SQLI GUARD - SQL INJECTION DETECTION USING GENERATIVE AI

ABSTRACT:

The SQLI Guard project is an advanced AI-powered suite designed to detect, analyze, and mitigate SQL injection attacks with cutting-edge machine learning and generative AI techniques. Leveraging a BERT-based model for query classification, a GAN for adversarial query generation, and Stable Diffusion for visualizing attack scenarios, the system provides robust detection of malicious SQL patterns, categorizing queries as Safe, Suspicious, or Malicious. It incorporates a community-driven pattern library to identify injection techniques, such as tautologies, stacked queries, and encoded payloads, with severity levels from low to critical. The suite offers comprehensive visualization tools, including 3D scatter plots, heatmaps, network graphs, to illustrate query structure, risk distribution, and sanitization flows.

Additional features include query sanitization, parameterized query generation, threat persona analysis, and a browser-based honeypot simulator to trap and study attacks. Integrated with a interface, natural language processing, and text-to-speech capabilities, SQLI Guard enables users to interact seamlessly, convert natural language to secure SQL, and explore attack simulations. The system also supports community contributions for pattern updates and provides detailed auto-explanations powered by the Groq-API, making it a versatile tool for cybersecurity professionals to enhance database security and resilience against SQL injection threats.

SQL Injection attacks can lead to unauthorized access, data breaches, and complete database compromise by exploiting vulnerabilities in web applications. TraditionalSQLi detection methods lack adaptability to evolving threats and zero-day attacks.This project introduces a Generative AI-powered SQL Injection Detection and Prevention System that leverages deep learning and AI-generated synthetic attack patterns to fortify database security.

PRETRAINED MODELS USED:

In this sections it describes about the pretrained model and its configurations used in the system .The SQLI Guard system leverages three pretrained models to detect, simulate, and visualize SQL injection attacks. Below is a concise overview of each model:

1.BERT (Bidirectional Encoder Representations from Transformers):

Model: bert-base-uncased (Hugging Face Transformers).

Purpose: Classifies SQL queries as "Safe," "Suspicious," or "Malicious" to detect injection risks.

Architecture: 12 transformer encoder layers (768-dimensional, 12 attention heads) with a classification head outputting 3 classes.

Role: Core detection engine, processing tokenized queries to predict injection likelihood.

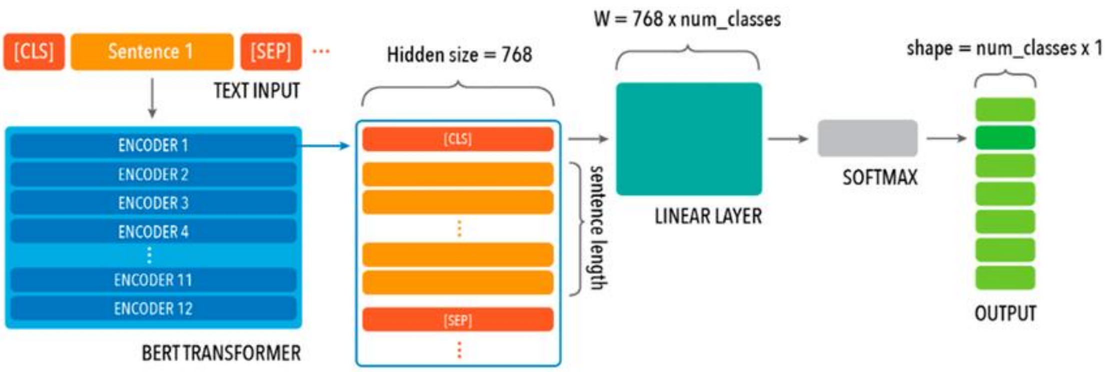


Fig 1 : Bert Base uncased model architecture

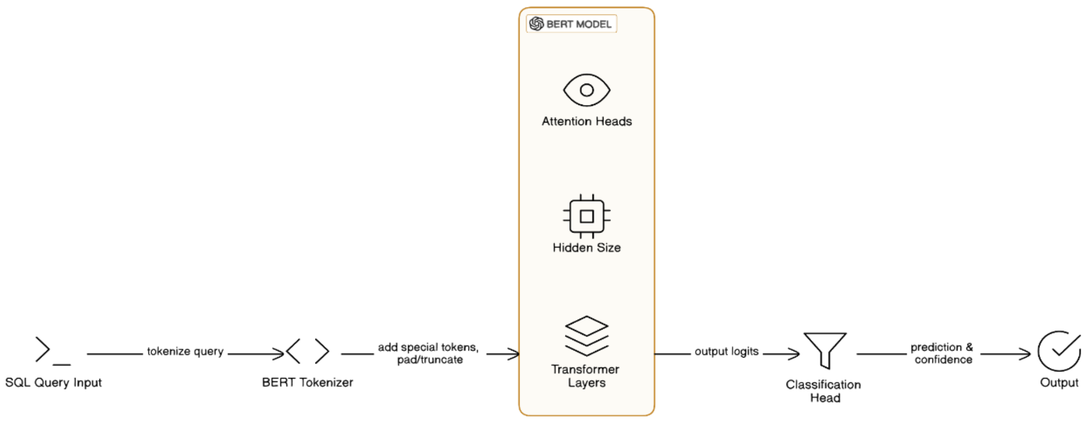


Fig 2 : Pretrained Bert – Based SQLI query Prediction Pipeline

The architecture and Pipeline illustrates BERT's ability to understand query context bidirectionally, enabling accurate classification of SQL injections. The transformer layers capture syntactic and semantic patterns (e.g., "OR 1=1" as malicious), while the classification head outputs risk levels.

2.GAN (Generative Adversarial Network):

Model: Custom-trained GAN with Generator and Discriminator (defined in SQLI Guard codebase).

Purpose: Generates adversarial SQL queries (e.g., “SELECT OR 1=1 --”) for red teaming to test detection robustness.

Architecture: Generator (3 linear layers: 100→256 + ReLU, 256→256 + ReLU, 256→50 + Tanh) maps noise to query tokens; Discriminator (3 linear layers: 50→256 + ReLU, 256→256 + ReLU, 256→1 + Sigmoid) evaluates query authenticity (not used in inference).

Role: Simulates attack scenarios to enhance system resilience.

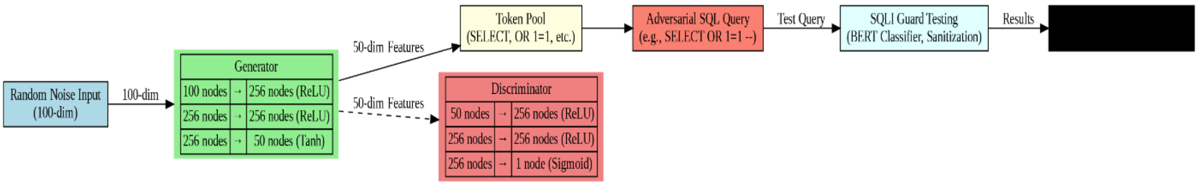


Fig 3 : Generative Adversarial Network architecture for AI Red Teaming

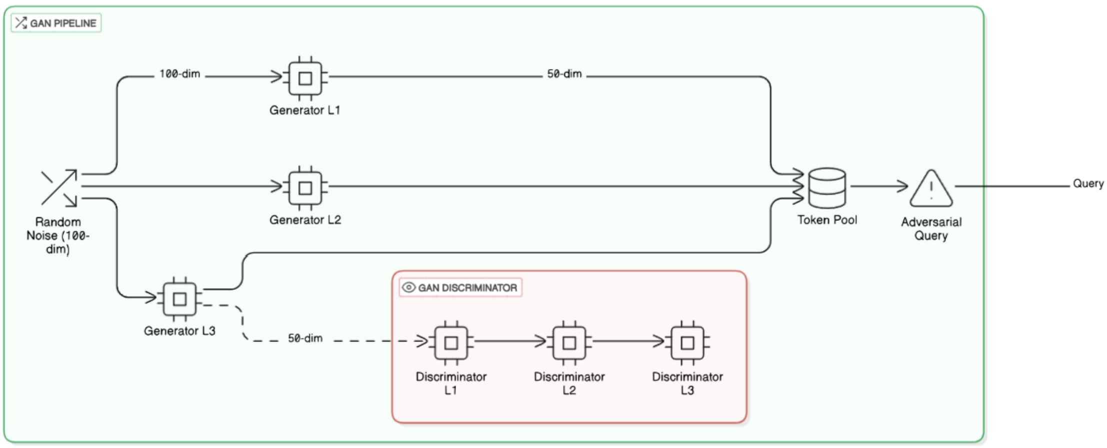


Fig 4 : GAN pipeline for Adversarial Query Generation

The figure illustrates GAN’s ability to generate realistic SQL injection queries. The Generator transforms random noise into query tokens, while the Discriminator (used during training) evaluates authenticity. The diagram emphasizes the Generator’s role in producing adversarial queries for testing and how the GAN generates adversarial queries to simulate SQL injection attacks within SQLI Guard. These queries are fed into the BERT classifier and sanitization modules to test system robustness, with results displayed via the Gradio interface.

3.Stable Diffusion:

Model: CompVis/stable-diffusion-v1-4 (Hugging Face Diffusers).

Purpose: Generates 512x512 images (e.g., comic-strip or cyberpunk styles) to visualize SQL injection attack scenarios.

Architecture: CLIP Text Encoder (24 transformer layers), Variational Autoencoder (VAE), and U-Net (residual blocks, 30 diffusion steps) for text-to-image generation.

Role: Creates visual attack stories for user engagement and reporting.

The architecture of the Stable Diffusion here, used in the code to generate images of SQL injection scenarios, is a latent diffusion model with a variational autoencoder (VAE), U-Net denoising network, and CLIP text encoder. It creates high-quality visuals like comic-strip or cyberpunk dashboards from prompts in generate_attack_story_image. The VAE compresses images into latent space, U-Net refines noisy latents, and CLIP conditions generation on text. It's chosen for its ability to produce detailed, context-relevant images, enhancing user understanding of cybersecurity threats. Loaded with CompVis/stable-diffusion-v1-4, it runs on CPU/GPU with half-precision on GPUs. However, it adds computational overhead and includes a fallback for errors. This integration blends AI analysis with intuitive visual storytelling.

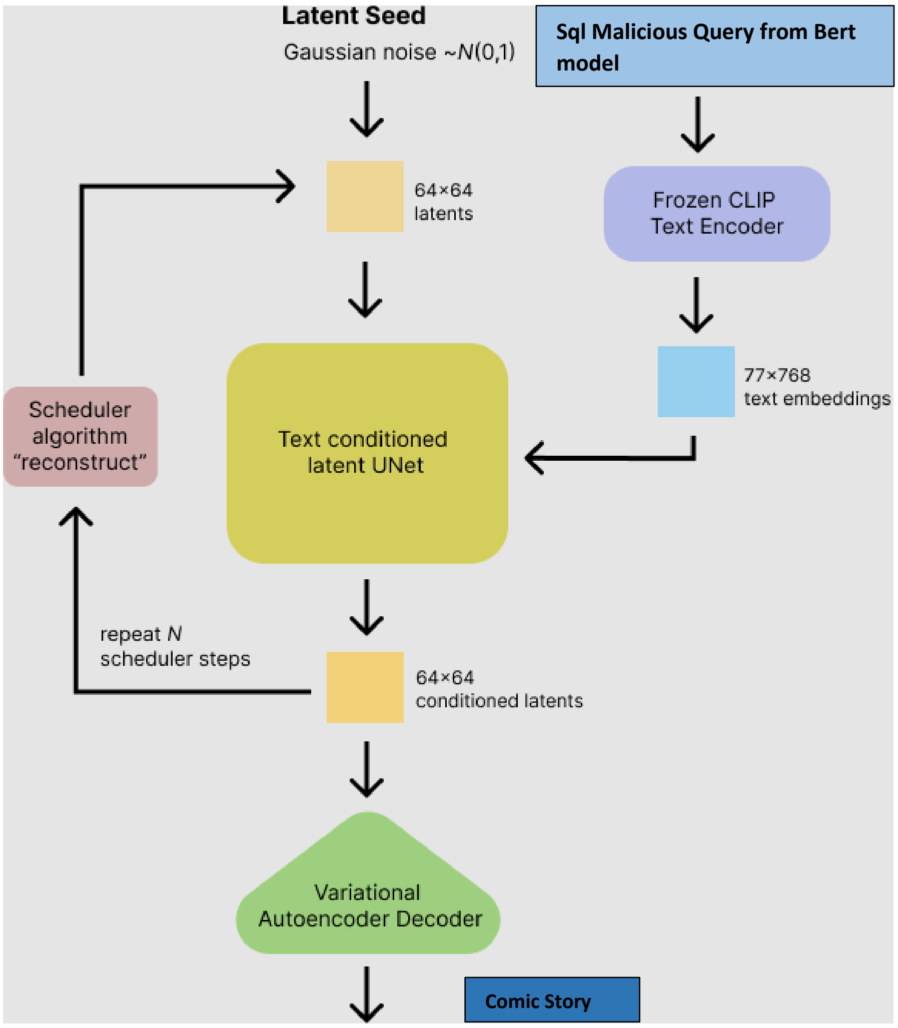


Fig 5 Stable Diffusion Architecture to create attack comic stories

Other Models Used:

4.Whisper (distil-whisper-large-v3-en or whisper-large-v3-turbo):

Purpose: Used for voice-to-SQL transcription in the voice_to_sql function.

Details: Accessed via the Groq API for audio transcription.

Purpose : Supports multiple languages (e.g., English with distil-whisper-large-v3-en, others with whisper-large-v3-turbo). Transcribes audio input to text, which is then converted to secure SQL.

The code integrates Text-to-Speech (TTS) and Speech-to-Text (STS) functionalities to enhance user interaction with SQL injection analysis. TTS is implemented in the text_to_speech function using the gTTS library, which converts AI-generated explanations (from generate_auto_explanation) into audio files (explanation.mp3). It processes markdown text, removes special characters, and generates English speech, improving accessibility for auditory learners, though it's limited to English and may fail with complex formatting. STS is used in the voice_to_sql function, leveraging the Groq API with models like distil-whisper-large-v3-en or whisper-large-v3-turbo to transcribe audio inputs into text, which is then converted to secure SQL queries. This enables voice-based query input, enhancing usability, but requires valid audio files and is sensitive to language settings. Both features, powered by external APIs (gTTS and Groq), add interactivity but introduce dependencies and potential error points, with fallbacks for failure cases. They align with the code's goal of providing multimodal interfaces for cybersecurity analysis.

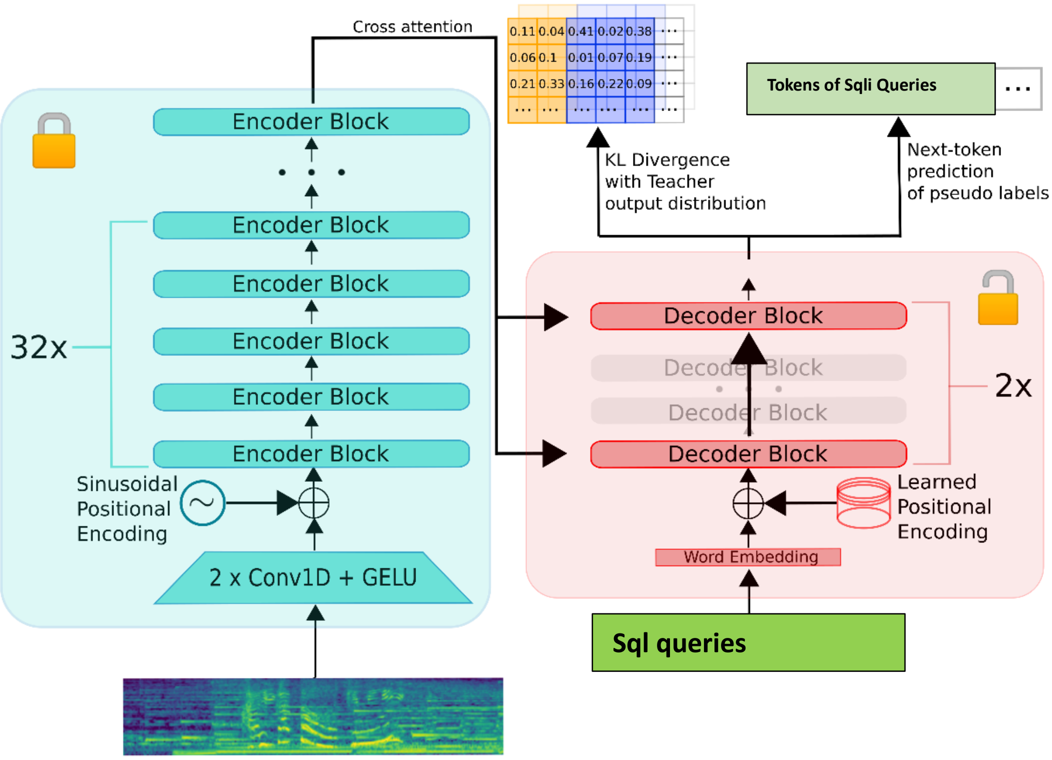


Fig 6 distil-whisper-large-v3-en (text to speech architecture)

In which the above illustrates the text to speech consists of auto explanation engine and the query sanitization explanation using the Gtts library for the sanitization flow of the queries.

SYSTEM ARCHITECTURE :

The architecture diagram illustrates a comprehensive system for SQL injection detection and visualization, integrating multiple AI pipelines and modules. The BERT Pipeline processes SQL queries by tokenizing them and passing them through a transformer-based classification model to generate predictions, though its effectiveness is limited without fine-tuning for SQL injection detection. The GAN Pipeline generates adversarial queries using a Generator (with 100-dimensional noise input and three hidden layers) and a Discriminator (with 50-dimensional input and three layers), feeding into an adversarial query pool to enhance model robustness, though it's underutilized in the current setup. The Stable Diffusion Pipeline employs a text encoder (CLIP), a U-Net, and a VAE to transform text prompts (e.g., attack story descriptions) into images, leveraging system modules for image generation and output. The System Modules section includes Sanitization (removing malicious patterns), Pattern Detection (using COMMUNITY_PATTERNS for regex-based identification), Visualizations (generating plots like heatmaps), and Honeypot (simulating attacks), all feeding into a Gradio interface for user interaction. Logs capture system activity, while sanitized queries, patterns, plots, and results are outputted, creating a multimodal system for analysis, visualization, and user feedback, though its reliance on pretrained models limits predictive accuracy without further training.

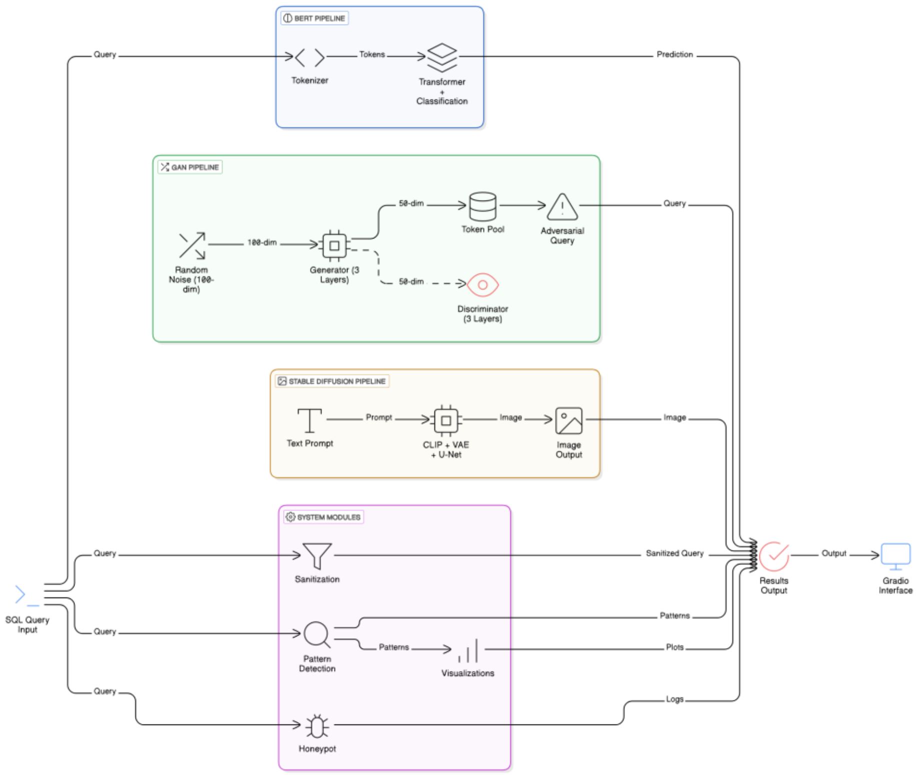


Fig 5 : System Architecture of SQLI – Guard System

SAMPLE CODE:

SQL Injection Predictor :

```
def predict_sqli(query):
    if not query or not query.strip():
        return "Safe", {"Safe": 1.0, "Suspicious": 0.0, "Malicious": 0.0}
    inputs = bert_tokenizer(query, return_tensors="pt", truncation=True,
padding=True).to(DEVICE)
    with torch.no_grad():
        outputs = bert_model(**inputs)
    logits = outputs.logits
    probs = torch.nn.functional.softmax(logits, dim=-1)
    pred_class = torch.argmax(probs).item()
    return CLASS_NAMES[pred_class], {CLASS_NAMES[i]: float(probs[0][i]) for i
in range(3)}
```

Model Loader:

```
def load_models():
    bert_tokenizer = BertTokenizer.from_pretrained(MODEL_NAME)
    bert_model = BertForSequenceClassification.from_pretrained(MODEL_NAME,
num_labels=3)
    bert_model.to(DEVICE)
    return bert_tokenizer, bert_model
bert_tokenizer, bert_model = load_models()
```

Pattern Visualizer (3D Scatter):

```
def visualize_pattern_bar(query):
    if not query or not query.strip():
        return None
    tokens = [t for t in re.findall(r"[^'\"\s]+|'[^']*'|" + r'"[^"]*"', query) if t]
    token_types = ["Keyword" if t.upper() in ["SELECT", "FROM"] else "String" if "'"
in t else "Identifier" for t in tokens]
    colors = ["#00B7EB" if t == "Keyword" else "#00FF9F" if t == "String" else
"#FFFFFF" for t in token_types]
    frames = [go.Frame(data=[go.Scatter3d(x=[i], y=[0], z=[0], mode="markers+text",
text=[t], marker=dict(size=12, color=c))], name=f"frame{i}") for i, (t, c) in
enumerate(zip(tokens, colors))]
    fig = go.Figure(data=[go.Scatter3d(x=[0], y=[0], z=[0], mode="markers+text",
text=[tokens[0]], marker=dict(size=12, color=colors[0]))], frames=frames)
    fig.update_layout(title="SQL Query Token Analysis (3D)",
scene=dict(bgcolor="#1E1E1E"), plot_bgcolor="rgba(30,30,30,0.95)")
    return fig
```

Mutation Visualizer (Line Plot):

```
def visualize_mutation_sunburst(query):
    logger.debug(f"Starting visualize_mutation_sunburst with query: {query}")
    labels = ["Original", "Mutation 1", "Mutation 2"]
    risk_scores = [10, 20, 30]
    logger.debug(f"Using placeholder data - Labels: {labels}, Risk Scores:
{risk_scores}")
    fig = go.Figure(data=[go.Scatter(x=labels, y=risk_scores, mode="lines+markers",
marker=dict(size=8, color="red"))])
    fig.update_layout(title="Mutation Risk Scores", xaxis=dict(title="Query"),
yaxis=dict(title="Risk Score (%)"), height=300)
    return fig
```

Threat Impact Visualizer :

```
def visualize_threat_impact(query):
    if not query or not query.strip():
        return None
    severity_counts = {"low": 0.2, "medium": 0.3, "high": 0.5, "critical": 0.6}
    for pat_name, pat_info in COMMUNITY_PATTERNS.items():
        if re.search(pat_info["pattern"], query, re.IGNORECASE):
            severity_counts[pat_info["severity"]] += 1
    fig = go.Figure(data=[go.Scatter(x=list(severity_counts.keys()),
y=list(severity_counts.values()), mode="markers+text",
text=list(severity_counts.values()))])
    fig.update_layout(title="Threat Severity Distribution", xaxis=dict(title="Severity"),
yaxis=dict(title="Count"), height=400)
    return fig
```

SQL Heatmap Visualizer:

```
def visualize_sqli_heatmap(query):
    if not query or not query.strip():
        return None
    tokens = query.split()
    risk_scores = [1.0 if any(p in t.upper() for p in ["'", "--", "UNION"]) else 0.1 for t in
tokens]
    frames = [go.Frame(data=[go.Heatmap(z=[risk_scores[:i+1]], x=tokens)],
name=f"frame{i}") for i in range(len(tokens))]
    fig = go.Figure(data=[go.Heatmap(z=[risk_scores], x=tokens)], frames=frames)
    fig.update_layout(title="SQL Query Heatmap (Injection Risk)", height=450,
xaxis=dict(tickangle=45))
    return fig
```

Query Sanitizer:

```
def sanitize_query(query, db_type="generic"):
    if not query or not query.strip():
        return "", "No query provided", ""
    sanitized = query
    replacements = []
    for pattern, info in {**{"'": {"replacement": "''"}, "--": {"replacement": ""}},
**COMMUNITY_PATTERNS}.items():
        if isinstance(pattern, str) and pattern in sanitized:
            sanitized = sanitized.replace(pattern, info["replacement"])
            replacements.append((pattern, info["desc"]))
    explanation = f"**Sanitization Report**\nOriginal: {query}\nSanitized patterns:\n"
+ "\n".join(f"- {p}: {d}" for p, d in replacements)
    return sanitized, explanation, generate_parameterized_query(query, db_type)
```

Parameterized Query Generator:

```
def generate_parameterized_query(query, db_type="generic"):
    tokens = query.split()
    param_query = ["?" if any(c in t for c in ["'", '"']) else t for t in tokens]
    parameterized = " ".join(param_query)
    if db_type == "mysql":
        parameterized = parameterized.replace("?", "%s")
    return f"**Parameterized Query**: {parameterized}"
```

Attack DNA Extractor:

```
def extract_attack_dna(query):
    if not query or not query.strip():
        return None, None, None
    signature = hashlib.sha256(query.encode()).hexdigest()[:16]
    patterns = {"union": {"active": "UNION" in query.upper(), "severity": "high"}}
    active_patterns = [p for p, info in patterns.items() if info["active"]]
    G = nx.DiGraph()
    G.add_edges_from([("Query Signature", p) for p in active_patterns])
    pos = nx.spring_layout(G)
    node_trace = go.Scatter(x=[pos[n][0] for n in G.nodes()], y=[pos[n][1] for n in
G.nodes()], mode="markers+text", text=list(G.nodes()))
    fig = go.Figure(data=[node_trace])
    fig.update_layout(title="Attack DNA Network Graph", height=400)
    return signature, active_patterns, fig
```

Attack Story Image Generator:

```
def generate_attack_story_image(query, style="comic-strip"):
    if not query or not query.strip():
        return None
    prompt = f"A {style} of a cybersecurity dashboard detecting SQL injection with
'{query[:50]}...' and a red warning."
    pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-
4").to(DEVICE)
    image = pipe(prompt=prompt, num_inference_steps=30).images[0] if
torch.cuda.is_available() else Image.new('RGB', (512, 512), 'black')
    return image
```

Attack Story Creator:

```
def generate_attack_story(query):
    if not query or not query.strip():
        return None
    image = generate_attack_story_image(query)
    return image
```

Auto-Explanation Generator:

```
def generate_auto_explanation(query):
    pred, conf = predict_sqli(query)
    detected = ["tautology" if "OR 1=1" in query.upper() else "none"]
    prompt = f"Explain why '{query}' was classified as {pred}. Detected: {',
'.join(detected)}. Confidence: {conf}."
    response = groq_client.chat.completions.create(model="llama-3.3-70b-versatile",
messages=[{"role": "user", "content": prompt}], max_tokens=300)
    return response.choices[0].message.content.strip()
```

Text-to-Speech Converter:

```
def text_to_speech(text):
    try:
        clean_text = re.sub(r'[#*_-]', '', text)
        tts = gTTS(text=clean_text, lang='en')
        audio_file = "explanation.mp3"
        tts.save(audio_file)
        return audio_file
    except Exception as e:
        logger.error(f"Error generating speech: {str(e)}")
        return None
```

Natural Language to Secure SQL Converter:

```
def nl_to_secure_sql(nl_query):
    if not nl_query or not nl_query.strip():
        return "", None, "No input provided"
    prompt = f"Convert '{nl_query}' to a secure SQL query. Return: 1. Query 2.
Explanation###"
    response = groq_client.chat.completions.create(model="llama-3.3-70b-versatile",
messages=[{"role": "user", "content": prompt}], max_tokens=300)
    parts = response.choices[0].message.content.split("###")
    secure_query = parts[0].strip() if len(parts) > 0 else ""
    explanation = parts[1].strip() if len(parts) > 1 else ""
    return secure_query, None, explanation
```

Query Execution Timeline Visualizer:

```
def query_execution_timeline(query):
    if not query or not query.strip():
        return None
    stages = [{"stage": "Parsing", "start": 0, "duration": 2}, {"stage": "Execution",
"start": 2, "duration": 3}]
    fig = go.Figure([go.Bar(x=[s["duration"]], y=[s["stage"]], base=[s["start"]],
orientation="h") for s in stages])
    fig.update_layout(title="Query Execution Timeline", xaxis=dict(title="Time
(seconds)"), height=400)
    return fig
```

Threat Persona Generator:

```
def generate_threat_persona(query):
    if not query or not query.strip():
        return "Unknown", 0.0, "No query provided"
    complexity_score = 0.5 if "UNION" in query.upper() else 0.1
    persona = "Botnet Scanner" if complexity_score > 0.3 else "Script Kiddie"
    return persona, complexity_score, f"Persona: {persona}, Score:
{complexity_score}"
```

Browser Honeypot Simulator:

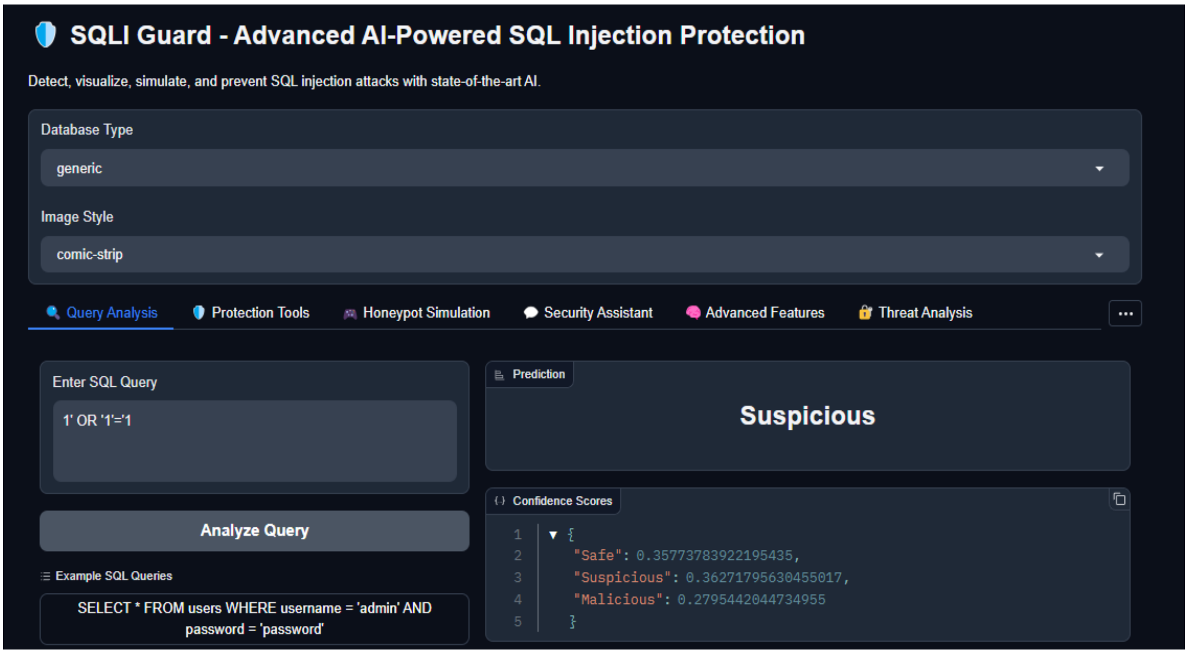
```
def honeypot_browser_simulation(input_query):
    if not input_query or not input_query.strip():
        return None, "No input provided"
    pred, conf = predict_sqli(input_query)
```

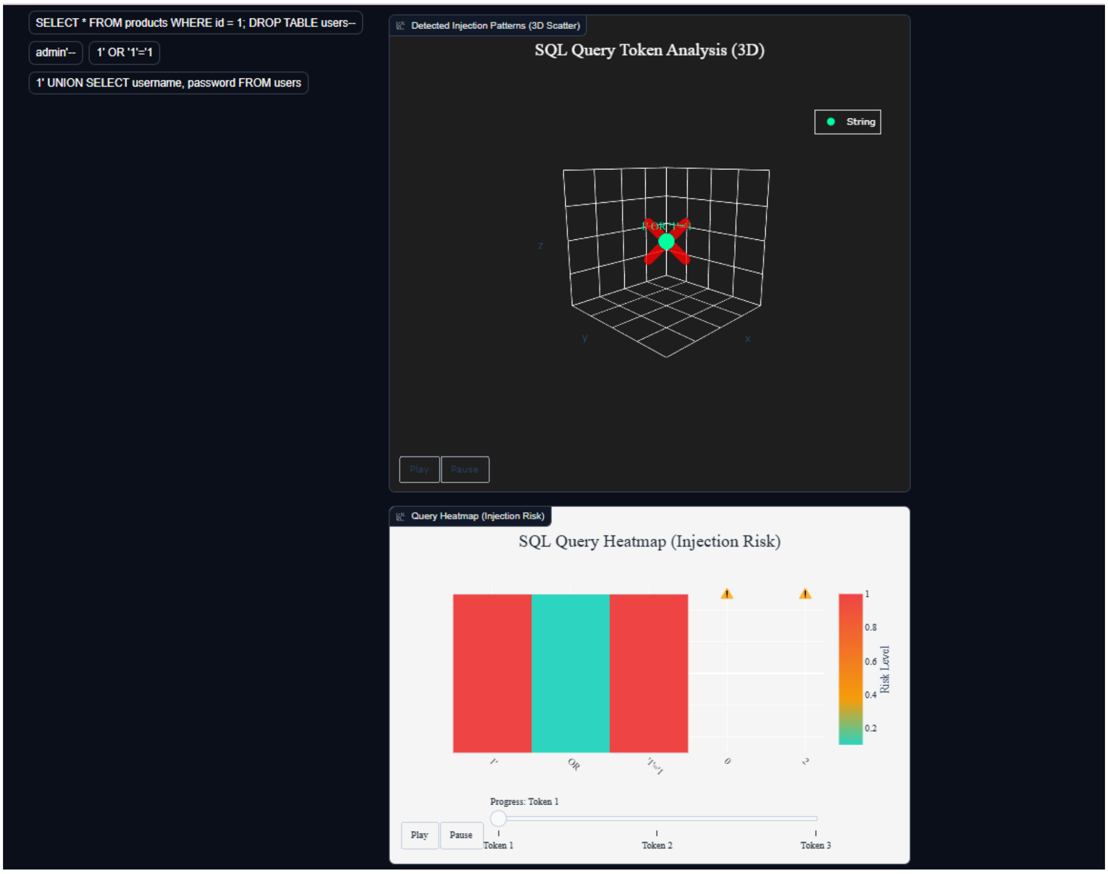
```
    fig = go.Figure([go.Pie(labels=list(conf.keys()), values=list(conf.values()))])
    fig.update_layout(title="Honeypot Confidence Scores", height=300)
    return fig, f"Classified as {pred}"
```

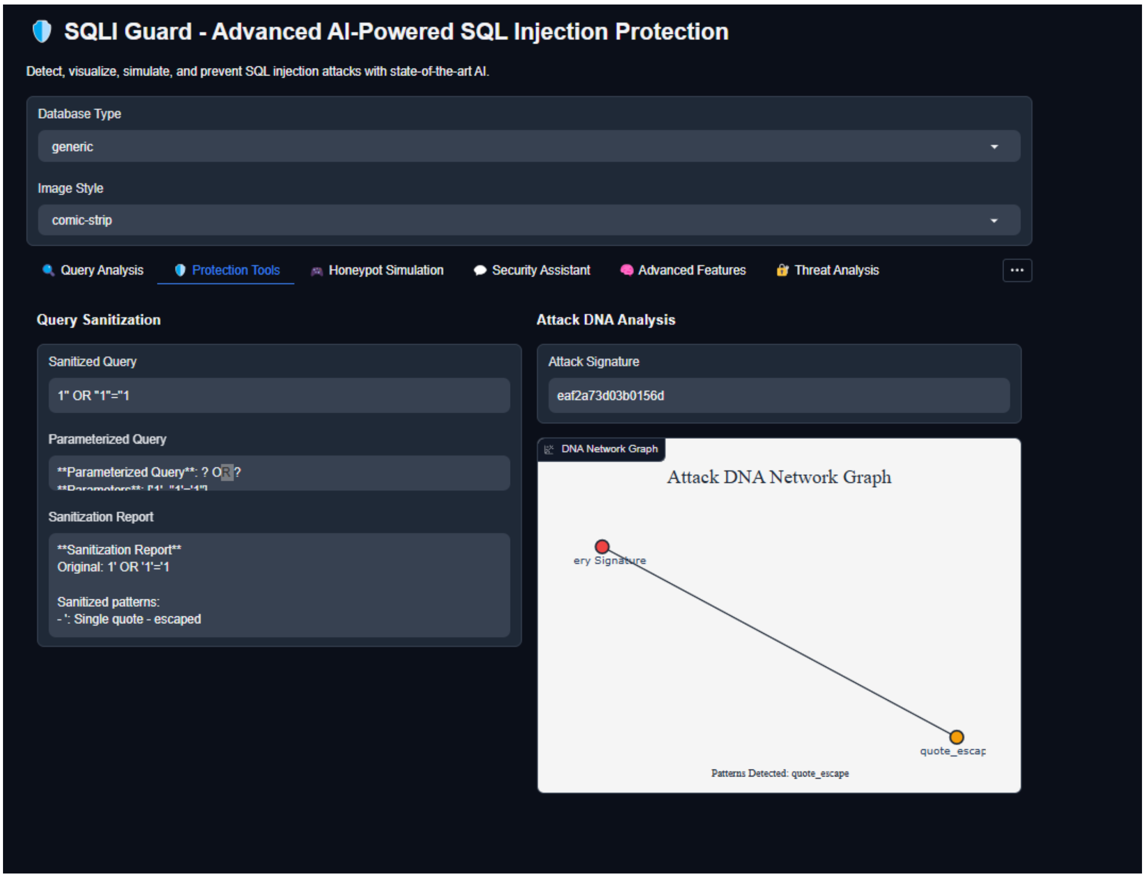
Honeypot Simulator:

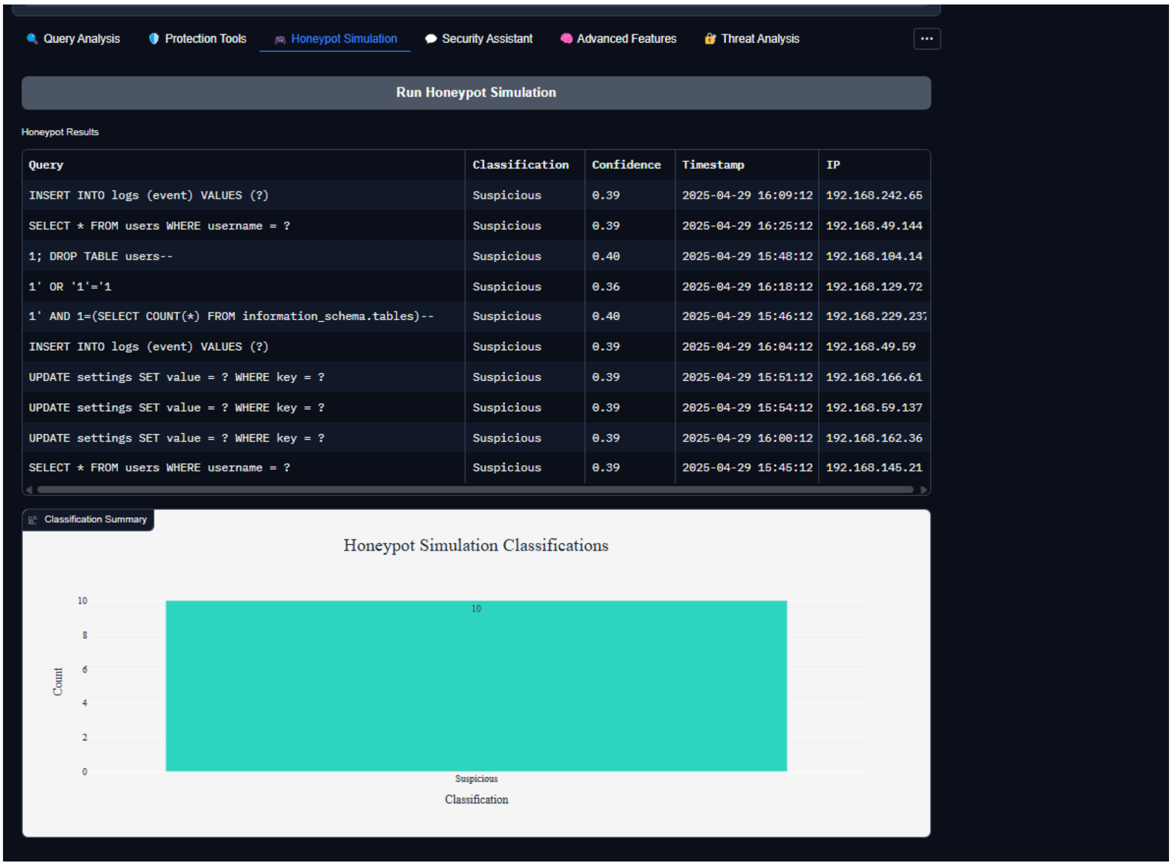
```
def honeypot_simulation():
    safe_patterns = ["SELECT * FROM users WHERE username = ?"]
    malicious_patterns = ["1' OR '1'='1"]
    results = [{"Query": random.choice(malicious_patterns if random.random() < 0.4
else safe_patterns), "Classification": predict_sqli()[0]} for _ in range(10)]
    df = pd.DataFrame(results)
    fig = go.Figure([go.Bar(x=df["Classification"].value_counts().index,
y=df["Classification"].value_counts())])
    fig.update_layout(title="Honeypot Simulation Classifications", height=400)
    return df, fig
```

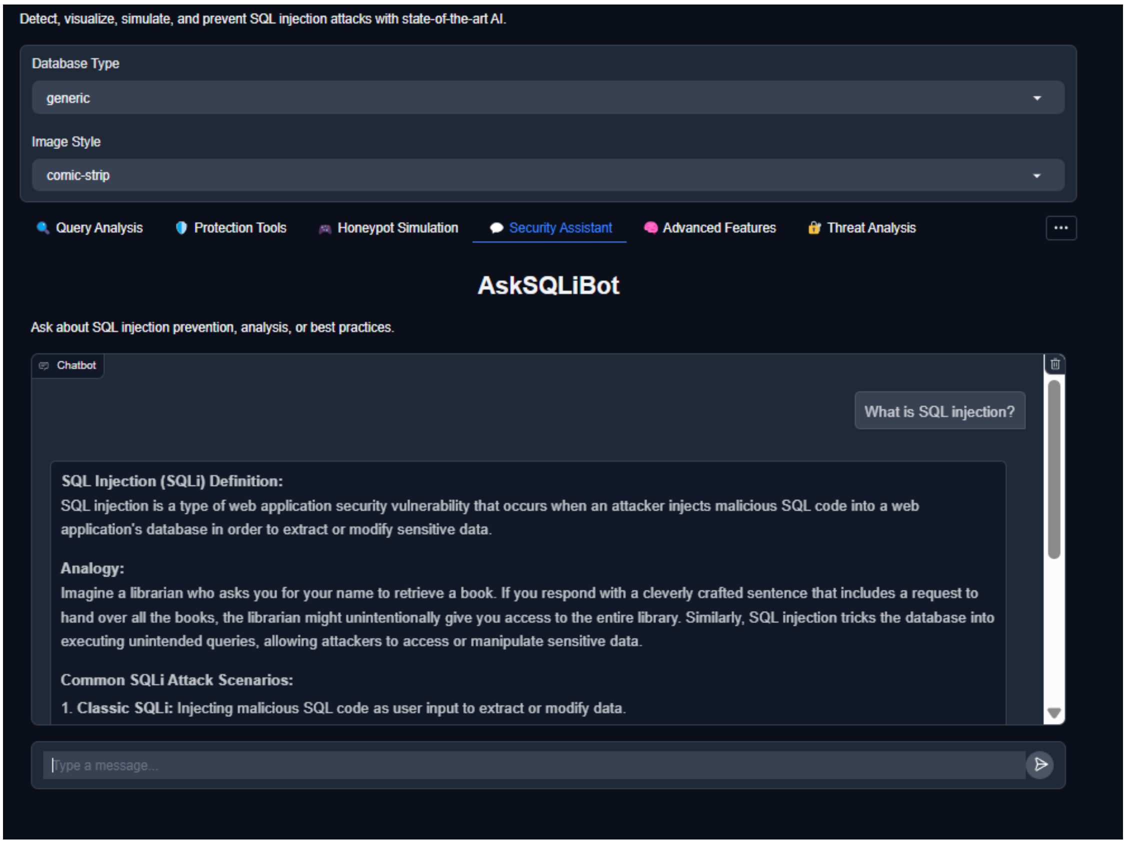
OUTPUT:

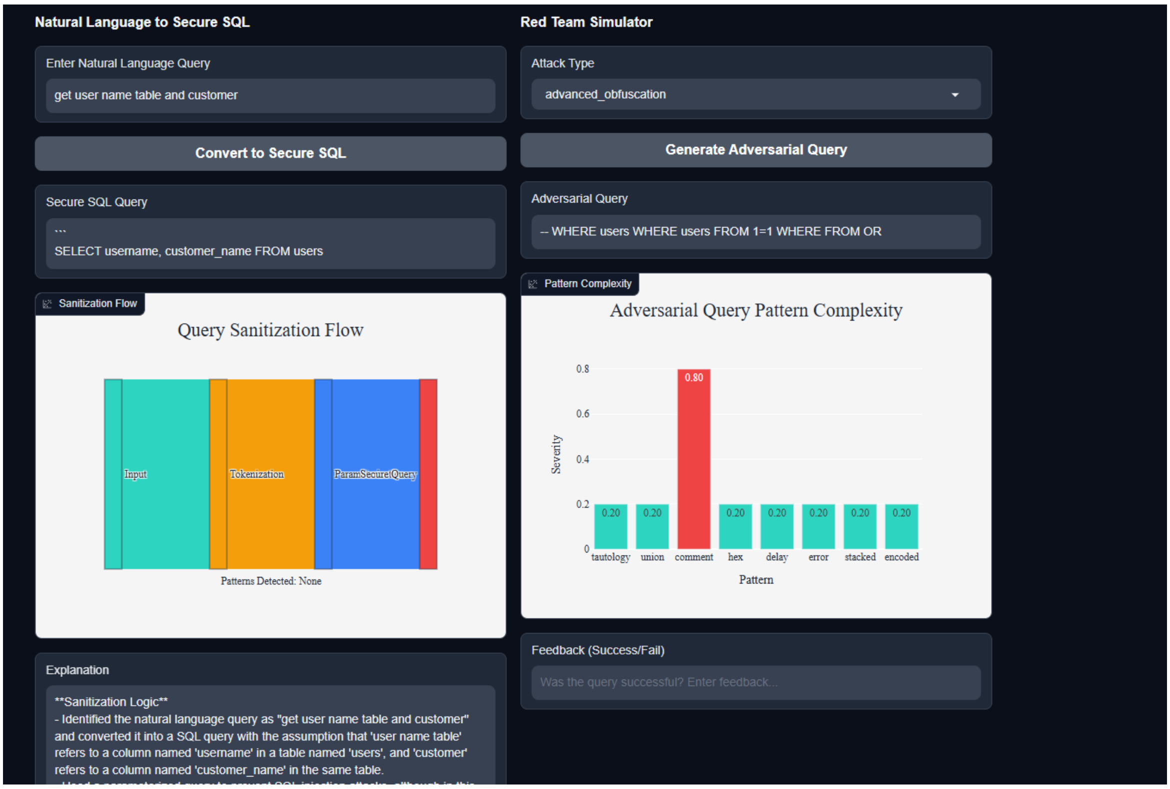


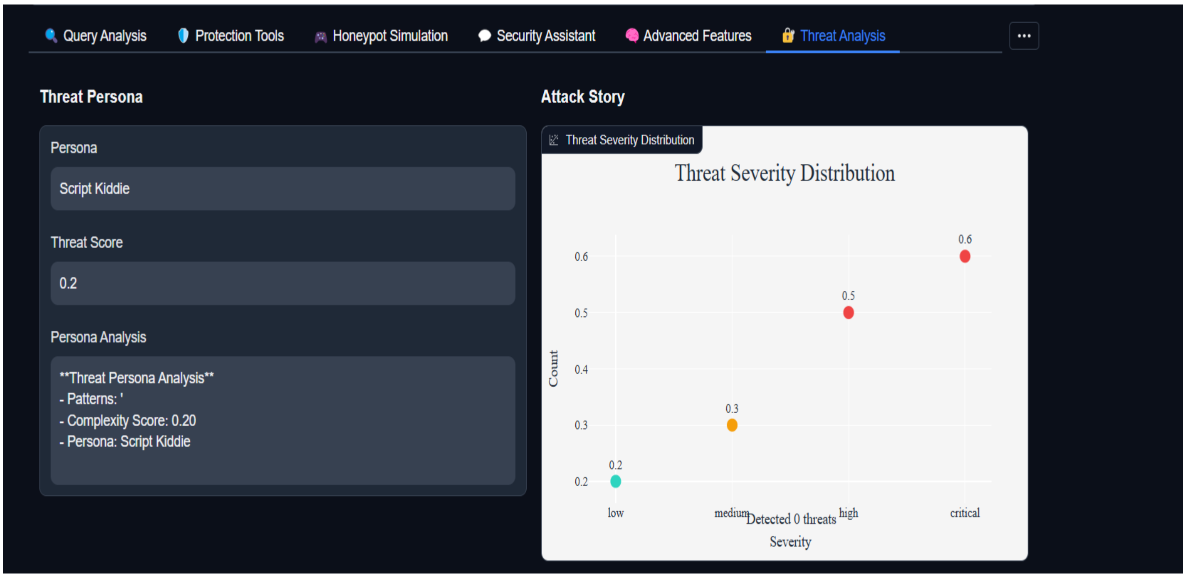

SELECT * FROM products WHERE id = 1; DROP TABLE users--
admin'--
1' OR '1'='1
1' UNION SELECT username, password FROM users
Detected Injection Patterns (3D Scatter)
SQL Query Token Analysis (3D)
String
Play
Pause
Query Heatmap (Injection Risk)
SQL Query Heatmap (Injection Risk)
Risk Level
Progress: Token 1
Play
Pause
Token 1
Token 2
Token 3


SQLI Guard - Advanced AI-Powered SQL Injection Protection
Detect, visualize, simulate, and prevent SQL injection attacks with state-of-the-art AI.
Database Type
generic
Image Style
comic-strip
Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
Query Sanitization
Sanitized Query
1" OR "1"="1
Parameterized Query
Parameterized Query: ? OR ?
Sanitization Report
Sanitization Report
Original: 1' OR '1'='1
Sanitized patterns:
- ': Single quote - escaped
Attack DNA Analysis
Attack Signature
eaf2a73d03b0156d
DNA Network Graph
Attack DNA Network Graph
ery Signature
quote_escap
Patterns Detected: quote_escape


Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
Run Honeypot Simulation
Honeypot Results
Query | Classification | Confidence | Timestamp | IP
INSERT INTO logs (event) VALUES (?) | Suspicious | 0.39 | 2025-04-29 16:09:12 | 192.168.242.65
SELECT * FROM users WHERE username = ? | Suspicious | 0.39 | 2025-04-29 16:25:12 | 192.168.49.144
1; DROP TABLE users-- | Suspicious | 0.40 | 2025-04-29 15:48:12 | 192.168.104.14
1' OR '1'='1 | Suspicious | 0.36 | 2025-04-29 16:18:12 | 192.168.129.72
1' AND 1=(SELECT COUNT(*) FROM information_schema.tables)-- | Suspicious | 0.40 | 2025-04-29 15:46:12 | 192.168.229.23
INSERT INTO logs (event) VALUES (?) | Suspicious | 0.39 | 2025-04-29 16:04:12 | 192.168.49.59
UPDATE settings SET value = ? WHERE key = ? | Suspicious | 0.39 | 2025-04-29 15:51:12 | 192.168.166.61
UPDATE settings SET value = ? WHERE key = ? | Suspicious | 0.39 | 2025-04-29 15:54:12 | 192.168.59.137
UPDATE settings SET value = ? WHERE key = ? | Suspicious | 0.39 | 2025-04-29 16:00:12 | 192.168.162.36
SELECT * FROM users WHERE username = ? | Suspicious | 0.39 | 2025-04-29 15:45:12 | 192.168.145.21
Classification Summary
Honeypot Simulation Classifications
Count
10
8
6
4
2
0
10
Suspicious
Classification

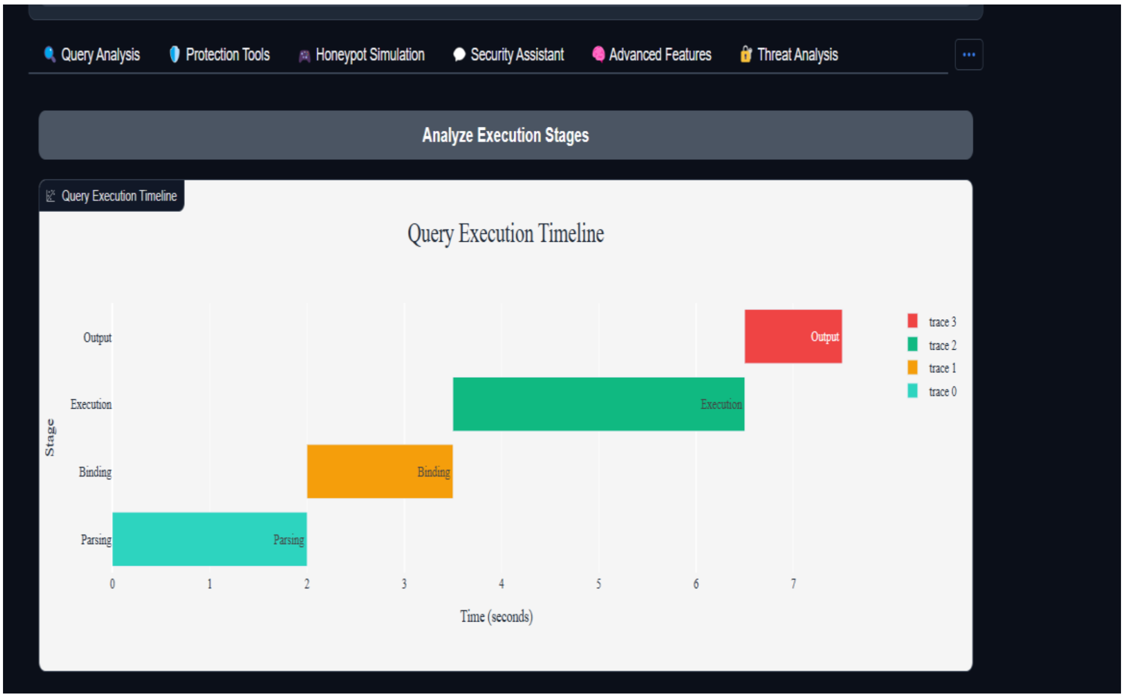

Detect, visualize, simulate, and prevent SQL injection attacks with state-of-the-art AI.
Database Type
generic
Image Style
comic-strip
Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
AskSQLiBot
Ask about SQL injection prevention, analysis, or best practices.
Chatbot
What is SQL injection?
SQL Injection (SQLi) Definition:
SQL injection is a type of web application security vulnerability that occurs when an attacker injects malicious SQL code into a web application's database in order to extract or modify sensitive data.
Analogy:
Imagine a librarian who asks you for your name to retrieve a book. If you respond with a cleverly crafted sentence that includes a request to hand over all the books, the librarian might unintentionally give you access to the entire library. Similarly, SQL injection tricks the database into executing unintended queries, allowing attackers to access or manipulate sensitive data.
Common SQLi Attack Scenarios:
1. Classic SQLi: Injecting malicious SQL code as user input to extract or modify data.
Type a message...

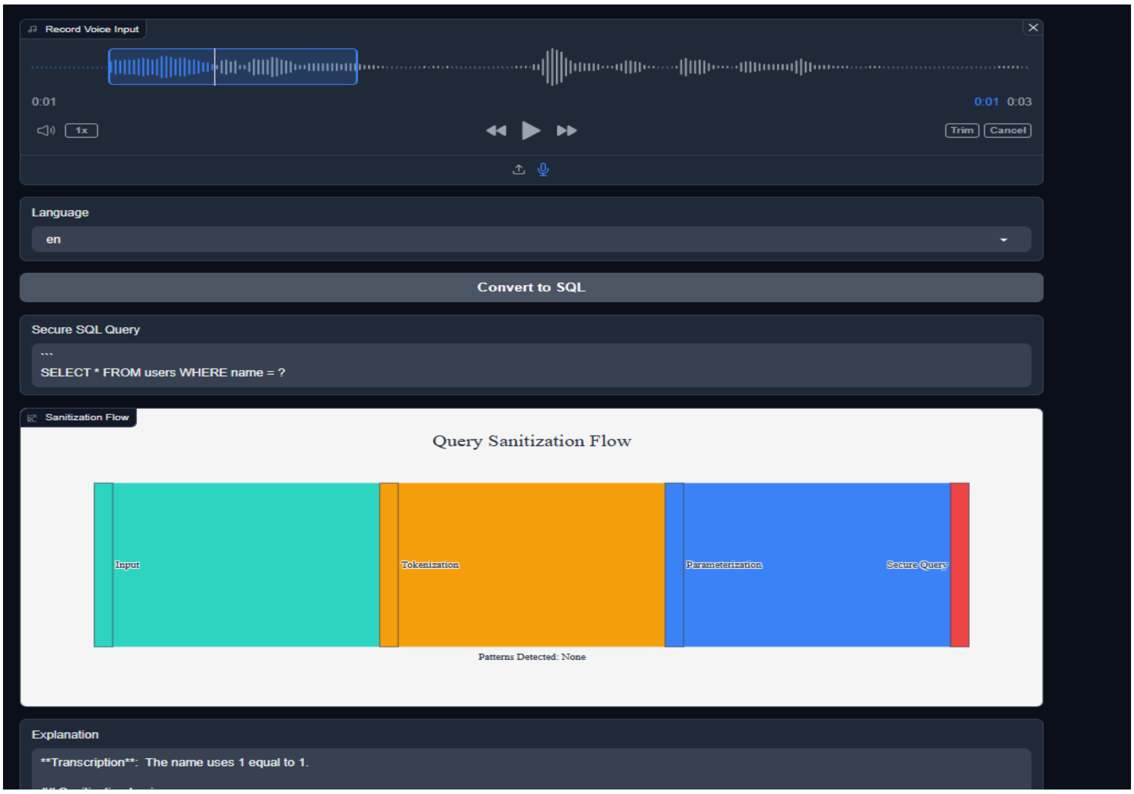

Natural Language to Secure SQL
Enter Natural Language Query
get user name table and customer
Convert to Secure SQL
Secure SQL Query
SELECT username, customer_name FROM users
Sanitization Flow
Query Sanitization Flow
Input
Tokenization
ParamSecureQuery
Patterns Detected: None
Explanation
Sanitization Logic
- Identified the natural language query as "get user name table and customer" and converted it into a SQL query with the assumption that 'user name table' refers to a column named 'username' in a table named 'users', and 'customer' refers to a column named 'customer_name' in the same table.
Red Team Simulator
Attack Type
advanced_obfuscation
Generate Adversarial Query
Adversarial Query
-- WHERE users WHERE users FROM 1=1 WHERE FROM OR
Pattern Complexity
Adversarial Query Pattern Complexity
Severity
Pattern
tautology
union
comment
hex
delay
error
stacked
encoded
0.80
0.20
Feedback (Success/Fail)
Was the query successful? Enter feedback...

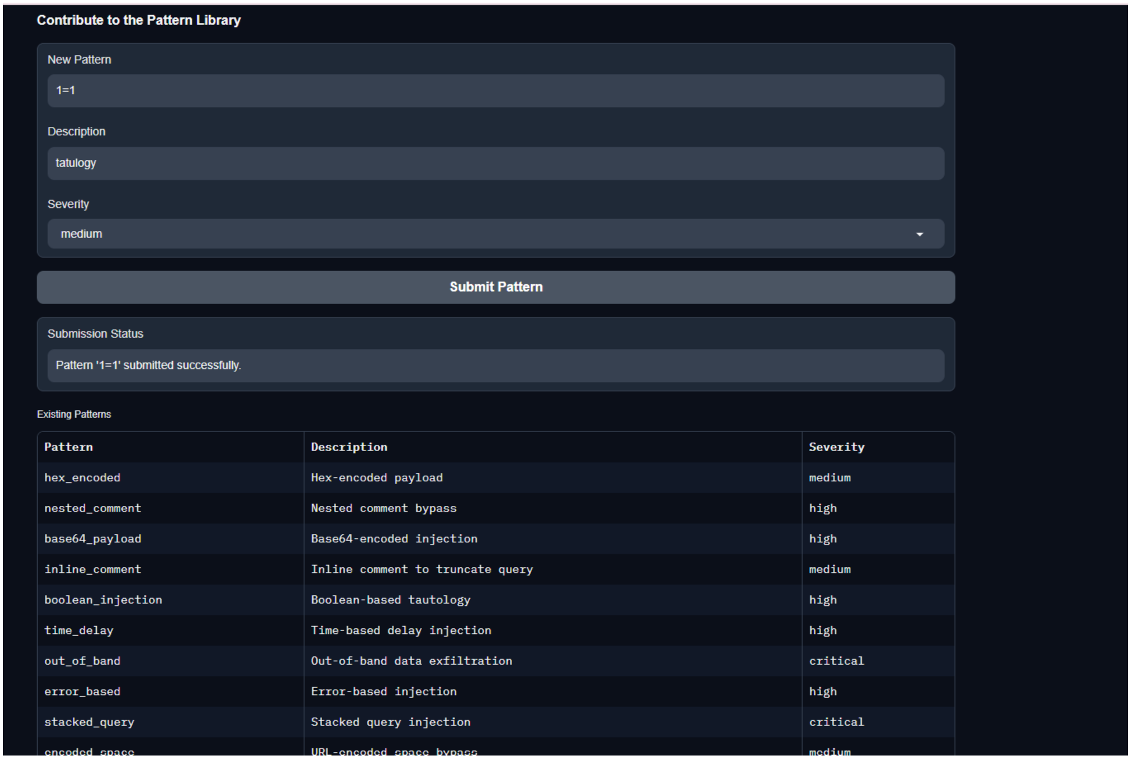

Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
Threat Persona
Persona
Script Kiddie
Threat Score
0.2
Persona Analysis
Threat Persona Analysis
- Patterns: '
- Complexity Score: 0.20
- Persona: Script Kiddie
Attack Story
Threat Severity Distribution
Threat Severity Distribution
Count
Severity
low
medium
high
critical
Detected 0 threats

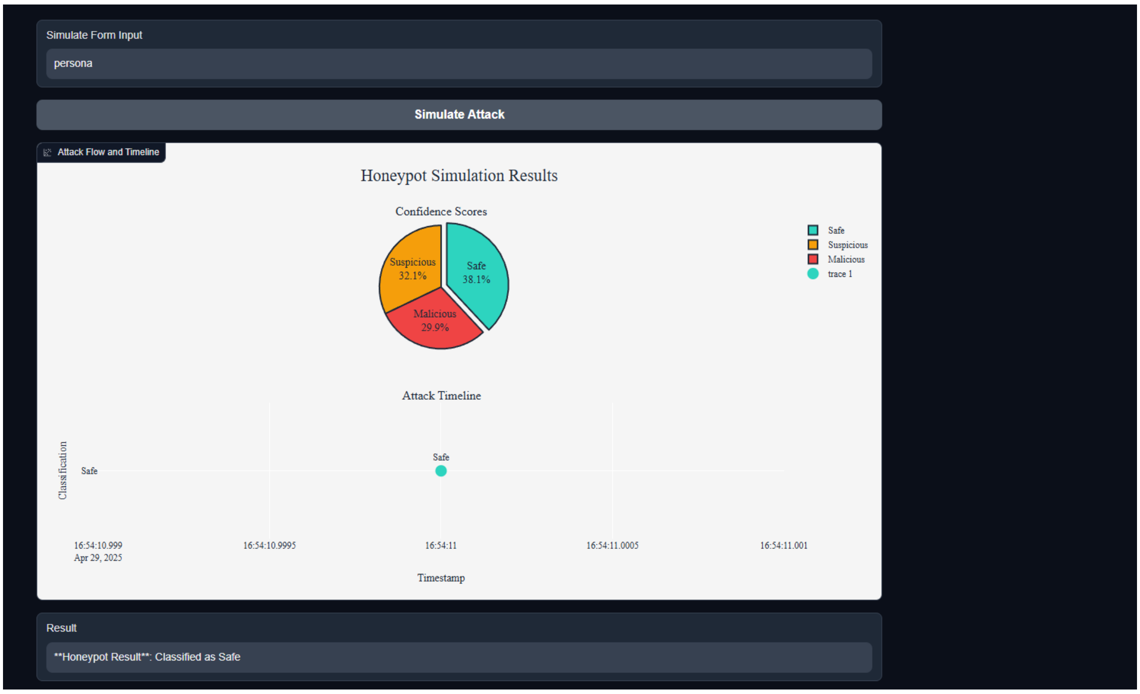

Attack Story Comic (Stable Diffusion)


Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
AI-Generated Explanation
Explanation
Query Analysis: '1' OR '1'='1'
The query `'1' OR '1'='1'` has been classified as Suspicious due to its potential to be used in SQL injection attacks. Here's a breakdown:
* The query uses a logical OR operator, which can be used to manipulate the conditions of a SQL query.
* The condition `'1'='1'` is always true, which can be used to bypass authentication or authorization mechanisms.
Listen to Explanation
0:00
2:06
1x
Use via API · Built with Gradio · Settings


Query Analysis
Protection Tools
Honeypot Simulation
Security Assistant
Advanced Features
Threat Analysis
Analyze Execution Stages
Query Execution Timeline
Query Execution Timeline
Output
Execution
Binding
Parsing
Stage
Time (seconds)
trace 3
trace 2
trace 1
trace 0


Record Voice Input
0:01
0:01 0:03
1x
Trim
Cancel
Language
en
Convert to SQL
Secure SQL Query
SELECT * FROM users WHERE name = ?
Sanitization Flow
Query Sanitization Flow
Input
Tokenization
Parameterization
Secure Query
Patterns Detected: None
Explanation
Transcription: The name uses 1 equal to 1.


Contribute to the Pattern Library
New Pattern
1=1
Description
tatulogy
Severity
medium
Submit Pattern
Submission Status
Pattern '1=1' submitted successfully.
Existing Patterns
Pattern
Description
Severity
hex_encoded
Hex-encoded payload
medium
nested_comment
Nested comment bypass
high
base64_payload
Base64-encoded injection
high
inline_comment
Inline comment to truncate query
medium
boolean_injection
Boolean-based tautology
high
time_delay
Time-based delay injection
high
out_of_band
Out-of-band data exfiltration
critical
error_based
Error-based injection
high
stacked_query
Stacked query injection
critical


Simulate Form Input
persona
Simulate Attack
Attack Flow and Timeline
Honeypot Simulation Results
Confidence Scores
Suspicious 32.1%
Safe 38.1%
Malicious 29.9%
Safe
Suspicious
Malicious
trace 1
Attack Timeline
Classification
Safe
Safe
16:54:10.999
Apr 29, 2025
16:54:10.9995
16:54:11
16:54:11.0005
16:54:11.001
Timestamp
Result
Honeypot Result: Classified as Safe