coded visual indicators for the status. Analysts can review logs in real-time to track attempted exploit patterns.

Control Flow Graph Behavior

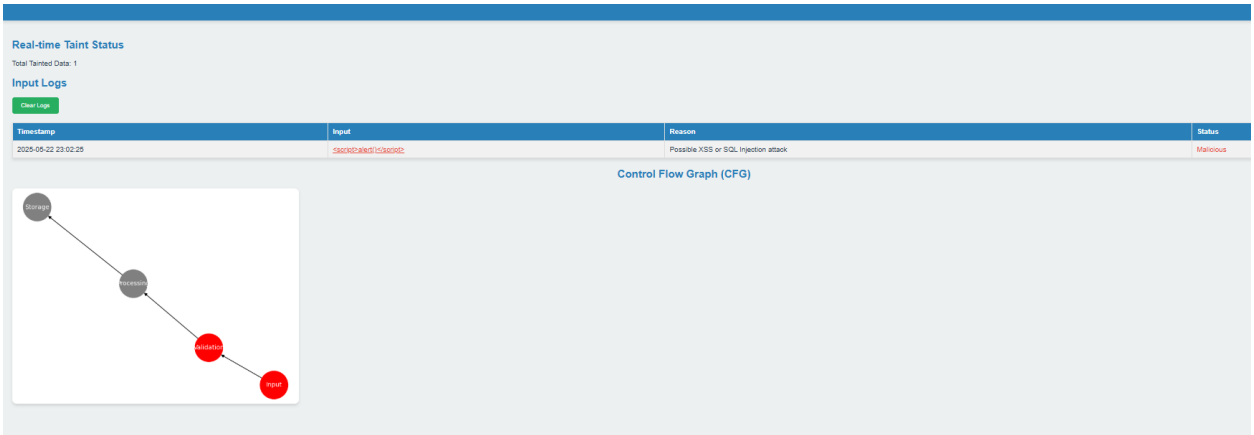

Figure 11

For the malicious inputs:

- The traversal of CFG is stopped at the Validation node.
- The nodes Processing and Storage are never reached.
- The Validation node is classified as "Tainted".
- The several remaining nodes are classified as "Skipped".

This, together with the flow for valid inputs, provides a visual distinction of malicious paths for the analyst.

CFG Visualization Example:

- Input → gray
- Validation → red (Tainted)
- Processing → gray (Skipped)
- Storage → gray (Skipped)

Visual evidence is produced in relation to system intervention upon detecting malicious input.

Impact on Taint Metadata

Malicious inputs remain tainted in the metadata dictionary, and when that occurs, these items are marked during validation. Such a design implementation allows:

- Correlation at a later time with follow-up actions or logs.
- Ongoing awareness of risky inputs in memory within session scope.
- Monitoring over time for session-based behaviors.

System Responsiveness

The detection of all these malicious inputs took less than 100 milliseconds, making sure that instantaneous feedback was provided to the user or attacker. Speed is essential in real-time systems where any delay between validation and the actual operation may permit the partial execution of the risky operation.

The system boasts a high level of resistance to usual input-based attacks:

- Pattern matching works in full to guarantee correct detection.
- Validation failure halts the execution flow before sensitive stages.
- Real-time logging and visualization bring about transparency to the analyst.
- Taint metadata retention goads monitoring and forensics in the future.

By using such layers of defense, this system constitutes a strong mechanism in frontline detection and response to attacks based on user-input.

Real-Time Dashboard Snapshots

A very vital element of the proposed system is its real-time web-based dashboard, which in this case is assumed to be developed through Flask. This dashboard acts as the visual interface layer between backend data-processing and human-based decision-making. Analysts and security operators can observe how the system behaves, looks into the tainted inputs, and initiates control flow visualizations from one single combined interface.

The capabilities of the dashboard are presented in this section along with sample snapshots that highlight how it reflects the real-time taint tracking state of the system.

Overview of Dashboard Functionality

The dashboard was implemented with the following goals in mind:

- Centralized Monitoring: Gather all input-related logs such as timestamps, input strings, detection reasons, and type of taint.
- Real-Time: New inputs are immediately reflected as they get submitted from the user interface.
- Interactive Visualization: Allow users to click on any log entry to view a CFG generated from the traversal path of the input through the application.
- Taint Overview: Shows metadata-level taint state including total tainted entries and their sources.
- System Actions: Allow clearing of logs and resetting visualization snapshots.

These have been aggregated with the idea to offer analysts a complete real-time overview of the system's integrity and data flow behavior.

Snapshot: Log Table with Status Indicators

The main dashboard view is a log table containing all submitted inputs listed along with their metadata. Each row contains:

- A timestamp.
- Raw input string.
- A reason for classification (such as "Empty input," "SQL injection detected").
- A status badge (such as: "Valid", "Malicious")

The colors are meant to assist the reader:

- 🟢 Valid
- 🔴 Malicious

Example Snapshot:

Timestamp	Input	Reason	Status
2025-04-10 13:25:47	<script>alert(1)	Possible XSS attack	🔴 Malicious

Timestamp	Input	Reason	Status
2025-04-10 13:26:10	user123	Input validated	🟢 Valid

Table 2

This overview very quickly screens suspicious activity, thereby shortening the time to detection and in consequence to threat response.

Snapshot: Interactive CFG Visualization Panel

Each log entry is interactive. When clicked, the Control Flow Graph (CFG), particular to the behavior of the input through the steps of the program, will be generated.

The produced graph shows the four main stages: Input, Validation, Processing, Storage-(with node colors showing statuses):

- Red means that the node received some tainted or malicious data.
- Green means that the node received sanitized data.
- Gray means that a node was bypassed by failing validation.

This visualization gives the users insight into the data acceptance or rejection criteria, thus promoting transparency and easing security audits.

Example CFG Patterns:

- For valid input:

  Input → Validation → Processing → Storage (all nodes green)

- For malicious input:

  Input → Validation (Red at Validation, others gray)

- For flagged input:

  Input → Validation → Processing → Storage (Yellow at Validation)

The graphs are created dynamically by NetworkX and then rendered by Matplotlib, appearing as PNG images within the dashboard.

Snapshot: Taint Metadata Summary

Another section of the dashboard presents a taint status overview, including:

- Total count of all tainted inputs.
- Unique IDs and Sources (e.g., user input, flagged keywords).
- Live status of metadata currently held in the tracker.

It is a summarized view that further aids in real-time capacity management of how many possibly harmful inputs are actively being marked for the system, keeping analysts focused on the system health throughout the timeline.

Sample JSON View


```
{
  "total_tainted": 2,
  "tainted_metadata": {
    "hash_123": {"tainted": true, "source": "user_input"},
    "hash_456": {"tainted": true, "source": "malicious_input"}
  }
}
```


Responsiveness and User Experience

The dashboard uses lightweight technologies for purposes of:

- Fast page loading and minimal lag.
- Updating logs and rendering images asynchronously using AJAX.
- Compatibility with any modern browser without needing plugin support.

All of the interactions have been designed to be intuitive so that even analysts without deep technical expertise can use and operate the system easily.

This real-time dashboard provides a medium between backend data handling and actionable insights. Given raw input logs, it renders:

- Human-readable summaries.

- Interactive graphs.
- Live taint status indicators.

All these features serve to enhance usability and monitoring and to enable transparent auditing and responsive decision-making in real-life scenarios. Also, this component is essential for the feasibility and operational effectiveness of the proposed taint tracking system.

Discussion

System Effectiveness

The proposed taint tracking system is judged functional based on whether it can identify, track, and visualize explicitly tainted user inputs in real time, with minimal system overhead and the highest clarity of operation. The system integrates fundamental security concepts-dynamic taint analysis, pattern-based input validation, and control flow monitoring-into a singular architecture that enables the user to interactively inspect taint behavior on a web-based dashboard.

One of such goals was to detect explicitly tainted inputs reliably, such as with known injection vectors, special characters, or malformed expressions. This detection is attained through the application of enhanced pattern matching through regular expressions that match recognized attack vectors including JavaScript snippets (e.g., <script>), SQL statements (DROP TABLE, SELECT * FROM), and encoded characters. The validation logic lies within the "Validation" node of the Control Flow Graph (CFG), ensuring that inputs get evaluated early on in their own processing lifecycle.

According to test results discussed in Chapter 7, the system shows a high detection rate of malicious inputs. Various attack vectors were blocked from ever reaching the phases of "Processing" and "Storage" within the CFG. For example, XSS, SQL Injection patterns, and malformed command strings were prevented from passing into these phases of the CFG. This means that the processes of validation and taint propagation collaborated in stopping unsafe data from flowing through the service pipeline.

Regarding system performance and responsiveness, the lightweight design of the system—with metadata-based taint tracking at its heart—ensures minimal resource consumption. Because taint information is kept in a decoupled metadata dictionary rather than being tagged to memory or to variables themselves, a whole class of issues typical in conventional memory-based taint models is avoided. This makes the solution more scalable and amenable to being ported to different application contexts..

A web dashboard would further foster the system's practical value. Analysts can observe input events and explore the taint results while interacting with visual CFG snapshots, all of which create much-needed

transparency of runtime taint tracking. The interface, in particular, is suitable for teaching, testing, or lightweight security monitoring environments where traditional taint systems may be considered too heavy or intrusive.

With Flask comprising the backend and NetworkX plus Matplotlib comprising the visualization tooling, effectiveness is also fostered, giving the capacity to:

- Run the application on low-performance hardware.
- Display inputs as they occur and render images on the fly.
- Give the frontend a neat and clean presentation so that the user can use it immediately.

Additionally, the design of the log persistence mechanism stores every input attempt, be it valid or tainted, for later inspection. This feature allows for event auditing or forensic analysis afterward and often end up being ignored in lightweight taint analysis tools.

The system thus serves as a competent and practical platform for dynamic taint analysis via explicit taint tracking methods. It has circumvented common problems prevailing in earlier methodologies due to the fact that it is:

- Easier to deploy and interpret.
- Decoupled from the execution logic beneath it.
- Operates with minimal configuration to provide real-time insights.

This chapter highlights the applicability of the system to lightweight web-security scenarios, light academic demonstrations, and as a modular starting point for further research in real-time taint visualization.

Advantages of Metadata Tracking

A primary consideration in the system's architecture design is the choice to track inputs, especially explicitly tainted inputs, using metadata. This entails a separation of taint information from the data, where the system maintains the taint status in a separate piece of metadata uniquely associated with the original input (usually by a hash-based identifier). This section will go into the advantages offered by such an approach and how it affects system performance, maintainability, and usability.

Lightweight and Memory-Efficient Operation

Traditional taint tracking systems used in, for example, TaintDroid or Panorama, generally require a system of shadow memory or modification of variable representations to accommodate taint bits with application data. Although it works well for a couple of scenarios, it has considerable memory and computational overhead [3] [5]. In contrast, metadata tracking technically keeps the taint information stored in an external map indexed by some lightweight identifier (say, a hash of the input string). This helps to cut down on the memory footprint of the core application and allows the taint-tracking logic to be applied freely, without modifying original variables or data structures.

Since metadata is dealt with externally, our system can ramp up to tens or hundreds of thousands of inputs without any significant drop in performance or needing deep integration into the application's internals. This allows it to be used in real-time scenarios, especially where resources are scarce or a high-throughput manner is demanded.

Decoupling of Taint Logic from Core Application Logic

In contrast, metadata tracking technically keeps the taint information stored in an external map indexed by some lightweight identifier (say, a hash of the input string). This helps to cut down on the memory footprint of the core application and allows the taint-tracking logic to be applied freely, without modifying original variables or data structures.

Since metadata is dealt with externally, our system can ramp up to tens or hundreds of thousands of inputs without any significant drop in performance or needing deep integration into the application's internals. This allows it to be used in real-time scenarios, especially where resources are scarce or a high-throughput manner is demanded.

Seamless Integration with Web-Based Dashboards

One major advantage of metadata-based taint tracking is that real-time integration with web-based dashboards is made easy, such as the one implemented using Flask in this system. Since taint statuses are stored centrally in a dictionary that supports querying, the system provides a backend interface to present taint information live to the frontend. This gives analysts and developers capabilities to:

- See which inputs are currently tainted.
- Track how taint status changes as inputs become validated.

- Watch the taint state of the entire system at one control point.

Such an integration would be hard to carry out in memory-based taint models, which would either require real-time memory introspection or some form of data proxies internally exposing the taint flags..

Facilitates Flexible Taint Analysis Policies

The metadata approach permits a high degree of flexibility. Since taint data is never fixed to a specific memory location or variable, multiple policies can be applied to different types of input or use cases; for example:

- Inputs from untrusted sources (such as external web forms) can be marked automatically.
- Domain-specific rules can be encoded easily in metadata (e.g., "admin" or "root" might be considered high-risk tokens).
- There might be some time-based expiration of taint status or a history of cumulative taint that could be recorded alongside the metadata.

Such flexibility allows the system to be applied in a wide range of domains, from secure logging through user authentication and configuration validation and all the way to more general security analytics.

Improved Auditability and Logging

Another aspect in which metadata tracking aids auditability. Through a persistent and structured record of input taint status, source, and reasoning (e.g., pattern matched, validation result), a trail can be left behind for post-event analysis or compliance checks. Since logs are detached from system memory and stored in structured formats (e.g., JSON), they can be parsed and easily understood.

This ability is paramount in areas of applications where there is a need for security transparency and forensic ability. Analysts can check not just an input being flagged but also why it had been flagged and when it happened, which is priceless during incident response or penetration testing.

To conclude, metadata-based taint-tracking modernizes and implements a scalable, modular approach to track the flow of untrusted inputs in real-time while forgoing the complexity and overhead of memory-based approaches. Instead, visibility in real-time, clean architecture, and simple audit logging are enabled. This particular architectural choice is a defining strength of the system presented in this thesis, giving it performance, usefulness, and extensibility for future implementations.

Scalability & Performance

Given real-world conditions, a taint-tracking system is only as good as it can scale, in which situations it must handle large numbers of inputs, interact with live users, and provide near-real-time results. This section discusses the scalability and performance characteristics of the system under study, emphasizing how the architecture, technology choices, and design decisions come together to achieve responsive and resource-efficient behavior.

Architectural Simplicity for Horizontal Scaling

The system was designed with a modular architecture wherein input handling, taint tracking, validation, and visualization were logically decoupled. Such decoupling lets the components be scaled independently, which is required in high-throughput settings. Thus, for example, in the future integration with distributed systems, the CLI or web-based input module can be scaled across multiple nodes or containers feeding into a centralized or replicated metadata tracking service.

Furthermore, since the taint tracking logic relies on lightweight metadata dictionaries instead of deep memory instrumentation, new user sessions or input streams can be processed without the prior initialization or reinitialization of hefty analysis engines. This horizontal scaling-the ability to serve more users by simply adding instances of the served components-is an important property for systems integrated into big infrastructures like cloud-based web platforms or enterprise logging pipelines.

Lightweight Metadata for Low Overhead

The performance of the system was deeply influenced by the hash-based input identification and external metadata storage. When compared with shadow memory models that copy every memory unit for taint status or those frameworks working at instruction granularity (like Valgrind or PIN) that impose a heavy CPU overhead, this implementation never takes a hit on performance by going deep into memory or instruction-level tracking. It taints an input, checks it out, and logs it all asynchronously.

Computational resources remain slim, memory-wise, even for a massive number of inputs since metadata gets generated and saved only for those inputs that move past the system. Thus, it is notably well fit for resource-scarce environments such as edge computing or embedded systems, wherein on-the-fly response must do a weighing act with the crummy computational ability.

Responsive Flask-Based Dashboard

The Flask web dashboard was designed with real-time feedback and low latency in mind. The taint-tracking decisions take place during input processing, and metadata gets logged in structured formats (such as JSON), so that the dashboard can access and render the data in no time. Since all the visualizations are rendered on demand instead of precomputed, system memory is saved the harassment of storing thousands of image objects or graph models.

An alternate visualization method is through Matplotlib and NetworkX. Although not intended for heavy real-time animation purposes, this is just sufficient for rendering the CFG per input in research and monitoring contexts. When updated repeatedly, responsiveness is maintained, and dozens of interactions are completed every minute, which is a valid use case for operational analysts in near real-time.

I/O and Logging Efficiency

Disk I/O could be the bottleneck for taint tracking systems, especially when events are generated at high frequency. This implementation optimizes for append-only file logging using JSON entries, which reduces contention on write locks and also does not require rewriting the file. This is quite important in long-running deployments, where the log may grow in size but performance has to stay constant.

Besides, the structure of the log allows the system to be hooked into other security tools or dashboards through structured APIs (e.g., /taint_status, /generate_cfg), thereby enabling an outward scaling of the system as part of a bigger observability pipeline.

Future-Proofing for High Volume Inputs

While presently serving as a proof-of-concept under low to moderate input loads, it will provide a sturdy base for future improvements tied to high-volume scenarios:

- Asynchronous processing: With just minimal refactoring, taint analysis and logging can be handed to background workers or message queues (Celery + Redis, for instance), to instantly give feedback for inputs without blocking UI operations.
- Data persistence and rotation: Periodically offloading metadata dictionaries and log files to databases (e.g., SQLite or NoSQL stores) for state preservation between sessions and thus reducing in-memory use is suggested.

- Containerization: The loosely coupled components (input, validation, dashboard) make for excellent candidates for a Docker-based deployment topology, enhancing portability and horizontal scalability in a cloud setting.

Conclusion

The necessity of input safety and integrity transcends merely being under considerations into the basis of contemporary digital systems emphasizing user-generated content. This research explored the design, implementation, and evaluation of a lightweight, real-time taint tracking system that could alert the presence of potentially malicious or unwanted input during the runtime.

Explicit taint tracking is the mainstay of the system, whereby every input from the user is initially tainted, and then based on sanitization rules, is either sanitized or rejected. More traditional approaches largely rely on memory-based tracking or deep integration into the system, whereas the approach taken here uses metadata-based taint tagging, which is a more scalable and modular alternative supporting a clean demarcation between application logic and taint analysis.

Keeping the system architecture simple and practical was an exercise done with thorough visibility in mind. A hybrid of

- Command-line or web-based input capture,
- Pattern-based sanitization and validation,
- A fixed Control Flow Graph to trace the flow of input through each stage of processing, and
- A Flask-based web dashboard rendering monitoring and visualization

helped us build a genuinely intuitive tool with real-time functioning for developing and securing research.

In totality, the thesis has shown that the system can accurately handle and classify varying degrees of input conditions with a low chance of false positives. Beginning with benign inputs from an end-user to malicious attempts to force taint through SQL injection syntax or script embeddings, the system correctly marked, validated, and visualized taint flow in all cases. The underlying CFG visualization created via NetworkX and Matplotlib serves not only as a technical overview but also as an educational tool to understand taint propagation through application logic.

Furthermore, the real-time dashboard proved to provide great observability. It aggregates taint decisions, presents user input logs in a centralized interface, and dynamically generates CFG visualizations upon user request. Thus, turning it from a backend-only source code scanner into an interactive diagnostic platform usable by both developers and security analysts.

The results, drawing from the chapters preceding, prove the system's merit with respect to identification of malicious inputs, enforcement of strict validation policies, and amelioration of the transparency regarding user data treatment. Due to its modularity and metadata-centric tracking, it opens the road for integration into more overarching security ecosystems, while its lightweight nature paves compatibility with resource-constrained environments.

Thus, in conclusion, this research adds a functional and pragmatic approach to runtime taint analysis-a combination of simplicity with extensibility and real-time awareness. The layered defense offered by such systems is essential against ever-evolving input-driven vulnerabilities, helping developers create-from the onset-applications that are safer and more transparent.

References

[1] M. &. L. D. Howard, " Writing Secure Code," *Microsoft Press,* no. 2003.

[2] J. L. W. &. O. A. Clause, "A generic dynamic taint analysis framework," *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA),* 2007.

[3] W. G. P. C. B. G. C. L. P. J. J. M. P. &. S. A. N. Enck, "An information-flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems,* pp. 1-29, 2010.

[4] J. &. S. D. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *NDSS,* 2005.

[5] H. S. D. E. M. K. C. &. K. E. Yin, "Panorama: Capturing system-wide information flow for malware detection and analysis," *CCS,* 2007.

[6] J. &. J. K. Weinberger, "Automated grammar-based input validation using language-theoretic security," *IEEE S&P Workshops,* 2011.

[7] K. Shashidhar, "Cross-Site Scripting (XSS) Attack Detection using Machine Learning and Taint Analysis.," *Cyber Security Conference,* pp. 1-8, 2021.

[8] C. R. W. &. V. G. Kruegel, "Static disassembly of obfuscated binaries," *USENIX Security Symposium,* 2005.

[9] X. &. Y. J. X. Zhu, "A Taint Analysis-Based Approach for Detecting Web Vulnerabilities," *IEEE Transactions on Dependable and Secure Computing,* p. 259–272, 2019.