Detecting Data Leaks in Multi-User LLM Apps via Automated User-Scoped Taint Analysis

Sanjib Kumar Sen
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA
ssen@islander.tamucc.edu

Bozhen Liu
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA
bozhen.liu@tamucc.edu

Abstract

LLM-Powered Apps (LPAs) are increasingly being deployed on shared cloud servers where multiple users interact at the same time. This creates data leak risks where one user's private data can flow into another user's session through shared caches, misconfigured access control policies, or concurrency bugs in multi-threaded and event-driven code. Existing taint analysis tools do not handle these risks well because they need manual source-sink marking, only track control and dataflow, and do not account for access control policies, complex app structures, or runtime environments.

In this paper, we propose Access Flow Guard (AFG), a framework that automatically identifies taint sources based on user identity and permission level to detect cross-user data leaks in LPAs. Instead of manually marking sources and sinks as existing taint approaches require, AFG introduces a Multi-User, Multi-Permission (MUMP) setup. MUMP automatically identifies user input entry points by matching function signatures against patterns derived from our dataset of popular open-source LPAs. Each identified input is tagged with the user's identity and permission level. AFG then applies Scoped Taint Pointer Analysis (STPA), building on existing pointer-based taint analysis techniques, to propagate user-tagged objects through a Pointer Assignment Graph. STPA detects where different users' data scopes overlap. We discuss key research challenges, including treating LLM components as black boxes and refining static results through dynamic testing. This approach aims to serve as a basis for systematic security testing of LLM-integrated software in shared multi-user environments.

CCS Concepts: • **Theory of computation → Program analysis**; • **Security and privacy → Software security engineering**.

Keywords: Static Taint Analysis, Pointer Analysis, Multi-user Multi-Permission, LLM-Powered Applications

SOAP '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2709-2/2026/06
https://doi.org/10.1145/3814987.3814995

ACM Reference Format:
Sanjib Kumar Sen and Bozhen Liu. 2026. Detecting Data Leaks in Multi-User LLM Apps via Automated User-Scoped Taint Analysis. In *Proceedings of the 15th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3814987.3814995

1 Introduction

Taint analysis, which tracks the flow of user input through a system to find where it may reach unsafe destinations, has been a standard technique for decades [3, 17]. It is widely used in tools for Android apps [4, 8, 22, 23], JavaScript applications [5, 18], and cloud security platforms [19], where all modules within an app follow a uniform implementation style and structure. However, LLM-Powered Apps (LPAs) are rapidly growing in popularity, using APIs from services such as Google Gemini, OpenAI GPT, and Meta LLaMA. These apps introduce new security challenges not covered by existing taint analysis tools. When LPAs built for single-user devices are deployed on shared cloud servers, multiple users share resources such as retrieval-augmented generation (RAG), APIs, and databases, creating risks of data leaks across user sessions.

Existing static taint analyses either use symbolic execution to filter input [16, 21] or apply propagation rules on dataflow graphs [1, 4]. The former often requires SMT solvers that struggle with complex constraints, and the latter is expensive with worst-case time complexity of $O(E \times D^3)$ [20]. Both approaches require manual marking of sources and sinks, which does not scale to real-world applications with multiple users and different permission levels. LPAs are a representative and increasingly common example of such applications. On the other hand, dynamic taint analysis imposes high runtime overhead, often causing 30–50× slowdown [6]. Even state-of-the-art tools such as `libdft` still introduce up to 6X slowdown [12], largely because they require instrumenting source or binary code for each application.

In this paper, we propose Access Flow Guard (AFG), a framework for detecting cross-user data leaks in multi-user LPA deployments. AFG's core contribution is the Multi-User, Multi-Permission (MUMP) setup, which automatically identifies taint sources from a dataset of user input function patterns derived from popular open-source LPAs [9, 10]. MUMP

tags each identified input with the user's identity and permission level, eliminating manual annotation. AFG then applies Scoped Taint Pointer Analysis (STPA), which builds on existing pointer-based taint analysis techniques [11, 14]. STPA propagates user-tagged objects through a Pointer Assignment Graph (PAG) and detects data scope overlaps between users. In summary, our contributions are as follows:

- We propose MUMP, an automated taint source identification approach that matches function signatures against a pre-built dataset recording file paths, parameter types, and return types of user input functions from popular open-source LPAs.
- We apply MUMP with STPA, which builds on existing pointer-based taint analysis [11, 14] to propagate user-tagged objects through a PAG and detect cross-user data scope overlaps. We demonstrate the approach on a real-world LPA example.
- We discuss open research challenges, including treating LLM components as black boxes during taint propagation, handling concurrency in multi-threaded and event-driven architectures, and refining static results through dynamic analysis.
- The prototype of STPA is publicly available [2].

2 Background and Motivation

Existing Taint Analyses. Existing taint analyses focus on tracking control and data flow [17, 21]. However, real-world security vulnerabilities in software often arise from a combination of several factors, including misuses of sink functions, misconfigurations in access control policies, complex app architecture, and app's runtime environment [9]. Existing taint analyses overlook these factors, leaving a significant gap when applied to LPAs that operate in multi-user shared environments.

Our study of popular open-source LPAs on GitHub [9, 10] shows that they primarily use TypeScript, JavaScript, Python, Go, and Rust. Most of these projects currently employ an event-driven architecture without threading, presenting minimal security risks in their current form. However, as these projects grow to include multi-threading and event-driven designs, or evolve into more complex architectures (e.g. microservices), they face increased safety and security challenges. This risk becomes significant in apps that use LLM APIs, since not all APIs are designed to be thread-safe, which can cause data corruption, inconsistent data states, or system crashes that could be exploited. Moreover, most LPAs are built for single-user use on personal devices such as smartphones and personal computers, with a relatively simple runtime environment. Installing and deploying these apps in cloud environments such as Google Cloud or AWS, where resources are shared among multiple users, introduces security challenges that are difficult to test and debug due to the non-determinism of multi-threaded and event-driven systems.

Table 1. Comparison of taint analysis approaches for multi-user LPA environments. ✓ = supported, ✗ = not supported, **P** = partially supported.

Feature	FlowDroid	Clang SA	libdft	P/Taint	AFG
Auto source/sink	✗	✗	✗	✗	✓
MUMP-aware	✗	✗	✗	✗	✓
Access control	✗	✗	✗	✗	✓
Pointer-based	✗	✗	✗	✓	✓
LLM black-box	✗	✗	✗	✗	**P**
Low overhead	✓	✓	✗	✓	✓

Table 1 compares existing taint analysis tools across six features relevant to multi-user applications. FlowDroid [4], Clang Static Analyzer [1], and libdft [12] all require an analyst to manually specify sources and sinks before the analysis can begin. None of them can distinguish between different users' data or model the access control policies that govern which users may access which resources. P/Taint [11] advances beyond these tools by using pointer analysis to propagate taint instead of relying solely on dataflow graphs. However, it still requires manual source and sink configuration and does not support multi-user or access control reasoning. On the dynamic side, libdft instruments source or binary code at runtime, imposing overhead that limits practical adoption. No existing tool can analyze LLM components whose internals are unavailable for inspection.

Motivating Example. To demonstrate these risks, consider a real-world example from our study. Two users, A and B, each with admin access, install the same LPA on a Linux cloud server. This app, designed for a single user on a local machine, uses LLM APIs and interacts with Google Calendar to manage events. It uses Redis for caching to improve performance. Because it is a single-user app, it starts Redis as a system service via `systemd` on Linux, caching data without user distinction. The app automatically saves snapshots to the `dump.rdb` file located at the default directory `/var/lib/redis/`. Users A and B are unaware of this setup and rely on the app for personal use. When User A queries the app, the LLM response containing User A's private calendar events is cached in Redis. Later, when User B asks the same question, the app returns User A's cached response instead of generating a fresh one for User B. This is a direct data leak of User A's private information to User B, caused by the combination of a single-user caching design, the lack of user-level access control in the cache, and the shared runtime environment on the cloud server. Figure 1 illustrates how AFG processes this scenario. In Stage 1, MUMP tags User A's and User B's inputs with their respective identities and produces user-tagged taint sources. In Stage 2, STPA builds a PAG from the application code, propagates the tagged objects through constraints, and detects that both users' data

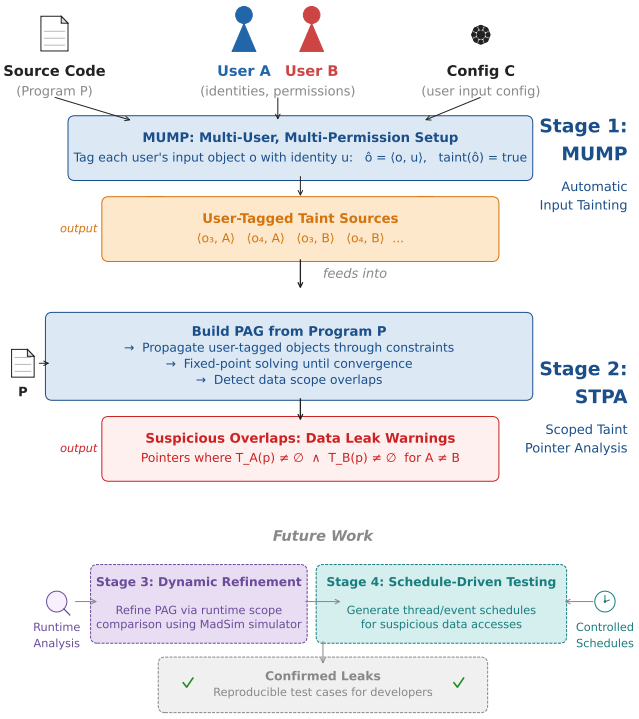

Figure 1. AFG framework pipeline applied to the motivating example. Stage 1 (MUMP) tags user inputs. Stage 2 (STPA) detects data scope overlaps. Stages 3 and 4 are future work.

scopes overlap at the Redis cache pointer. The output is a data leak warning identifying the shared cache as suspicious. Stages 3 and 4, shown as future work, would refine this result through runtime analysis and schedule-driven testing to confirm whether the leak is triggered under a concrete execution order.

This example shows how leaks in LPAs do not come from a single source flowing to a single sink in the traditional taint analysis sense. Instead, they arise from the interaction between the app's architecture, its access control assumptions, and the runtime environment. Existing taint analysis tools cannot detect this type of leak because they do not model per-user data scopes or access control policies. In a multi-user setting where each user has different permission levels and authentication methods such as tokens, OAuth, or multi-factor authentication, a new approach is needed that can automatically identify taint sources based on user identity, account for these policies during analysis, and reason about concurrent execution where non-deterministic thread and event scheduling can expose data from one user's session to another.

How AFG Addresses These Gaps. AFG addresses all six limitations in Table 1 through MUMP and STPA, described in detail in Section 3. For LLM black-box components, AFG treats all outputs of an LLM API call as tainted by its inputs. We plan to refine this strategy in future work.

Algorithm 1: AFG: MUMP Setup and STPA

Input: $P \leftarrow$ program, $U \leftarrow$ user set, $C \leftarrow$ user input config
Output: $S \leftarrow$ set of suspicious overlaps
`// Stage 1: MUMP Setup`
for *each user* $u \in U$ **do**
  for *each input object* o *associated with* u *in* C **do**
    $\hat{o} \leftarrow \langle o, u \rangle$ `// tag with user identity`
    $taint(\hat{o}) \leftarrow true$
  end
end
`// Stage 2: Scoped Taint Pointer Analysis`
Build PAG from program P
Initialize $pt(p)$ with origin-tagged objects for all pointers p
repeat
  for *each constraint* c *in PAG* **do**
    Update pt sets according to constraint type (Table 2)
    Propagate taint: if source node is tainted, mark destination as tainted
  end
until *fixed point (no changes in* pt *or taint sets)*
$S \leftarrow$ DetectOverlap(U, pt)
return S

DetectOverlap(U, pt):
Input: $U \leftarrow$ user set, $pt \leftarrow$ points-to sets
Output: $S \leftarrow$ set of suspicious pointers
$S \leftarrow \emptyset$
for *each pointer* p *in PAG* **do**
  for *each pair of distinct users* $A, B \in U$ **do**
    $T_A(p) \leftarrow \{\hat{o} \in pt(p) \mid \hat{o} = \langle o, A \rangle\}$
    $T_B(p) \leftarrow \{\hat{o} \in pt(p) \mid \hat{o} = \langle o, B \rangle\}$
    if $T_A(p) \neq \emptyset \wedge T_B(p) \neq \emptyset$ **then**
      $S \leftarrow S \cup \{p\}$
    end
  end
end
return S

3 Access Flow Guard (AFG)

AFG consists of four stages in total. This paper presents the first two: Automatic Input Tainting and STPA. We discuss the remaining stages, dynamic refinement and schedule-driven testing, in Section 6.

In the first stage, AFG introduces a Multi-User, Multi-Permission (MUMP) setup that eliminates the need for manual source and sink marking. MUMP automatically identifies user input functions by matching their signatures against a pre-built dataset. The dataset is predefined from our study of open-source Rust LPAs [9, 10], analogous to the predefined

source and sink lists that ship with FlowDroid [4]. It records every function that receives user input, along with its file path, parameter types, and return types. Given a new LPA, MUMP matches these signatures against the target's source code, identifies Rust's Mid-level Intermediate Representation (MIR) callsites of input functions, and attaches user origin tags, avoiding manual per-target annotation. Each user in the MUMP configuration is defined by an identity $u \in U$ and a permission level. The permission level specifies what resources the user is allowed to access, such as read-only, read-write, or admin. Every object allocated through a matched input function is tagged with the corresponding user identity as its origin. Two users with different permission levels accessing the same resource produce two separately tagged object sets. This separation enables STPA to detect overlaps between users' data scopes. This idea of analyzing multiple execution contexts is inspired by Multi-Context Analysis from Context-Oriented Programming [7].

The second stage uses STPA, which builds on two existing techniques: the origin concept for unifying threads and event handlers [14], and the P/Taint insight that unifies points-to and information-flow analysis [11]. Origin unifies threads and event handlers through two parts: an entry point that marks the start of a thread or event handler, and a set of attributes that capture pointers to memory objects used in that context. STPA extends this by tagging abstract objects that represent user input with a unique identifier as their origin. Formally, each abstract object in the analysis is represented as a pair $\hat{o} = \langle o, u \rangle$, where o is the allocation site and $u \in U$ is the user identity from the MUMP setup. For each pointer variable p, STPA computes a points-to set $pt(p) \subseteq \hat{O}$, where $\hat{O}$ is the set of all origin-tagged objects. It then propagates these tagged objects through a PAG using constraint-based fixed-point solving.

The PAG encodes program statements as constraints over pointer variables. Table 2 lists the five constraint types that STPA derives from program instructions, along with their points-to update rules and taint propagation effects. These constraints are solved iteratively until a fixed point is reached. During solving, taint propagates alongside points-to information: if a source node is tainted, the destination node in the same constraint becomes tainted. As noted in Section 1, building a PAG avoids the complexity of a cross-function dataflow graph, which has worst-case time complexity of $O(E \times D^3)$ [20]. Because STPA builds on this unification of points-to and information-flow analysis, both points-to and taint information can be computed together in a single pass, differing only in their initial settings.

After STPA reaches a fixed point, AFG detects data scope overlaps between users. Let $T_u = \{\hat{o} \in pt(p) \mid \hat{o} = \langle o, u \rangle\}$ denote the set of origin-tagged objects belonging to user u that are reachable from pointer p. A suspicious overlap is flagged when there exists a pointer p such that $T_A(p) \neq \emptyset \wedge T_B(p) \neq \emptyset$

Table 2. PAG constraint types with their points-to update and taint propagation rules. Here o denotes an allocation site and $\hat{o} = \langle o, u \rangle$ denotes an origin-tagged object, where u is the user identity from the MUMP setup.

Type	Statement	Points-to	Taint
AddrOf	$p = \&o$	$\hat{o} \in pt(p)$	$source(\hat{o}) \Rightarrow taint(p)$
Assign	$p = q$	$pt(q) \subseteq pt(p)$	$taint(q) \Rightarrow taint(p)$
Store	$*p = q$	$\forall \hat{o} \in pt(p) :$ $pt(q) \subseteq pt(\hat{o})$	$taint(q) \Rightarrow$ $taint(\hat{o})$
Load	$p = *q$	$\forall \hat{o} \in pt(q) :$ $pt(\hat{o}) \subseteq pt(p)$	$taint(\hat{o}) \Rightarrow$ $taint(p)$
Offset	$p = \&q\text{->}f$	field-sensitive	propagate

for distinct users A and B, meaning that both users' data is reachable from the same reference. The analysis also handles inter-thread communication by modeling channel operations such as send and receive in Rust's message-passing primitives, ensuring that data flowing between threads through channels is properly tracked.

To illustrate how AFG works, consider the Rust code in Figure 2, which reproduces the scenario from our motivating example. Two threads simulate queries from User A and User B, each with its own Redis client (`client1` on port 6379 and `client2` on port 6380). Both threads call `spawn_user_query`, which obtains a Redis connection via `get_connection()`. In the MUMP setup, AFG automatically tags User A's input objects (the query string and username) with origin A, and User B's input objects with origin B. STPA then constructs the PAG for this program. Each user's input variables (such as `question` and `user`) are represented as abstract heap objects distinguished by their origins: $\langle o3, A \rangle$ for User A and $\langle o3, B \rangle$ for User B. The Redis client objects (`client1` and `client2`), labeled `o1` and `o2` in the code, are also tagged with their respective user origins because they are allocated in user-specific contexts. At this point, the data scope accessed by each user appears separate, with no overlap.

However, when both threads call `get_connection()`, the returned connection objects (`o5`) point to the same underlying Redis server. The `cached_answer` retrieved by a `con.get` call and the `answer` stored by a `con.set` call share the same Redis key space. STPA detects that the points-to sets of these objects overlap across the two user contexts, meaning User B's `con.get` operation can return data that was stored by User A's `con.set` operation. AFG flags this overlap as suspicious and reports it as a potential data leak where User A's private calendar information could be exposed to User B through the shared cache.

4 Expected Benefits

Because MUMP derives taint sources from user identity rather than per-target based source-sink configuration, AFG generalizes across LPAs without requiring a separate source and sink configuration for each target application. This makes

```rust
use redis::{Commands, Client};
use std::sync::Arc;
use std::thread;

fn main() {
    let client1 = Arc::new(
        Client::open("redis://127.0.0.1:6379/")
            .unwrap()); // o1
    let client2 = Arc::new(
        Client::open("redis://127.0.0.1:6380/")
            .unwrap()); // o2
    let h1 = spawn_user_query(
        client1, "What is my first event today?",
        "UserA");
    let h2 = spawn_user_query(
        client2, "What is my first event today?",
        "UserB");
    h1.join().unwrap();
    h2.join().unwrap();
}

fn spawn_user_query(
    client: Arc<Client>, question: &str,
    user: &str
) -> thread::JoinHandle<()> {
    let question = question.to_string(); // o3
    let user = user.to_string(); // o4
    thread::spawn(move || {
        let mut con =
            client.get_connection().unwrap(); // o5
        let key = format!("query:{}", question);
        let cached_answer: Option<String> =
            con.get(&key).unwrap_or(None);
        if let Some(answer) = cached_answer {
            println!("{} got: {}", user, answer);
        } else {
            let answer =
                call_llm_api(&question); // o6
            println!("{} got: {}", user, answer);
            let _: () =
                con.set(&key, answer).unwrap();
        }
    })
}

fn call_llm_api(question: &str) -> String {
    format!("Your first event today is ...")
}
```

Figure 2. A simplified LPA where two users share a Redis cache. User A's LLM response is cached via `con.set`. User B may retrieve it via `con.get`, causing a data leak.

the analysis practical for the growing ecosystem of open-source LPAs, where hundreds of projects use different frameworks, APIs, and caching strategies. Adding a new user to the analysis requires only a new entry in the user configuration, not a new set of taint rules.

The unified computation of points-to and taint information over the PAG also makes the analysis suitable for integration into development workflows where fast feedback is important. Incremental code changes require re-solving only the affected constraints in the PAG rather than rerunning the entire analysis.

We first tried LLVM IR, but Rust-specific handling such as demangling, struct-type recovery through bitcasts, and trait-object resolution required significant additional workload, so we pivoted to Rust MIR. MIR keeps Rust-level type names, function paths, and trait-dispatch information, so `afg` can match SDK call signatures and follow dynamic dispatch directly. Although the analyzer is Rust-specific, the MUMP dataset and STPA algorithm can generalize to other languages, with signature catalogues for JavaScript, TypeScript, and additional non-Rust languages included as scaffolding for future per-language analyzers.

AFG does not replace existing security measures such as authentication, encryption, or access control lists. Instead, it complements them by identifying architectural flaws and data flow paths that those measures do not cover. As shown in the running example, even when an LPA correctly authenticates its users, a shared cache without user-level namespacing still leaks data across sessions. AFG detects these structural vulnerabilities that arise from the combination of single-user application design, shared multi-user deployment, and the absence of per-user data isolation in the caching layer.

Limitations. AFG has known blind spots. Signature matching is exact on demangled Rust paths, so generics or trait-object indirection can mask matches when the catalogue is not extended accordingly. The MIR-level analyzer does not model `unsafe` blocks or FFI boundaries, where taint can escape through raw pointers. The post-pass is context-insensitive because the points-to dump it consumes carries no context annotations.

5 Preliminary Evaluation

We implemented AFG's Scoped Taint Pointer Analysis (STPA) as `afg`, a Rust post-pass over a MIR-level pointer analysis. The prototype was implemented based on RUPTA [13]. `afg` reads RUPTA's points-to dump together with a user configuration listing each user's MIR origins (hand-written for our example; automated population from the MUMP dataset is future work), seeds origin tags on the listed locals, propagates tags through the points-to relation to a fixed point, and reports abstract objects reached by two or more users.

Setup. We ran `afg` on a simplified variant of Figure 2 that replaces the Redis client with an in-memory `Arc<Mutex<HashMap<String, String>>>` carrying the same shared-cache semantics. Each user's configuration lists three MIR locals in `demo::main`: the cloned `Arc`, the question string,

Table 3. AFG preliminary evaluation on the example.

Metric	Value
#Reachable functions (CS / CI)	354 / 208
#Call-graph edges (CS / CI)	489 / 353
#Pointers	1,239
#Points-to relations	1,328
RUPTA analysis time	149 ms
`#afg` fixed-point iterations	4
`afg` post-pass time	810 μs
End-to-end runtime	≈150 ms

and the user-identifier literal. Experiments ran on an Apple M-series laptop.

Results. Table 3 reports the end-to-end sizes and timings. The cross-user leak ground truth for this example is the set of three abstract objects reached by both users through the shared cache: the `Arc<Mutex<HashMap<String, String>>>` allocation itself (an `alloc::sync::ArcInner` via `alloc::boxed::{impl#0}::new`); the HashMap's bucket storage (`hashbrown::raw::alloc::inner::do_alloc`); and the cached answer `String` (`alloc::str::{impl#4}::to_owned`). Counting a flagged node as a true positive if it is in the ground truth and a false positive otherwise, `afg` reports 3 true positives and 0 false positives: it flags exactly these three objects, each tagged as reached from both UserA's and UserB's seeds.

6 Future Plan

The STPA analysis described in this paper over-approximates because it does not account for runtime access control policies that may prevent certain data flows. To address this, we plan to refine the static PAG results through dynamic analysis built on MadSim [15], a deterministic simulator for distributed systems in Rust. We will insert lightweight macros at sensitive sinks identified by STPA instead of instrumenting the entire application. A monitoring macro placed at functions such as Redis `get` and `set` will capture the current stack trace and log memory accesses during execution. By running the application under the MUMP setup with multiple simulated users, we can observe the actual memory and function scopes accessed by each user under the application's access control policies. Comparing these runtime scopes with the static PAG allows us to remove edges that correspond to infeasible data flows, reducing false positives in the overlap detection. A path that looks unused in one run may be used in another. The dynamic refinement stage therefore *down-weights* edges instead of deleting them, and AFG still reports them as possible leaks. The schedule-driven testing stage then replays these edges under controlled thread and event schedules, catching leaks that only appear under rare orderings.

A separate challenge is handling LLM API calls during taint propagation. Since the internal behavior of LLM components is not available for static analysis, these calls act as black boxes in the PAG. Our current approach conservatively treats the output of an LLM API call as tainted by all of its inputs, which is sound but may introduce false positives. In future work, we plan to investigate API-level specifications from LLM service providers, such as input and output schemas. These specifications could be used to build summary models that approximate the taint behavior of black-box components. This would allow STPA to reason about LLM calls with greater precision without requiring access to model internals.

We also plan to extend AFG with schedule-driven testing to confirm the data leaks identified by the static and dynamic stages. Using the refined PAG and the runtime scope information, we will generate specific thread and event schedules that exercise the suspicious overlapping data accesses. Running the application under these controlled schedules in MadSim will determine whether a leak is actually triggered under a concrete execution order, providing developers with reproducible test cases rather than abstract warnings.

For evaluation, we plan to apply AFG to 10–12 popular open-source LPAs drawn from our previous work [9, 10], including projects that use LLM APIs for chat, code completion, and document retrieval. Ground truth combines manual audit of shared-state sink points with synthetic leak injection, where a known cross-user flow is inserted and AFG is checked for a matching report. The taint configuration files (MUMP dataset) in our repository already define source and sink specifications for several of these projects. We will measure the number of suspicious overlaps detected, the false positive rate after dynamic refinement, and the analysis time. We will also compare AFG with existing taint analysis tools for Rust. This comparison will assess whether user-tagged pointer analysis detects vulnerabilities that traditional source-sink approaches miss.

7 Conclusion

This paper presented AFG, with MUMP and STPA as its core contributions. MUMP automatically identifies taint sources by matching function signatures, removing the need for manual annotation. STPA detects cross-user data access by leveraging MUMP's tainted sources. Challenges remain in reducing static over-approximation, modeling LLM black-box components, and confirming suspected leaks through controlled scheduling. Addressing these will determine whether automated user-scoped taint analysis can become a practical security tool for multi-user LLM-integrated software.

Acknowledgments

This work was supported in part by the CAHSI-Google Institutional Research Program and in part by the NSF under Grant No. 2451516; we express our sincere appreciation for their support.

References

[1] 2024. Taint Analysis Configuration — Clang 19.0.0git Documentation. https://clang.llvm.org/docs/analyzer/user-docs/TaintAnalysisConfiguration.html. Accessed 22-03-2024.

[2] 2026. GitHub - apace-lab/AFG — github.com. https://github.com/apace-lab/AFG. [Accessed 30-04-2026].

[3] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, and Jia Song. 2019. Dynamic Taint Analysis Tools: A Review. *International Journal of Computer Science and Security (IJCSS)* 13, 6 (2019), 231–244.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/2594291.2594299

[5] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1687–1700. doi:10.1145/3243734.3243823

[6] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 196–206. doi:10.1145/1273463.1273490

[7] Pascal Costanza. 2008. Context-Oriented Programming in ContextL: State of the Art. In *Celebrating the 50th Anniversary of Lisp (LISP50)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1529966.1529970

[8] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Trust and Trustworthy Computing (TRUST 2012) (Lecture Notes in Computer Science, Vol. 7344)*. Springer, 291–307.

[9] Julia Gomez-Rangel, Young Lee, and Bozhen Liu. 2025. Security in the Wild: An Empirical Analysis of LLM-Powered Applications and Local Inference Frameworks. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. IEEE, 149–159.

[10] Julia Gomez-Rangel, Alvaro Vazquez, Young Lee, Kadir Alpaslan Demir, and Bozhen Liu. 2025. Rising Fast, Prone to Risk: How Open-Source LLM-Powered Apps Are Designed and Secured. In *Proceedings of the 2025 International Workshop on Artificial Intelligence× Software Engineering (AIxSE). IEEE.*

[11] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-To and Taint Analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 102 (2017). doi:10.1145/3133926

[12] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/2151024.2151042

[13] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. 2024. A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24)*. Association for Computing Machinery, New York, NY, USA, 60–72. doi:10.1145/3640537.3641574

[14] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. Association for Computing Machinery, New York, NY, USA, 725–739. doi:10.1145/3453483.3454073

[15] MadSim Contributors. 2023. MadSim: Magical Deterministic Simulator for Distributed Systems in Rust. https://github.com/madsim-rs/madsim. Accessed 06-10-2023.

[16] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 65–80.

[17] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*. Internet Society, San Diego, CA, USA.

[18] Jacques A Pienaar and Robert Hundt. 2013. JSWhiz: Static Analysis for JavaScript Memory Leaks. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–11.

[19] Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. 2014. CloudSafetyNet: Detecting Data Leakage Between Cloud Tenants. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*. Association for Computing Machinery, New York, NY, USA, 117–128.

[20] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy (SP)*. IEEE, Washington, DC, 317–331. doi:10.1109/SP.2010.26

[22] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. IEEE, 101–104.

[23] Jie Zhang, Cong Tian, and Zhenhua Duan. 2019. FasTDroid: Efficient Taint Analysis for Android Applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 236–237.

Received 2026-03-11; accepted 2026-04-15