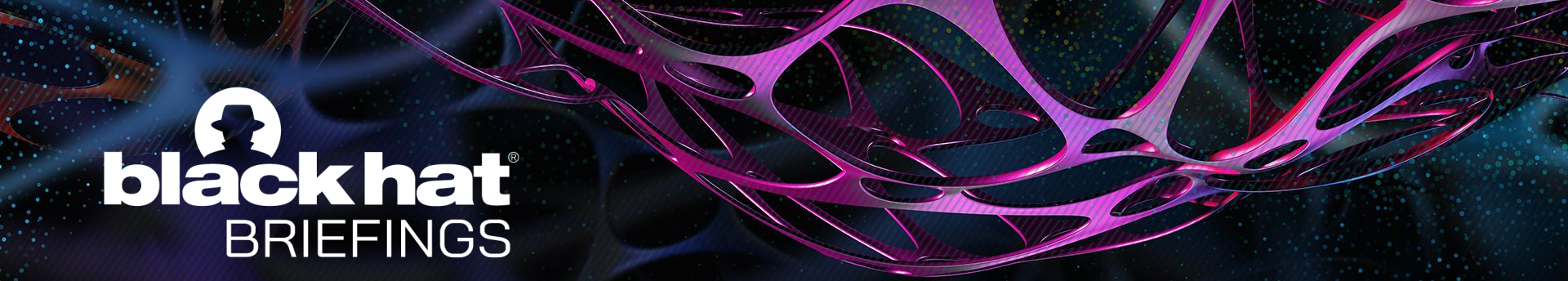


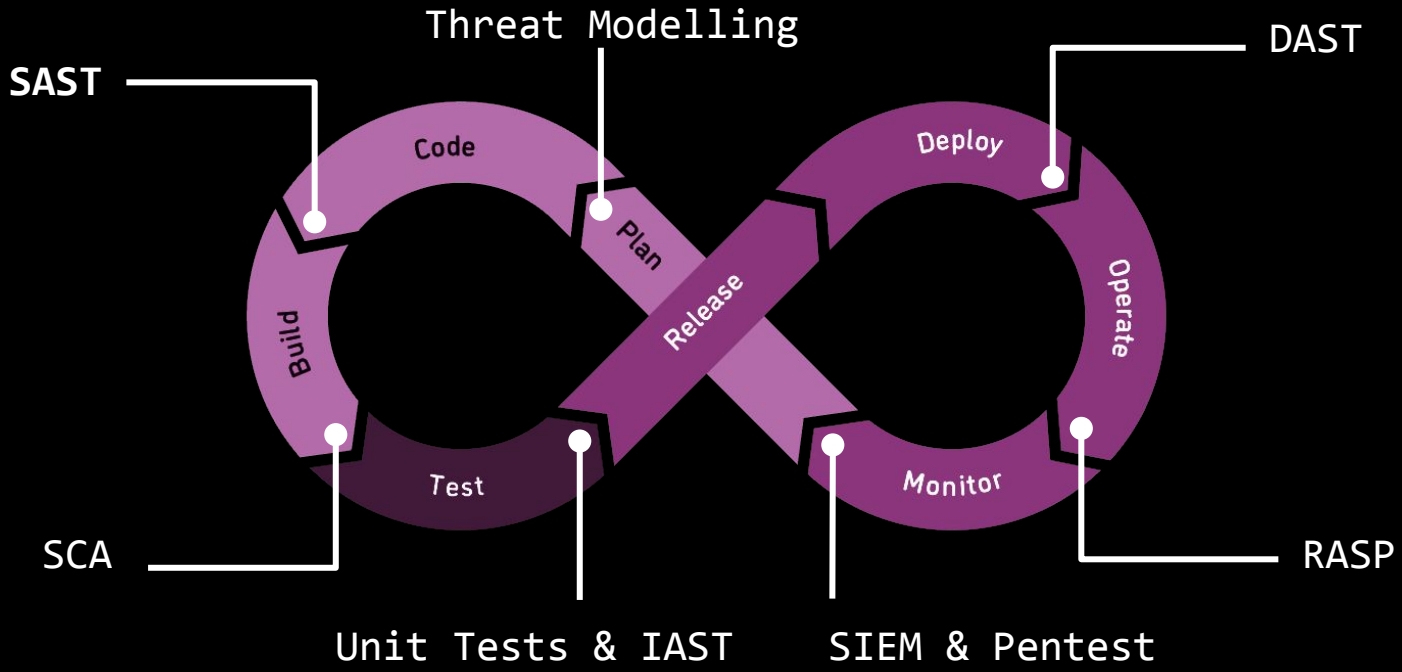
DECEMBER 10-11, 2025
EXCEL LONDON / UNITED KINGDOM

Unsafe Code Detection Benchmark

Stress-Testing SAST & LLMs on Modern Web Backends

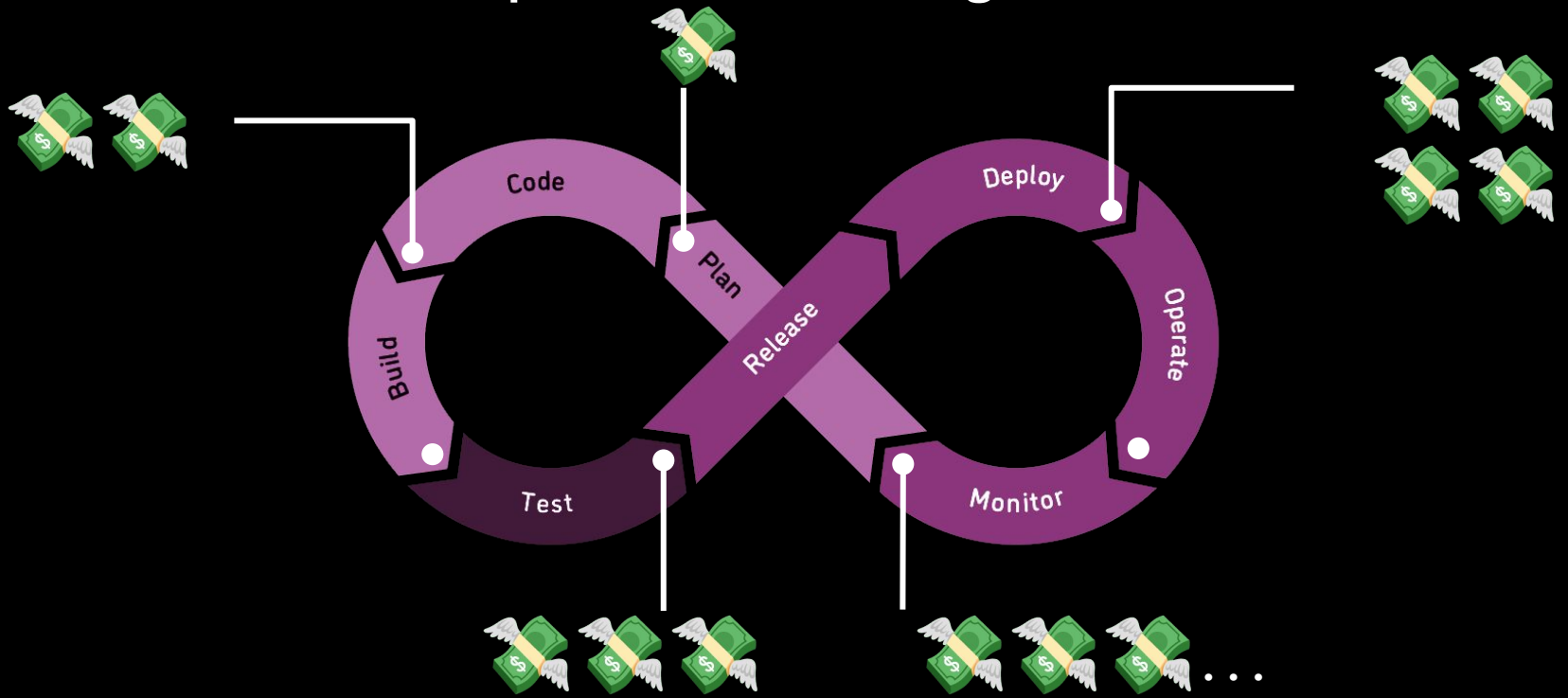
Andrew Konstantinov & Irina Iarlykanova

How Do We Know Our Code is Secure?



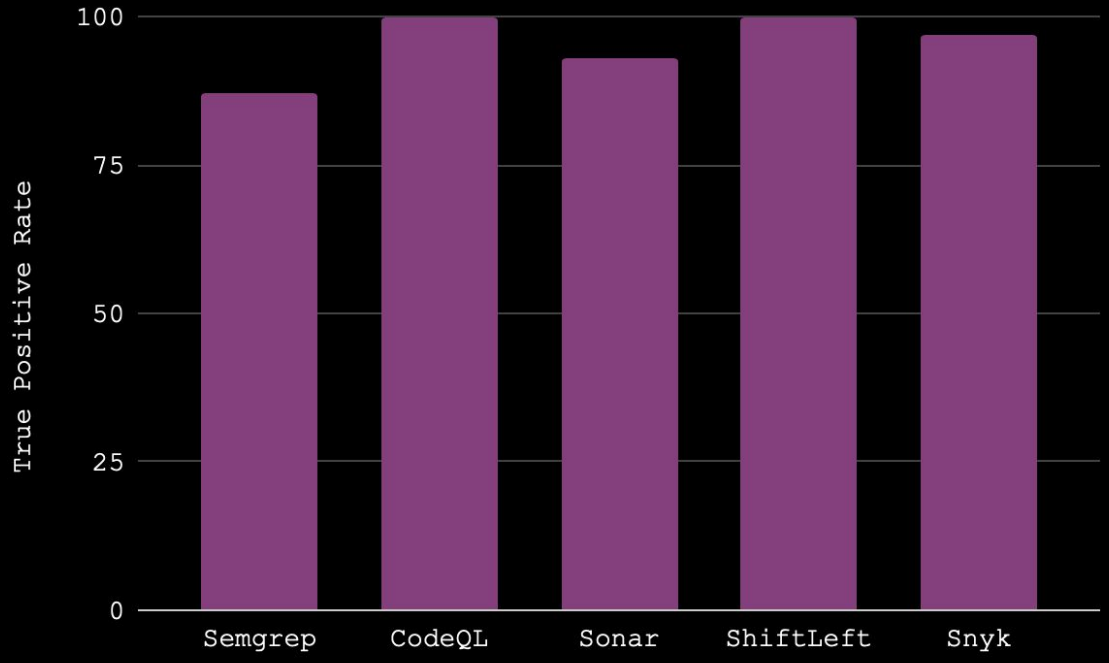
https://www.virtasant.com/blog/sdlc-methodologies

How Much Do We Spend On Securing Our Code?



https://www.virtasant.com/blog/sdlc-methodologies

The Promise of SAST



The Promise of SAST

According to benchmarks, SAST is a solved problem

- **OWASP Benchmark**: 80%-100% detection rates
- **Vendor claims**: "Comprehensive coverage"

Green CI/CD pipeline = secure code. Right?

0% - Detection Rate on Realistic Vulnerabilities

0% – Detection Rate on Realistic Vulnerabilities

Did we do something wrong? 🤔💭

Tools Found Easy Bugs, Missed Logic

CONFUSION VULNERABILITIES

33 examples across 5 categories, Python/Flask

- **Source:** Query vs Body vs JSON merging
- **Authentication:** Middleware vs Handler
- **Authorization:** Permission check on X, action on Y
- **Cardinality:** Validate one, process many
- **Normalization:** "Krústy" ≠ "Krusty"

Meet Authors

Andrew Konstantinov

ISO @Korsit
Source Code Auditor
Bug Bounty Hunter

Irina Iarlykanova

BSc @Maastricht University
SOC Analyst @DIVD
Open to work!

Agenda

1. **Unsafe Code Lab**

2. **Why Modern Code Breaks Detection**
   a. The “Clean Code” Trap
   b. The Middleware Trap
   c. The State Trap

3. **Why SAST Fails**

4. **The AI Factor**
   a. Can LLMs save us?

5. **The Path Forward**

Unsafe Code Lab

📚 Realistic vulnerable code
🌐 Vulns across frameworks
🪲 One concept at a time

Every vulnerability:

- Fits modern best practices
- Passes code review
- Is caused by “fixing” previous vuln
- Comes with exploit scenario

unsafe-code Public

Unpin | Watch 0 | For

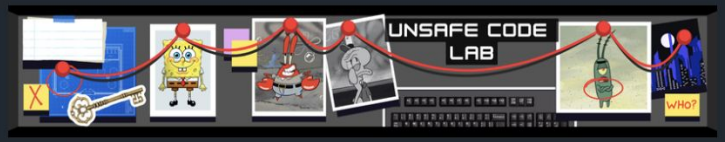
8 Branches | 0 Tags | Go to file t | Add file | Code

lrench1k Update CONTRIBUTING.md with con... ba412c6 · 5 days ago | 186 Commits

.claude	Housekeeping	3 weeks ago
.serena	Add AI automation for consistent docs q...	last month
docs	Fix index.yml regeneration and restore co...	last week
foundations	Replace framework-specific reload with ...	3 weeks ago
tools/docs	Fix index.yml regeneration and restore co...	last week
vulnerabilities	Fix index.yml regeneration and restore co...	last week
.gitignore	Update the typos and regenerate README...	3 months ago
.python-version	Move to uv for python management	3 months ago
CONTRIBUTING.md	Update CONTRIBUTING.md with contrib...	5 days ago

README | Contributing | MIT license | Security

Unsafe Code Lab



Unsafe Code Lab is a hands-on security training ground for code reviewers and penetration testers. Learn to spot vulnerabilities in production-quality code by understanding *why* they happen: refactoring drift, framework design patterns, and subtle API misuse in modern web frameworks like Flask, Django, FastAPI, and Express.js.

Who this is for

> *“The road to hell is paved with refactoring.”*

Best practices can introduce and mask vulnerabilities:

- Separate concerns → Validation/Execution split
- Validate early → Middleware reads different source than handler
- DRY → Shared helpers with implicit assumptions

Our thesis: security tools are not keeping up with modern development patterns.

Real-World Business Impact

Financial Loss
Pay $0.01 for $1,000 order | Refund same item 5×

Account Takeover
Become manager | Hijack accounts | Session fixation

Data Breach
View competitor's orders | Enumerate all coupon codes

Business Disruption
Steal reviews | Make competitor's page show your content

Our Corpus vs OWASP & API Top 10

OWASP Top 10 Web 2021	OWASP Top 10 API 2023
A01 Broken Access Control	API1 Broken Object Level Authorization
A02 Cryptographic Failures	API2 Broken Authentication
A03 Injection	API3 Broken Object Property Level Authz
A04 Insecure Design	API4 Unrestricted Resource Consumption
A05 Security Misconfiguration	API5 Broken Function Level Authorization
A06 Vulnerable and Outdated Components	API6 Unrestricted Access to Sensitive Business Flows
A07 Identification and Authentication	API7 Server Side Request Forgery
A08 Software and Data Integrity	API8 Security Misconfiguration
A09 Security Logging and Monitoring	API9 Improper Inventory Management
A10 Server Side Request Forgery	API10 Unsafe Consumption of APIs
Total: 5	Total: 6

One Root Cause → Many Symptoms

	Input Source Confusion	Authentication Confusion	Authorization Confusion	Cardinality Confusion	Normalization Confusion
	Financial, Takeover	Financial, Takeover, Breach	Financial, Takeover, Breach	Financial, Privesc	Financial, Takeover, Breach
OWASP	A01, A04, API1, API3, API6	A01, A07, API2, API5	A01, API1, API3, API5	A01, API4, API6	A03, A04, A07, A08, API1, API2, API10
CWE	20, 287, 346, 384, 602, 639, 706, 915, 1284	284, 287, 302, 384, 650, 1390	200, 284, 287, 639, 862, 863	20, 285, 384, 639, 682, 799, 863, 1284	20, 89, 135, 176, 178, 180, 185, 200, 436, 706, 863

Case Study 1

The Clean Code Trap

The Exploit

```
POST /cart/42/checkout
Content-Type: application/json

{
  "coupon_codes": [“SAVE50”, “30OFF”, “SAVE50”],
  "delivery_address": "124 Nowhere Road"
}
```

Result: 50% discount applied twice

This Looks Obvious. Why Does It Reach Prod?

The vulnerable flow is split across “clean” refactors.

- Validator: Takes input, removes garbage, returns clean set.
- Executor: Takes validated context, performs action.

Everyone agrees the exploit is trivial *once you see it*, but the code around it is exactly what we encourage in reviews.

The Validation (Good Code?)

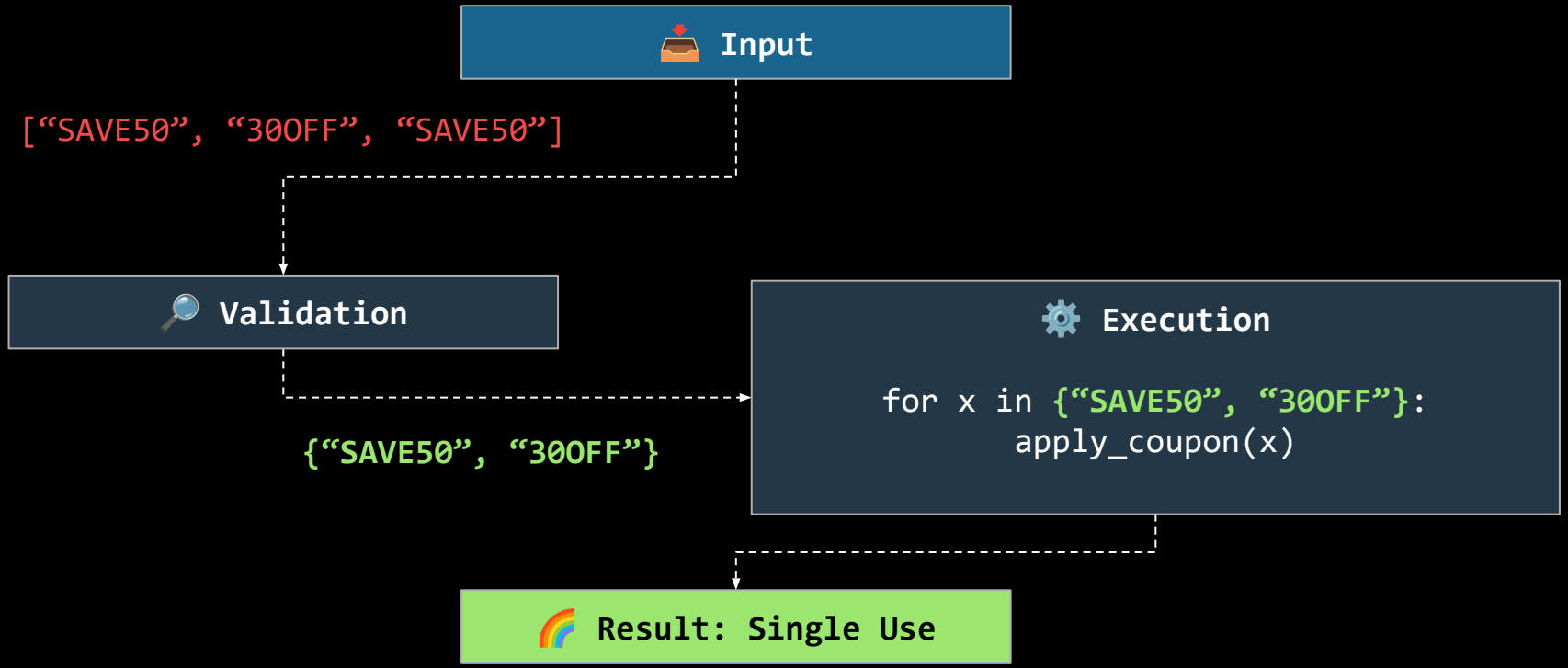
```python
def extract_single_use_coupons(coupon_codes: list[str]) -> set[str]:
    valid_coupons = set()
    for code in coupon_codes:
        if is_valid_and_unused(code):
            valid_coupons.add(code)
    return valid_coupons


# Input:  ["PROMO1", "PROMO2", “PROMO1”]
# Output: {"PROMO1", “PROMO2” }  ← Deduplicated
```

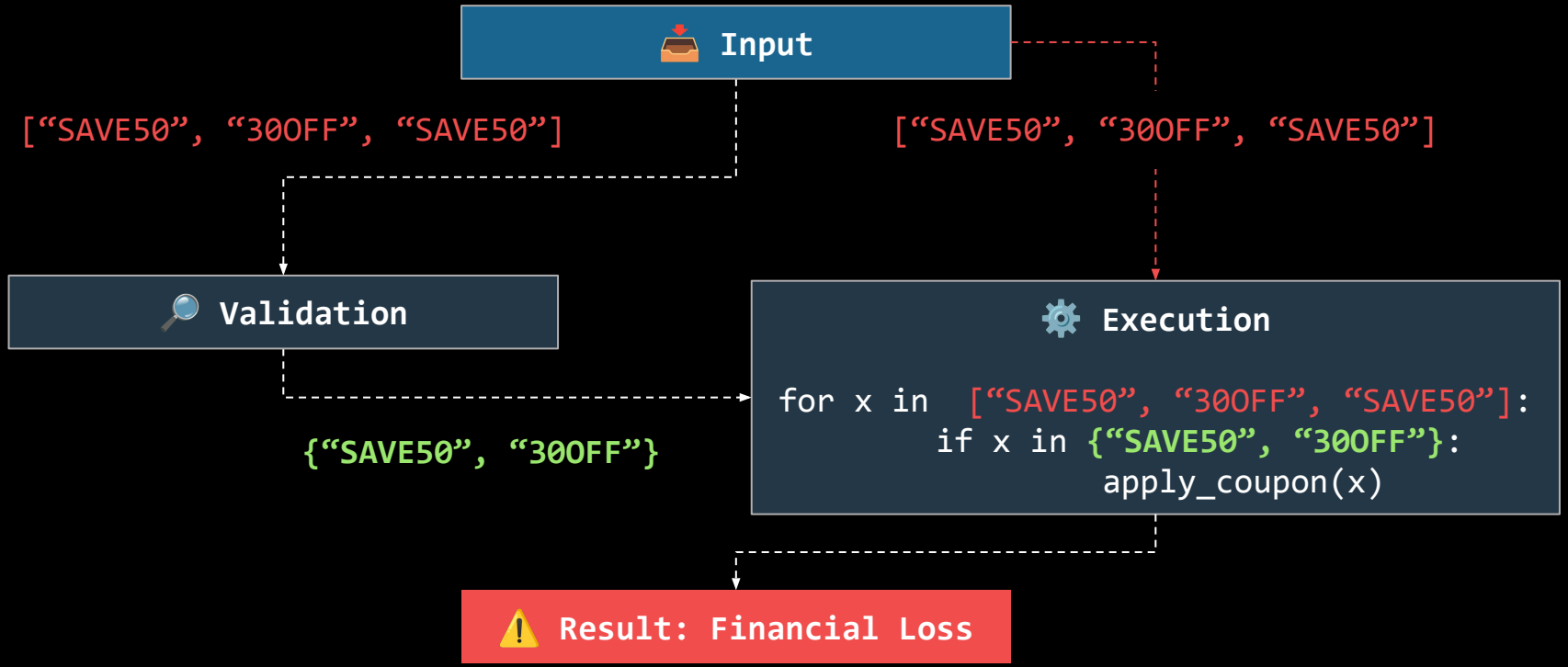
The Execution

```
def apply_coupons(order: Order, coupons: list[str], valid_c: set[str]):
    for code in coupons: # Iterates the ORIGINAL LIST
        if code in valid_c: # Checks against the SET
            apply_discount(order, code)
```

Validation & Execution Flow: Expectation



Validation & Execution Flow: Reality



Imagine if you could ask your SAST

> *"Show me everywhere a transformed collection (Set) differs from its iteration target (List)."*

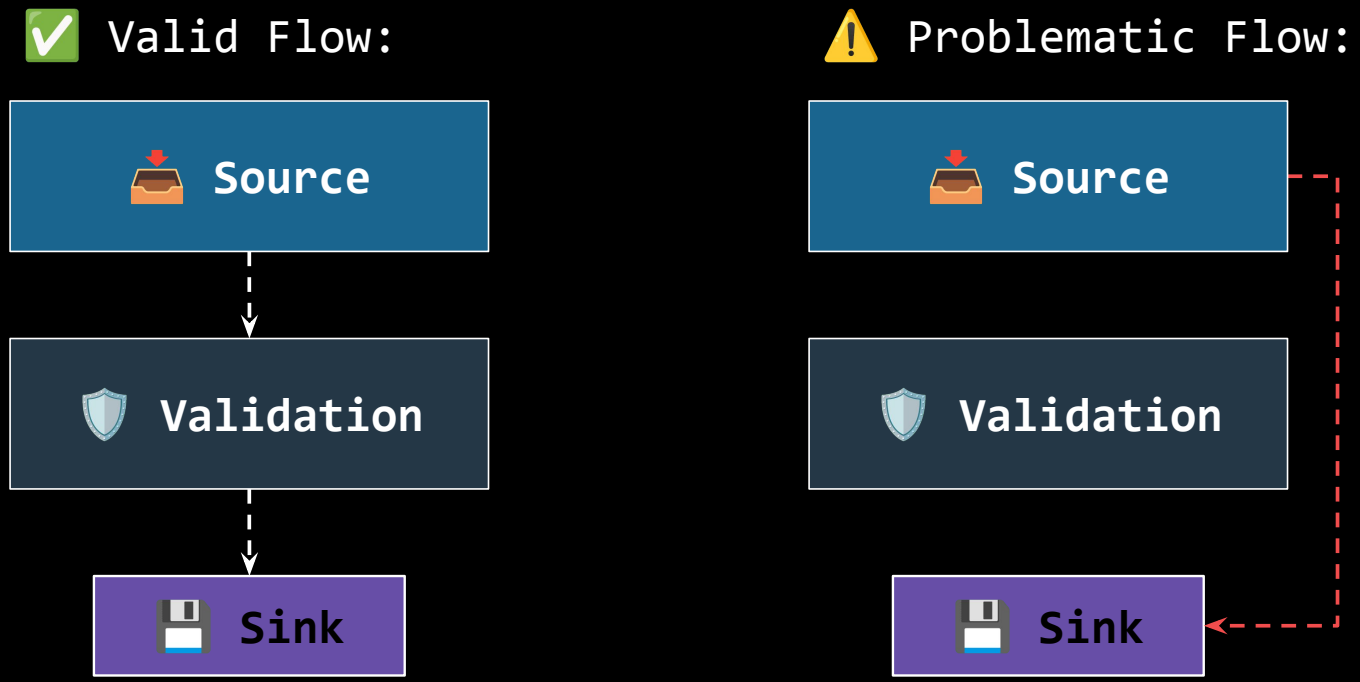
Today's tools don't model collection semantics. They just see "Data Flow".

They don’t see “Is the validated thing the same thing we’re iterating over?”

SAST Capability Levels

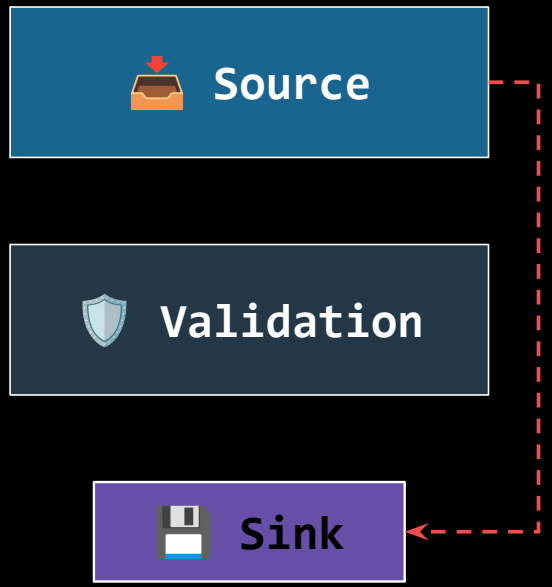
SAST Past	SAST Present	SAST Future
Security Linter	Taint Analysis	???
Find code that will always lead to vulnerabilities: eval()/exec()	Checks user input: 1) Does it reach sink? 2) Is it validated?	

Taint Analysis: What It Expects

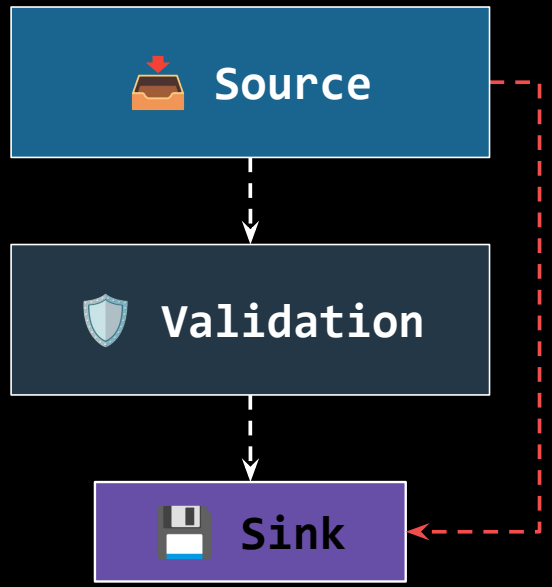


SAST Failure Mode #1

SAST expected:



In our code:



Naive SAST Approach

```
- id: v403-naive-set-list-duplication
  patterns:
    - pattern-inside: |
        valid_single_use = extract_single_use_coupons($CODES)
        ...
    - pattern: |
        for $CODE in $CODES:
          if $CODE in valid_single_use:
            ...
```

Hardcodes function and variable names.
Breaks if code is refactored.
Does not match semantically identical bugs.

Pragmatic SAST Approach

```yaml
- id: v403-pragmatic-enforce-reassignment
  patterns:
    - pattern: $RES = extract_single_use_coupons($INPUT)
    - pattern-not: $INPUT = extract_single_use_coupons($INPUT)
  message: |
    Convention: Result must be assigned back to same variable.
```

Establishes contract: `coupons = extract_single_use_coupons(coupons)`
Works regardless of downstream loop patterns.
Self-documenting the intent in the code structure.

Case Study 2

Validate Early, Fail Fast

The Exploit

```
POST /cart/42/checkout?tip=20
Content-Type: application/json

{
  "tip": -50,
  "delivery_address": "124 Nowhere Road"
}
```

Result:

- Middleware validates tip=20
- Handler applies tip=-50
- Customer gets $50

The Validation Middleware

The team was fixing a Mass Assignment bug. They added middleware to block dangerous params.

```
@before_request
def security_check():
    # Helper to find data anywhere (Query, Body, Form)
    if get_param("order_id"):
        abort(400, "Mass assignment attempt detected")

    # While we're here, validate the tip!
    tip = get_param("tip")
        if tip and int(tip) < 0:
        abort(400, "Negative tip not allowed")
```

Textbook Defense in Depth.

The Helper Function

Flexible. Supports both URL and Body params.

```
def get_param(key):
    """Convenience helper for mobile + web support"""
    # 1. Check Query String (Mobile app sends here)
    if key in request.args:
        return request.args.get(key)

    # 2. Check JSON Body (Web app sends here)
    return request.json.get(key, None)
```

The Handler

Handlers usually prefer JSON body for POST requests.

Tip value is read directly from the body, bypassing helper.

```
def checkout():
    data = request.get_json() or request.args

    tip = data.get("tip")
    charge_customer(amount + tip)
```

Middleware: Expectation

🌈 **Result:** Customer tips $20

Middleware: Reality

Imagine if you could ask your SAST tool

> *"Find all cases where Validation reads from container X but Execution reads from container Y, for the same field."*

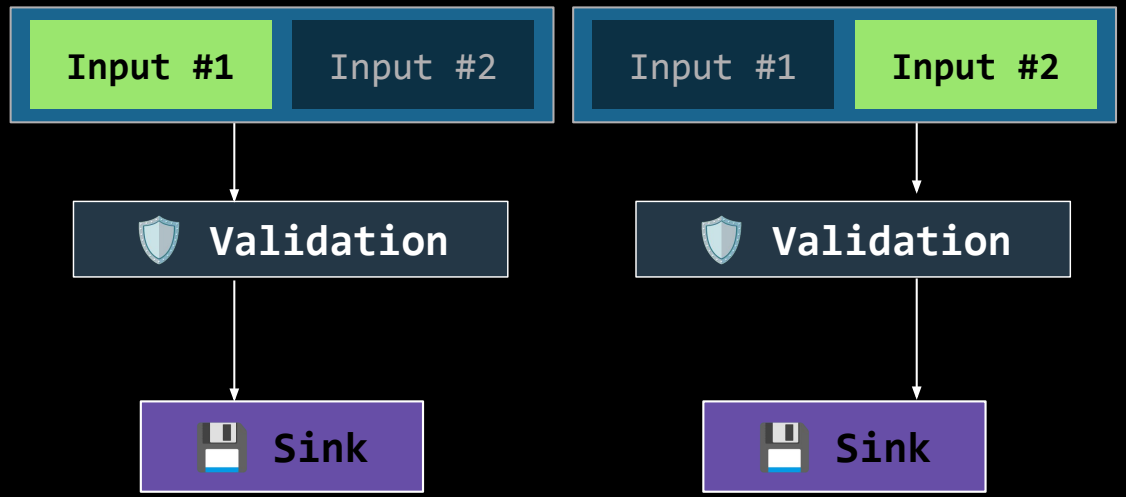
Examples:

- request.args vs request.json
- headers vs body
- form fields vs JSON

Today, most tools flatten this into just “user input”.

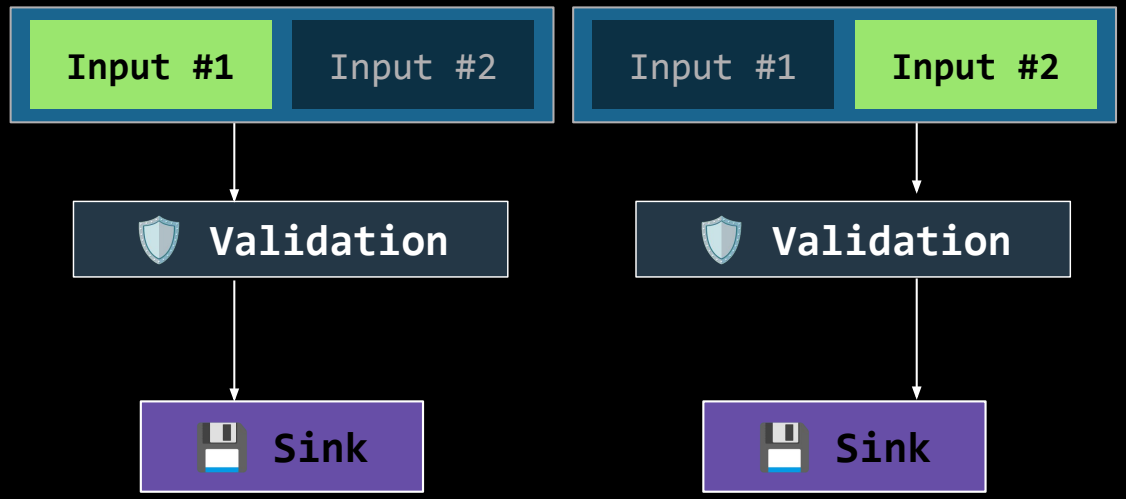
SAST Failure Mode #2

SAST tracks: single input at a time

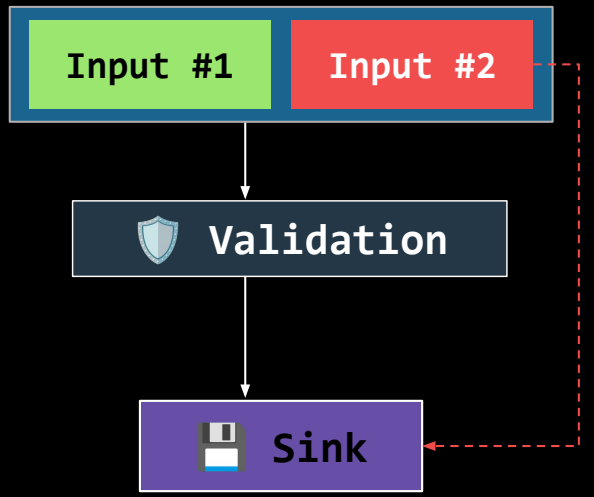


SAST Failure Mode #2

SAST tracks: single input at a time



Exploit: both present



Naive SAST Approach

```
- id: v104-naive-container-mismatch
  pattern: |
    ...
    request.args.get("tip")
    ...
    request.json.get("tip", ...)
```

- Specific to ‘tip’.
- Requires same file and will miss middleware/handler split.
- No understanding of validate-vs-execute semantics.

Flag Raw Access

```
{
  "sources": [
    { "name": "DirectJsonAccess",
      "comment": "Direct access to request.json" }
  ],
  "rules": [{
    "name": "v104: Direct JSON Access",
    "code": 5001,
    "sources": ["DirectJsonAccess"],
    "sinks": ["GeneralSink"],
    "message_format": "Data from request.json accessed directly"
  }]
}
```

- Only flags raw access, not the mismatch.

Pragmatic Approach: Ban Raw Container Access

```yaml
- id: v104-pragmatic-ban-raw-access
  patterns:
    - pattern-either:
        - pattern: request.json
        - pattern: request.args
        - pattern: request.form
    - pattern-not-inside: |
        def get_request_parameter(...):
          ...
```

- ALL parameter reads go through ONE function.
- Single source of truth for input handling
- Helper can validate, log, sanitize consistently

Pragmatic Approach: Annotation-based Convention

```
- id: v104-convention-undeclared-input-source
  patterns:
    - pattern-inside: |
        @$BP.route(...)
        def $HANDLER(...):
            ...
    - pattern-either:
        - pattern: request.args.get(...)
        - pattern: request.json.get(...)
    - pattern-not-inside: |
        @input_source(...)
        def $HANDLER(...):
            ...
```

- ALL parameter reads go through ONE function.
- Single source of truth for input handling
- Helper can validate, log, sanitize consistently

Case Study 3

The “Consume Once” Trap

Defense-in-depth: Database Level Scoping

```
def update_restaurant_menu_item(item_id, ...):
    """
    Safe-by-Design: Always scopes queries to the current tenant.
    Prevents IDORs by forcing a check against the trusted context.
    """
    # 1. Resolve the Tenant ID
    restaurant_id = get_trusted_restaurant_id()

    # 2. Scoped Query
    query = select(MenuItem).where(
        MenuItem.id == item_id,
        MenuItem.restaurant_id == restaurant_id
    )

    # ... executes update ...
```

Consume Once

```
def consume_param(name):
    """
    Safe Accessor: Reads a value and REMOVES it.
    Ensures the same parameter is never read twice.
    """
    return request.json.pop(name, None)

def get_trusted_restaurant_id():
    if g.get("authorized_restaurant"):
        return g.authorized_restaurant.id

    return consume_param("restaurant_id") or g.get("restaurant_manager")
```

The Safe Pattern

```
@bp.patch("/<int:restaurant_id>/menu/<int:item_id>")
@require_restaurant_manager
def update_menu_item(item_id):
    # Decorator already called get_trusted_restaurant_id()
    # Result cached in g.authorized_restaurant

    update_restaurant_menu_item(item_id, ...)  # Uses cached value ✓

# Inside decorator
def require_restaurant_manager(func):
    g.authorized_restaurant = get_trusted_restaurant()
    require_condition(g.authorized_restaurant.id == g.restaurant_manager)
    return func(...)
```

The Copy-Paste

```python
@bp.patch("/menu/<int:item_id>")
def update_menu_item_route(item_id: int):
    """Update a menu item."""
    # Copied from restaurants.py but without the decorator
    # Developer added manual authorization check instead.
    restaurant_id = get_trusted_restaurant_id()
    require_condition(restaurant_id == g.restaurant_manager)

    menu_item = update_restaurant_menu_item(item_id, ...)
```

Feature Update: Batch Support

```
def consume_param(name):
    """
    Safe Accessor: Reads a value and REMOVES it.
    Ensures the same parameter is never read twice with different values.
    """
    value = request.json.get(name)

    # FEATURE: Batch Operation Support
    # If it's a list, we consume items one by one.
    if isinstance(value, list):
        return value.pop(0) if value else None

    del request.json[name]
    return value
```

How many consume_param's?

```python
@bp.patch("/menu/<int:item_id>")
def update_menu_item_route(item_id: int):
    """Update a menu item."""
    # Copied from restaurants.py but without the decorator
    # Developer added manual authorization check instead.
    restaurant_id = get_trusted_restaurant_id()                # 1 consume
    require_condition(restaurant_id == g.restaurant_manager)

    menu_item = update_restaurant_menu_item(item_id, ...)      # 2 consume
```

First call: authorization check

Second call: database scoping

Same parameter, different values after pop()

Exploit Development

(attacker owns restaurant 2, modifies menu for restaurant 1)

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01
}
```

Blocked by DB

Exploit Development

(attacker owns restaurant 2, modifies menu for restaurant 1)

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01
}
```

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01,
    "restaurant_id": 1
}
```

Blocked by DB

Exploit Development

(attacker owns restaurant 2, modifies menu for restaurant 1)

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01
}
```

Blocked by DB

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01,
    "restaurant_id": 1
}
```

Blocked by authz

Exploit Development

(attacker owns restaurant 2, modifies menu for restaurant 1)

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01
}
```

Blocked by DB

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01,
    "restaurant_id": 1
}
```

Blocked by authz

```
PATCH /menu/101 HTTP/1.1
X-API-Key: API-KEY-2
Content-Type: application/json

{
    "price": 0.01,
    "restaurant_id": [2, 1]
}
```

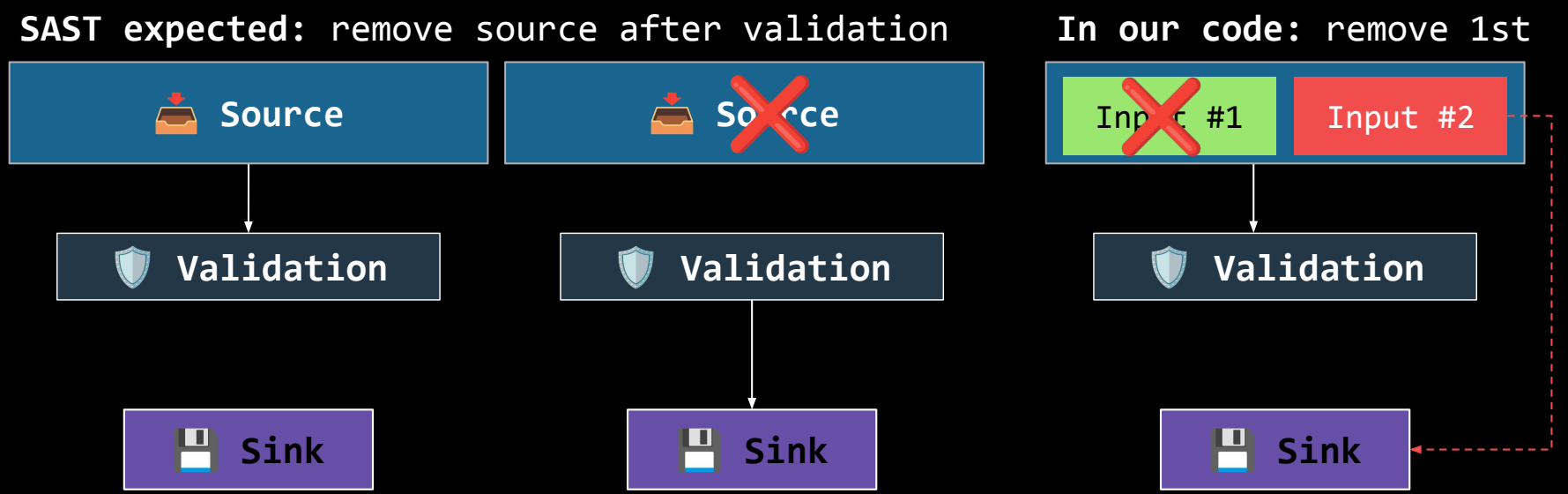
2 in authz, 1 in DB

Imagine if you could ask your SAST tool

> *"Show me where a stateful function (like pop) is called multiple times in a single request lifecycle."*

SAST is **Static**. It struggles with **Temporal State**.

SAST Failure Mode #3



Naive Approach: Detect Double Calls

```
- id: v405-naive-double-consumption
  patterns:
    - pattern-inside: |
        def $FUNC(...):
          ...
    - pattern: |
        $A = get_trusted_restaurant_id(...)
        ...
        $B = get_trusted_restaurant_id(...)
```

Only catches get_trusted_restaurant_id.

Doesn’t know about pop() inside or any other place where pop() is used.

Detect Multiple Calls to Stateful Function

```
from Call c1, Call c2
where
  c1.getFunc().(Name).getId() = "get_trusted_restaurant_id" and
  c2.getFunc().(Name).getId() = "get_trusted_restaurant_id" and
  c1 != c2 and
  c1.getScope() = c2.getScope() and
  c1.getLocation().getStartLine() < c2.getLocation().getStartLine() and
  c1.getArg(0).toString() = c2.getArg(0).toString()
select c2, "Second call to stateful function with same arguments."
```

Tracks across calls in same scope.

Still hardcodes get_trusted_restaurant_id.

Pragmatic: Naming Convention for Side Effects

```yaml
- id: v405-pragmatic-pure-getters
  patterns:
    - pattern-inside: |
        def $FUNC(...):
          ...
    - metavariable-regex:
        metavariable: $FUNC
        regex: ^get_
    - pattern-either:
      - pattern: $X.pop(...)
      - pattern: consume_param(...)
```

Establishes private convention: 'get_' functions must be pure/idempotent.

Avoids misleading names, but only works at a single function depth.

Pragmatic: Naming Convention for Side Effects

```yaml
- id: v405-convention-consume-must-cache
    patterns:
      - pattern: |
          def $FUNC(...):
            ...
      - metavariable-regex:
          metavariable: $FUNC
          regex: ^consume_
      - pattern-not: |
          def $FUNC(...):
            ...
            if hasattr(g, $KEY):
                return getattr(g, $KEY)
            ...
            setattr(g, $KEY, ...)
```

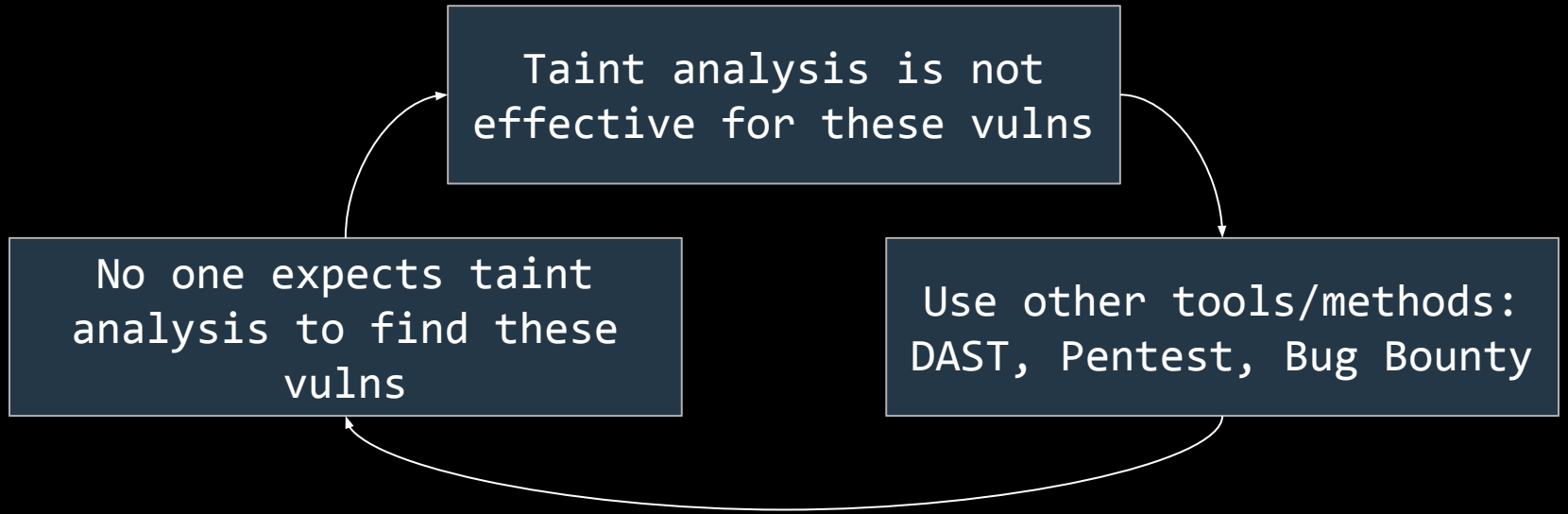
Requires consume_* helpers cache their values, for example on g global obj.

Why Can’t SAST Find Our Bugs?

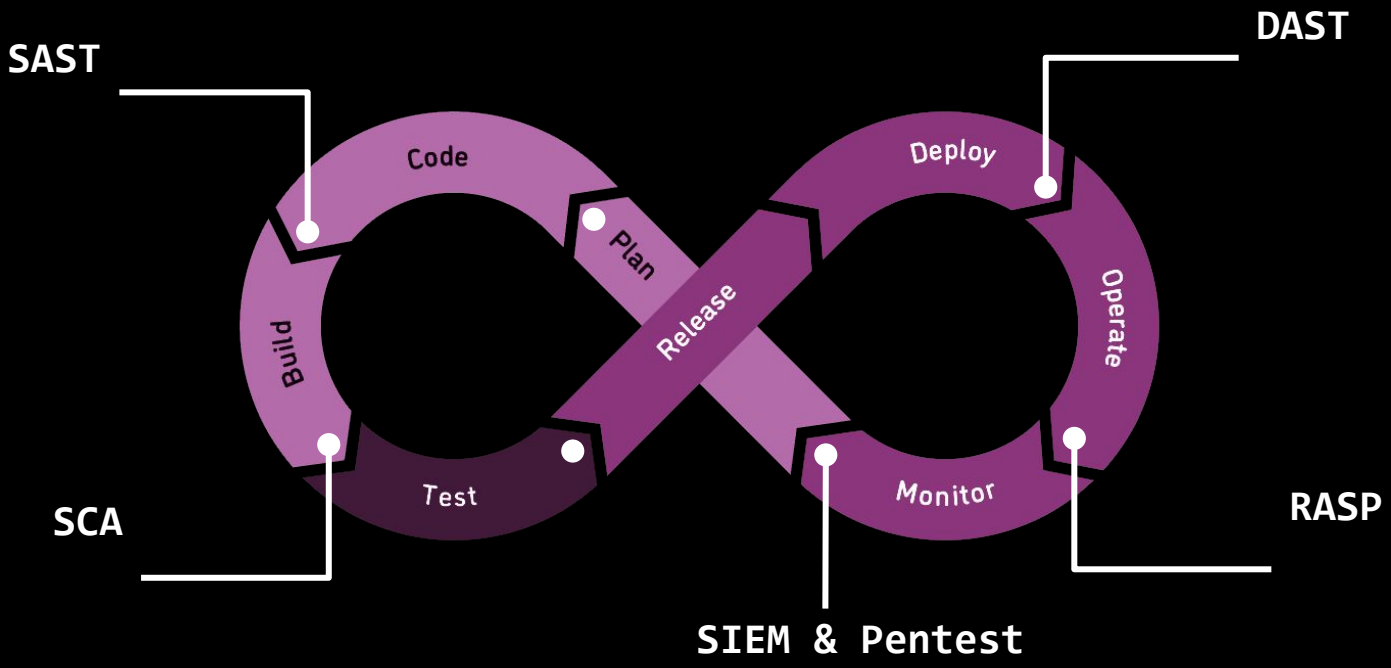
Vicious Circle: Low Expectations

Problem: We have complex vulns which SAST cannot find

Solution: DAST, Pentest, Bug Bounty



Recall SDLC Pipeline

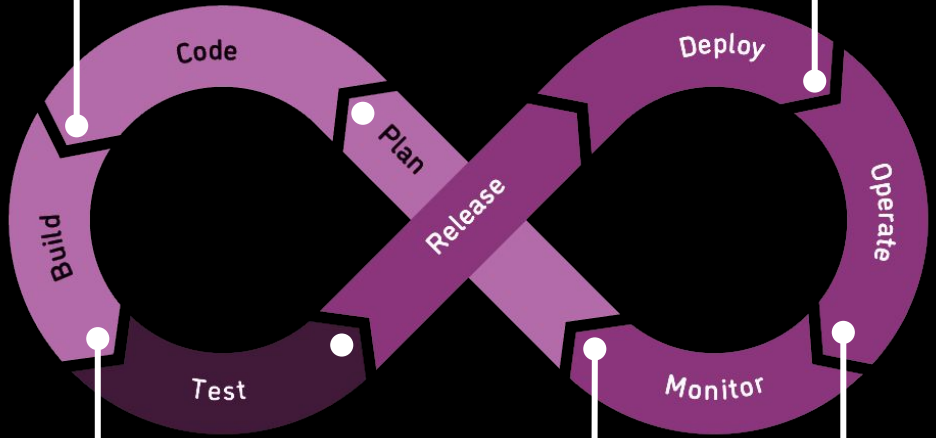


https://www.virtasant.com/blog/sdlc-methodologies

What We Can Find At Each Phase?

SAST
SQLi, XSS

DAST
HPP, CSRF, SSRF



SCA
known CVEs, vuln dependencies

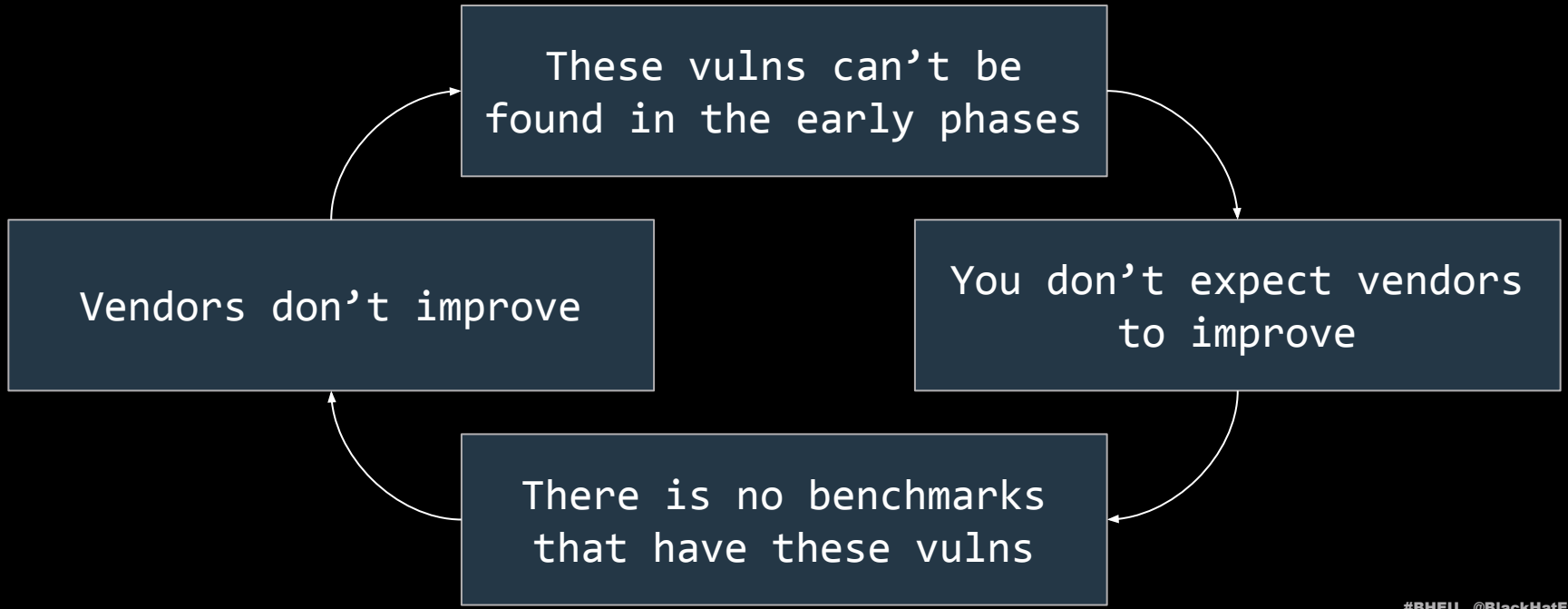
RASP
0-day exploit attempts

SIEM & Pentest
Auth bypass, IDOR

https://www.virtasant.com/blog/sdlc-methodologies

Vicious Circle: Business Impact

You don’t expect SAST to do DAST/pentester job



...or this was the case until November 2022 ⏳

With AI tools, common knowledge changes

Before:

- “These bugs are too hard for static analysis”
- “We’ll catch them with pentests and bug bounties”

Now:

- LLMs can often reason about these flows in natural language
- They can walk the exploit path we just followed manually

These vulns *can* be found in early phases.

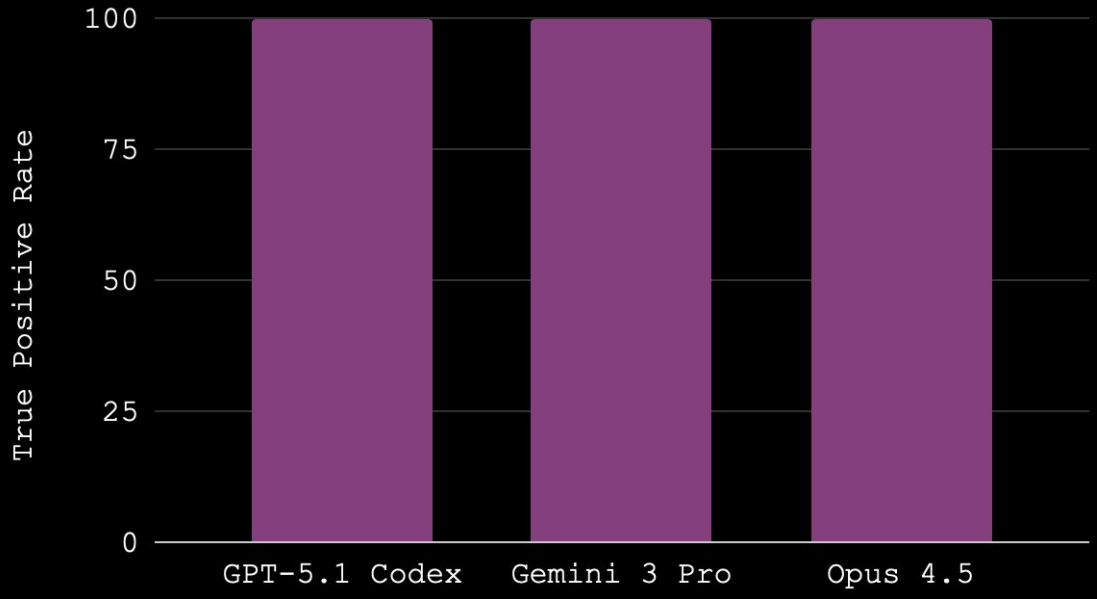
The question is: how do we make that reliable and scalable?

What AI Gets Right

- Understands business logic context
- Can reason about multi-file flows
- Finds authorization confusion patterns

But at what cost?

How AI Performs on Our Benchmark?



Should We Just Switch to AI?

Real-World Business Impact

Cost
100K+ lines repos -> Naive “scan with AI” scales poorly

Reliability
Prompt sensitivity, non-deterministic results

Coverage
Will it find other vulnerability types?

How Does SAST Fit Into the Picture?

- We cannot replace SAST with AI alone
- Today, SAST is underused and under-ambitious
- But SAST has one superpower AI doesn’t:
  - Precise, fast, repeatable – at scale

The Practical Path Forward

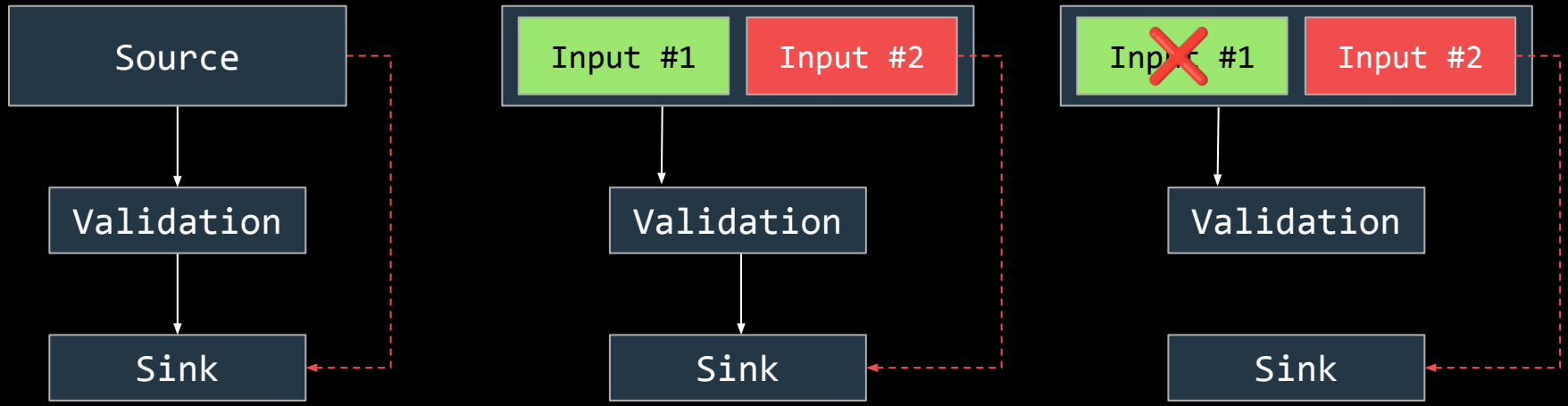
Our Thesis

- We built this corpus as basic security training
  - It does not try to fool SAST, AI, or humans
  - It uses normal “clean code” patterns

- Existing SAST engines:
  - Fail to find these bugs with built-in rules
  - Make it unreasonably hard to write custom rules

Achievable Gap

- Yet we can describe every failure using the same vocabulary SAST already uses for taint analysis:
  - Inputs, validations, containers, sinks, flows
- We don't need entirely new math to do better



> “Just because it is SAST, it does not have to be static”

SAST Past	SAST Present	SAST Future
Security Linter	Taint Analysis	**Security LSP**

- A long-running service, not a one-off CI job
- Exposes:
  - Call graphs, data-flow, and control-flow
- Powers:
  - AI agents that understand your codebase
  - Dynamic investigations and behavior modeling

Takeaways

1. Write custom SAST rules for first party code

   Generic rules find generic bugs

2. Use SAST as a security model, not a gate

   Think “security LSP” not “static checkbox”

3. If you can't write rules for it, simplify your code

   Complexity that confuses SAST also confuses humans & AI

Contact Us

Andrew Konstantinov

✉️ andrew@konst.lv

https://github.com/execveat

Irina Iarlykanova

✉️ irina.iarlykanova@gmail.com

https://github.com/Irench1k

👩‍💻 Open to work!

Unsafe Code Lab