ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis

Stanford InfoLab

ABSTRACT

In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit them programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an *ACIDRain attack*, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent interleavings. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory.

1. INTRODUCTION

For decades, database systems have been tasked with maintaining application integrity despite concurrent access to shared state [39]. The serializable transaction concept dictates that, if programmers correctly group their application operations into transactions, application integrity will be preserved [34]. This concept has formed the cornerstone of decades of database research and design and has led to at least one Turing award [2, 40].

In practice, the picture is less clear-cut. Some databases, including Oracle's flagship offering and SAP HANA, do not offer serializability as an option at all. Other databases allow applications

```
1   def withdraw(amt, user_id):           (a)
2     bal = readBalance(user_id)
3     if (bal >= amt):
4       writeBalance(bal − amt, user_id)
```

```
1   def withdraw(amt, user_id):           (b)
2     beginTxn()
3     bal = readBalance(user_id)
4     if (bal >= amt):
5       writeBalance(bal − amt, user_id)
6     commit()
```

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the `withdraw` function could both read balance $100, check that $100 ≥ $99, and each allow $99 to be withdrawn, resulting $198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as `SELECT FOR UPDATE` is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper, we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.

to configure the database isolation level but often default to non-serializable levels [17, 19] that may corrupt application state [45]. Moreover, we are unaware of any systematic study that examines whether programmers correctly utilize transactions.

For many applications, this state of affairs is apparently satisfactory. That is, some applications do not require serializable transactions and are resilient to concurrency-related anomalies [18, 26, 48]. More prevalently, many applications do not experience concurrency-related data corruption because their typical workloads are not highly concurrent [21]. For example, for many businesses, even a few transactions per second may represent enormous sales volume.

However, the rise of the web-facing interface (i.e., API) leads to the possibility of increased concurrency—and the deliberate exploitation of concurrency-related errors. Specifically, given a public API, a third party can programmatically trigger database-backed behavior at a much higher rate than normal. This highly concurrent workload can trigger latent programming errors resulting from incorrect transaction usage and/or incorrect use of weak isolation levels. Subsequently, a determined adversary can systematically exploit these errors, both to induce data corruption and induce un-

desirable application behavior. For example, the code in Figure 1 demonstrates a simple withdrawal function that checks whether a user has sufficient funds in their bank account. In Figure 1a, the code could exhibit anomalous behavior under concurrent execution, allowing the account to be overdrawn. Moreover, even after adding transaction logic as in Figure 1b, concurrent execution could elicit the same behavior under weak isolation.

These latent programming errors represent a potential security vulnerability, and the threat of systematic exploit is not theoretical: on March 2nd, 2014, the Flexcoin Bitcoin exchange was subject to such a concurrency-related attack:

> The attacker... successfully exploited a flaw in the code which allows transfers between Flexcoin users. By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, before balances were updated. This was then repeated through multiple accounts, snowballing the amount, until the attacker withdrew the coins [1].

As a result of this attack, all Bitcoins stored in the Flexcoin exchange were stolen, all users lost their stored Bitcoins, and the exchange was forced to shut down. This type of incident is not isolated; we are aware of several additional reports of malicious concurrency-related attacks, largely targeting Bitcoin and cryptocurrency exchanges [51, 55]. As web applications increasingly host valuable and sensitive data, attacks such as these may even become more common.

In this paper, we investigate the causes, detection, and prevalence of concurrency-related attacks on database-backed web applications, which we collectively title *ACIDRain* attacks.[1] We more formally define ACIDRain attacks, develop an analysis technique for detecting vulnerabilities to ACIDRain attacks, and apply this technique to a set of self-hosted eCommerce applications, identifying 22 vulnerabilities spanning over 2M websites. All 22 vulnerabilities manifest under the default isolation guarantees of popular transactional databases including Oracle 12c, and 17 vulnerabilities—due to incorrect transaction usage—manifest even under the strongest transactional guarantees offered by these databases.

To begin, we define a threat model for ACIDRain attacks. We consider attacks that trigger two kinds of *anomalies*, or behaviors that could not have arisen in a serial execution. First, if the database does not provide the application with serializable isolation (either because the database is not configured to do so or the database does not support serializability), then concurrently-issued transactions may lead to non-serializable behavior. We call these races due to database-level isolation level settings *level-based isolation anomalies*. Second, if the application does not correctly scope, or encapsulate, its logic using transactions, concurrent requests to the application may lead to behavior that would not have arisen sequentially. We call these races due to application-level transaction specification *scoping isolation anomalies*. The the impact of each of these types of anomalies is application-dependent. As a result, we examine a specific class of applications in this paper: popular eCommerce platforms, such as OpenCart [7], Spree Commerce [15], and WooCommerce [16].

We use this threat model to develop a cross-language analysis methodology to detect potential ACIDRain attacks. Web applications are written in a variety of languages and using a variety of

---

[1]Like acid rain in the Earth's atmosphere, ACIDRain attacks may be difficult to detect; an ACIDRain attack manifests in the form of regular API calls and resulting application and database activity, albeit at elevated levels of concurrency. This elevated concurrency triggers vulnerabilities resulting from incorrect use of ACID transactional databases, leading to corrupted data and/or more serious application compromise (e.g., stolen goods).

programming frameworks (e.g. Ruby on Rails). As a result, an analysis tool that operates on a per-language basis will have inherently limited applicability. Instead, we exploit the fact that our target applications are all *web-based* and *database-backed*. We analyze actual SQL traces (i.e., logs) using a new approach called *Abstract Anomaly Detection* (2AD). 2AD efficiently identifies potential level-based and scope-based anomalies that could arise from concurrently (re-)executing a set of API calls appearing in a given trace. This search space is enormous. Therefore, to enable efficient search, we extend the theory of weak isolation [17] to reason about both API calls and about re-executions. 2AD uses this theory to construct an *abstract history* that can be efficiently checked, representing the infinite space of concurrent schedules in a finite data structure.

Using 2AD analysis, we perform an audit of 12 popular self-hosted eCommerce platform applications, several of which are commercially supported, written in four languages using four different frameworks. We explore three attacks targeting invariants common to most eCommerce applications: attacks that allow users to steal items during checkout, to reuse gift cards to receive free items, and to corrupt store inventory ledgers. Using 2AD, we detect 22 new ACIDRain attacks. For example, in Magento [6], OpenCart [7], and Oscar [8], users can buy a single gift card, then spend it an unlimited number of times by concurrently issuing checkout requests. The total scope of the vulnerabilities we discover spans approximately 2M websites that use this software today, representing over 50% of all eCommerce websites (Section 4.2.1).

We subsequently discuss strategies for remediating these attacks and discuss our experiences reporting these vulnerabilities to developers, who have confirmed several thus far. We evaluate which databases provide sufficiently strong isolation guarantees to prevent these attacks. Of the 22 vulnerabilities, 17 occur due to incorrect transaction usage and are therefore not preventable without substantial code modification. We investigate common program behavior among vulnerable and non-vulnerable code paths and present constructive strategies for preventing attacks.

The remainder of this paper proceeds as follows. Section 2 defines ACIDRain attacks. In Section 3, we develop and formally motivate the 2AD analysis theory. Section 4 describes our experiences detecting and exploiting real vulnerabilities in eCommerce applications. Section 5 discusses related work, and Section 6 concludes.

2. ACIDRain ATTACKS

In this section, we define ACIDRain attacks more precisely and describe the threat model we consider in this paper.

Target Environment. We focus on attacks on web applications—applications that expose functionality to third-parties via programmatically accessible APIs, both over the Internet and via related protocols such as HTTP and REST. This applies to every website on the Internet. Our primary property of interest is that it must be possible to programmatically trigger API calls.

We are specifically interested in web applications that use databases to mediate concurrent access to state. A web application that executes requests serially is not subject to the attacks we consider here; however, concurrent request processing is common among web servers including Apache and Nginx. We consider transactional databases that allow users to group their operations into transactions consisting of ordered sequences of operations [43]. The database in turn provides varying *isolation guarantees* regarding the admissible interleavings of operations across transactions [17].

Attack Definition. We define an *ACIDRain attack* on a database-backed web application as an exploit allowing an attacker to elicit undesirable application behavior by issuing concurrent requests to

trigger non-serializable access to database-managed state. There are several salient characteristics of this formulation. First, we are interested in errors arising from access to *database-managed* state; we do not consider vulnerabilities that may arise due to access to state that is unknown to the database (e.g., a local file). Furthermore, we are interested in errors arising from *concurrent* access; we do not consider vulnerabilities that may arise during sequential access (e.g., failure to check permissions). Finally, the severity of an attack is application-specific; some concurrent behaviors may be benign, while others may be catastrophic. These characteristics shaped our problem formulation below. An application is vulnerable to an ACIDRain attack if two conditions are met:

C1: Anomalies possible. Under concurrent API access, the application may exhibit behaviors (i.e., *anomalies*) that could not have arisen under a serial execution.

A concurrency-related attack arises in the presence of behaviors that could not have occurred under a serial execution. These behaviors are effectively race conditions across concurrent operations, or, in the parlance of transaction processing, anomalies [17]. We consider two kinds of anomalies:

First, a transaction issued by a web application may exhibit non-serializable behavior during concurrent API calls. That is, while the gold standard of transaction isolation (serializable isolation) guarantees equivalence to some serial execution of transactions, not all databases will enforce serializability. Some databases do not provide serializability as an option at all, while others allow applications to select a weaker isolation mode [17, 19]. Under weaker isolation levels, transactions are subject to an array of behaviors that cannot occur under serial execution, the exact set of which depends on the particular isolation level and database [17]. We call these conventional isolation anomalies *level-based isolation anomalies* as they arise due to the database executing under non-serializable isolation levels.

Second, independent of the isolation level used, the transaction programming model requires the application to correctly encapsulate its logic within transactions. In the absence of explicit BEGIN TRANSACTION and COMMIT/ABORT commands, by default, many databases such as MySQL and PostgreSQL automatically execute each SQL operation as a separate transaction. As a result, if a web application performs multiple database operations without using transactions while servicing a single API request, then concurrent API requests may result in behavior that could not have arisen during a serial execution of API calls. We call these isolation anomalies arising from a lack of transactional encapsulation *scope-based isolation anomalies*. In this paper, we consider scoping at the level of individual API calls.

Given a set of isolation anomalies, we must determine whether any of these anomalies result in significant application behavior:

C2: Sensitive invariants. The anomalies arising from concurrent access lead to violations of application *invariants*.

In general, per Kung and Papadimitriou [45], every anomaly is problematic for *some* application; however, for a *given* application, is a given anomaly problematic? Again borrowing from the classical transaction processing literature, we capture key application properties via *invariants*, or logical predicates capturing an application's consistency criteria [34]. For example, an application might have an invariant that user IDs within a database are unique. Another application might specify that total revenue equals the sum of total orders placed. Each invariant is susceptible to violation under a particular set of anomalies.
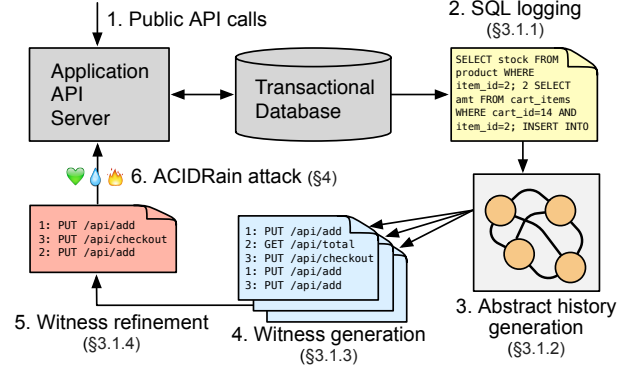

Figure 2: 2AD workflow to discover ACIDRain attacks.

To detect an application's vulnerability to ACIDRain attacks, we must identify potential anomalies, then determine whether application invariants are susceptible to the anomalies. Towards the former task, in the next section, we present a cross-platform methodology (based on analysis of traces of live database activity) that automatically identifies potential isolation anomalies. Determining invariants is more complicated, requiring either user interaction, invariant mining, or program analysis [32, 33]. As a result, in this paper we focus on a specific, concrete set of invariants found in eCommerce applications and examine a set of popular eCommerce applications to determine their susceptibility to attacks on these key invariants.

Threat model. We assume that an attacker can only access the web application via concurrent requests against publicly-accessible APIs (e.g., HTTP, REST). That is, to perform an ACIDRain attack, the attacker does not require access to the application server, database server, execution environment, or logs. Our proposed analysis techniques (Section 3) use full knowledge of the database schema and SQL logs, but, once identified, an attacker can exploit the vulnerabilities we consider here using only programmatic APIs.[2] This threat model applies to most Internet sites today.

3. 2AD: DETECTING ANOMALIES

ACIDRain attacks stem from anomalies that occur during concurrent execution. Detecting these anomalies is challenging. Many potential anomalies are never triggered under normal operation due to limited concurrency, rendering simple observation ineffective. We could use static analysis tools [50] to analyze an application's susceptibility to attacks. However, web applications are written using a variety of frameworks and languages. As a result, static analysis tools would necessarily have limited applicability.

To address these challenges, we developed a new, cross-platform methodology for detecting potential level-based and scope-based anomalies in web applications by analyzing logs of typical database activity. We call this approach *Abstract Anomaly Detection (2AD)*. Figure 2 shows an overview of the 2AD workflow.

Overview. The core idea behind 2AD is to execute API calls against a live application and database to generate a (possibly sequential) trace of database activity, then analyze the trace for potential anomalies that could arise under concurrent execution. This approach leverages the facts that our target applications all *i.*) expose API endpoints (e.g., via HTTP) that can be triggered programmati-

---

[2]That is, to efficiently identify vulnerabilities, our analysis makes use of non-public information in the form of database logs (e.g. SQL traces) and database schemas. However, the vulnerabilities themselves can be exploited without this private knowledge.

cally and *ii.*) are backed by a SQL database, allowing a common logging environment. The analysis determines whether (re-)executing API requests concurrently might yield anomalies, subsequently reporting the database tables and API calls that are susceptible to anomalies and that could be used in an ACIDRain attack.

While conceptually simple, this dynamic analysis, which we describe in detail in the remainder of this section, requires considerable work to achieve for two primary reasons:

First, existing models for database isolation reason about anomalies in a *particular* concurrent execution (i.e., a *history* [17, 25]). In contrast, we want to know whether anomalies are possible under *any* potential concurrent execution of a group of transactions generated by (possibly serial) API calls. Thus, we must generalize from concrete traces (Section 3.1.1) to possible concurrent interleavings of the operations in those traces, which we call trace *expansions*. Expansions allow API calls to be repeated, possibly with different inputs; thus, the set of expansions is infinite. We develop a new approach to simultaneously reason about all possible expansions. We introduce the concept of an *abstract history*, a finite graph representing all expansions of a given trace (Section 3.1.2). We provide a mechanism to "lift" a concrete trace to its corresponding abstract history and prove the equivalence of anomaly detection over the lifted abstract history and over the entire space of expansions (Section 3.1.3).

Second, to detect scope-based anomalies, we need to reason about behavior *across* transactions within the same API call. To do so, we extend Adya's theory of transaction isolation [17] to allow reasoning about API calls (Section 3.1.3). Roughly, this corresponds to adding API "supernodes" to the transaction conflict graph and our abstract history. We subsequently extend Adya's theory of weak transaction isolation to allow refinement of possible anomalies in abstract histories, including API calls (Section 3.1.4).

As we discuss in Section 3.1.3, 2AD is complete with respect to the trace: if there is a potential anomaly in a trace expansion, 2AD will find it. 2AD will also provide a corresponding *witness*, or concrete trace, demonstrating non-serializable behavior. However, depending on the isolation level of the database and execution environment of the application (e.g., due to application-level locking), some anomalies are impossible to trigger. Thus, 2AD leverages an *witness refinement* step (Section 3.1.4) to reduce false positives.

In the remainder of this section, we provide intuition and algorithms for performing 2AD. We provide a detailed formalism for 2AD (including proofs) in Appendix A. We describe trace generation in Section 3.1.1, abstract history lifting in Section 3.1.2, anomaly detection in Section 3.1.3, witness refinement in Section 3.1.4, and the benefits and limitations of the 2AD approach in Section 3.2.

3.1 2AD Concepts and Procedures

3.1.1 Trace Generation

2AD uses traces of normal application behavior to identify potential non-serializable expansions. Given that web applications are written in a number of languages and frameworks, we gather traces (logs) from the database rather than the application. These logs can be generated from normal activity, or generated for the explicit purpose of anomaly detection. For example, to check for anomalies in the checkout process of an eCommerce application, a 2AD penetration tester could add items to the store cart, provide address and payment details, then place an order.

From the database logs, we extract the sequence of transactions generated by each API call. At a high level, each transaction consists of a sequence of read and write operations in the database; we extract this sequence as well as the variables (e.g., columns) upon

```
1   def add_employee(first, last):          (a)
2       beginTxn()
3       count = readCount('employee')
4           .whereFirstName(first).whereLastName(last)
5       if count == 0:
6           write('employee', first, last, 0)
7       commit()

8   def raise_salary(amt):
9       write('employee').incrementSalary(amt)
10      beginTxn()
11      count = readCount('employee')
12      write('salary').updateTotal(count * amt)
13      commit()
```

```
1   BEGIN TRANSACTION                        (b)
2   SELECT COUNT(∗) FROM employees WHERE
        first_name='John' AND last_name='Doe'
3   INSERT INTO employees (first_name, last_name,
        salary) VALUES ('John', 'Doe', 50000)
4   COMMIT
5   UPDATE employees SET salary=salary+1000
6   BEGIN TRANSACTION
7   SELECT COUNT(∗) FROM employees
8   UPDATE salary SET total=total+3000
9   COMMIT
```

Figure 3: (a) Simplified code corresponding to two functions: one to add a new employee if the first and last names are unique (`add_employee`, lines 1-7) and one to give a raise to all employees and record the new total cost of all salaries. (`raise_salary`, lines 8-13). (b) A sample SQL database log from using the functions in (a) to add a third employee, "John Doe," to the database and then give all employees a raise of 1000.

which each read and write operation acts.[3] In the next section, we describe how to use this trace as a "seed" for, in effect, simulating the execution of all possible concurrent API calls.

In practice, logs may contain commands interleaved from concurrent API calls. Thus, we require that each command logged be associated with the specific API call that generated it. This association can be obtained in a variety of ways; one of the simplest approaches is to match the timestamp of the log with the timestamp of the API call.

Figure 3a shows a simple example payroll application implementing functionality to add an employee (who has a first name, last name, and yearly salary) to a database, along with functionality for giving a raise to all employees and recording the new total cost of all employee salaries. We will use this as our running example throughout this section. Figure 3b shows logs that could result from executing these two functions serially. While simple, this example highlights similar problems to those we have encountered in the more complex applications that we discuss in Section 4.

3.1.2 Abstract History Generation

Given a concrete trace generated by API calls, we determine whether concurrently executing a set of calls to the same APIs might result in non-serializable behavior. The primary challenge here

---

[3]As we are interested in database traces in SQL, we must reason about operations over sets of database records. Adya [17] provides a detailed discussion of predicate-based operations that operate over sets; we adopt his formalism by modeling predicate-based read and write operations that pertain to multiple records as single operations.
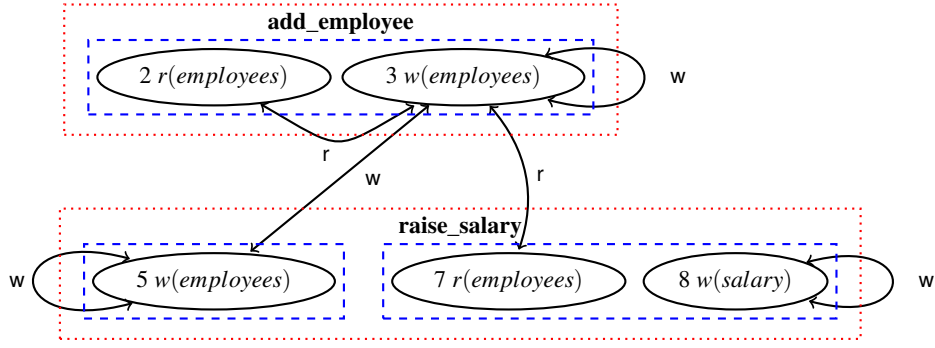
Figure 4: The abstract history corresponding to the trace in Figure 3b. Solid ellipses correspond to operations, dashed rectangles to transactions, and dotted rectangles to API calls. Edges are labeled with the type of conflict, and operations are labeled with the corresponding line from the trace in Figure 3b. There is no edge between node 5 and nodes 2 and 7 because these `COUNT` queries do not conflict with the update in 5. Per Theorem 1, each non-trivial cycle in the abstract history corresponds to a potential anomaly.

```
1  a1*: UPDATE employees SET salary=salary+1000
2  a2:  BEGIN TRANSACTION
3  a2:  SELECT COUNT(*) FROM employees WHERE
          first_name='John' AND last_name='Doe'
4  a2:  INSERT INTO employees (first_name, last_name,
          salary) VALUES ('John', 'Doe', 0)
5  a2:  COMMIT
6  a1:  BEGIN TRANSACTION
7  a1*: SELECT COUNT(*) FROM employees
8  a1:  UPDATE salary SET total=total+3000
9  a1:  COMMIT
```

Figure 5: Example non-serializable witness generated from the abstract history in Figure 4. The asterisks mark the operations used as the seed pair to find the cycle. The labels on the left correspond to the API call that generated the operation. This trace represents the scenario in which a new employee named "John Doe," who will be the third employee, is added to the database concurrently with salaries being raised by 1000.

is that existing theories of isolation pertain to concrete traces, or histories, of transactions [17, 25]. These theories can tell us whether a given execution obeys a given isolation level. However, we would like to reason about the infinite space of concurrent executions.

Thus, instead of reasoning about concrete histories directly, we introduce a concept that we call an *abstract history*. The abstract history is a finite multigraph (i.e., allows multiple edges between the same pair of nodes) that represents the set of all possible expansions of a given trace. We show how to detect whether an anomalous expansion of the trace exists and generate example expansions via a series of short walks over this abstract history graph:

An abstract history consists of three types of nodes—operation nodes, transaction nodes, and API nodes—and two types of edges—write edges and read edges. Transaction nodes are supernodes encapsulating each operation in the transaction, while API nodes are supernodes encapsulating all transactions in the API call. Undirected edges link two operation nodes and induce edges between the corresponding supernodes. For example, given operation o_1 in transaction t_1 and API call a_1 along with operation o_2 in transaction t_2 and API call a_2, a write edge between o_1 and o_2 induces a write edge between t_1 and t_2 as well as a_1 and a_2. As the abstract history is a multigraph, two nodes are allowed to have multiple edges between them; this occurs only between transaction nodes and API nodes, not operation nodes.

Given a trace consisting of a set of API calls, each containing a set of transactions, along with each transaction's sequence of read and write operations, we construct an abstract history as follows: first, create an operation node for each operation, a transaction node for each transaction, and an API node for each API call. Group the operations by transaction and the transactions by API call.

We say that two operations *conflict* if they access the same data items (i.e., columns or logical variables, *not* values) and at least one operation is a modification (i.e., write). (Note that in SQL, predicate- and set-based constructs such as `COUNT` and `UPDATE` conflict according to the predicates involved for each statement [17].) For each pair of conflicting operations, add an undirected *read edge* between them if one is a read and an undirected *write edge* between them if both are writes. As described above, these edges induce edges between the corresponding transaction and API nodes. Moreover, all types of nodes are allowed to have self-loops. Figure 4 shows the abstract history generated from Figure 3b.

When constructing the abstract history, we only record the tables and columns accessed, not the exact values in the operations. This allows us to collapse multiple instances of the same API call with the same access pattern into one API node, reducing the size of the abstract history and improving search speeds. In contrast, multiple calls to the same logical API function that result in different access patterns (e.g., because one call encountered invalid input) would be represented by different API nodes in the abstract history.

In summary, the abstract history is represented by a multigraph, with nodes for each operation, supernodes of operations for each transaction, and supernodes of transactions for each API call. Undirected write and read edges capture interactions between pairs of writes and pairs of reads and writes, respectively. Intuitively, the abstract history captures all possible concurrent interleavings of the API calls. We can use it to check for potential anomalies without enumerating all interleavings (Section 3.1.3).

3.1.3 Witness Generation

We can simultaneously find both level-based and scope-based non-serializable expansions of the original trace (i.e., witnesses) using the abstract history. Just as cycles among transactions indicate anomalies in traditional formalism for reasoning about concrete histories [17,25], cycles of API nodes in the abstract history correspond to potentially anomalous behavior in API calls.

However, simply checking for undirected cycles in the abstract history is insufficient. For example, concurrently executing an API call containing a single transaction $T_1 : w(x_1)$ never results in non-serializable behavior, but the corresponding abstract history contains

a cycle. We say this cycle is *trivial* because it only contains one operation per API node. In Figure 4, the cycle formed by the write self edge on operation 3 is also trivial. In contrast, we are interested in *non-trivial abstract cycles*, or cycles of edges between API nodes that contain edges induced by two or more distinct operations residing within a single API node. (Note that this definition does not preclude repetition of API nodes.) For example, the two self edges on operations 5 and 8 in Figure 4 form a non-trivial abstract cycle.

Searching for these non-trivial abstract cycles is sufficient to detect the presence of anomalous expansions. Specifically:

Theorem 1. (Informal) For a given trace T, there exists an expansion of the trace that results in anomalous behavior if and only if there is a non-trivial abstract cycle in the abstract history between API nodes corresponding to T.

Theorem 1 implies we can simply walk the abstract history graph and search for cycles to find anomalies. Remarkably, non-trivial cycle detection in this lifted form captures *any* and *all* possible anomalous expansions of a given concrete history. That is, this theorem states that this method is complete with respect to the trace—if a non-serializable expansion exists, 2AD will find it.

The soundness property implied by this theorem (i.e., that there exists an anomalous expansion for each cycle) assumes that all anomalous expansions (witnesses) are in fact achievable via some concurrent re-execution of the API calls. However, if the database provides *any* isolation guarantees, or if programs perform complex control flow or perform concurrency control external to the database, 2AD as currently stated will report false positives. Thus, we introduce a "witness refinement" step in in Section 3.1.4.

Appendix A provides a formal statement and proof of this theorem as well as a more detailed description of how to generate witness traces from a given non-trivial abstract cycle. Finally, recall that Theorem 1 refers to the soundness and completeness of 2AD as a method of detecting potential *anomalies* in a given trace of transaction and API call activity. Thus, 2AD reasons about observed traces of an application's behavior and potential re-execution of API calls that produced those traces, not the safety of the entire application (e.g., changing values or control flow that does not appear in the trace itself; see also Section 3.2).

Witness-Finding Algorithm. Given our goal of finding non-trivial abstract cycles, 2AD starts by selecting a *pair* of operations in the same API call (o_1 in transaction t_1 and API call a_1, o_2 in transaction t_2 and API call a_1, o_1 precedes o_2 in the serial ordering of a_1). We search for non-trivial abstract cycles among the API nodes, constraining the first edge in the cycle to have o_1 as an endpoint and the last edge in the cycle to have o_2 as an endpoint. To find level-based anomalies, we examine only pairs where $t_1 = t_2$. To find scope-based anomalies, we look at pairs where $t_1 \neq t_2$.

Each of these connectivity queries can be answered via simple depth-first search (DFS). Moreover, each cyclic path found can be used to build a witness trace of the potentially anomalous execution. Informally, we construct a witness by walking the cycle from o_i to o_j and recording all operations that we encounter, then add the remaining operations of the API nodes used in the cycle such that the final history respects the ordering of operations within each API call (Appendix A, Lemma 4). The full witness finding algorithm analyzes all pairs of operations within the same API call.

Example. There are several non-trivial cycles in Figure 4. The path including operations 5, 3, and 7 forms a cycle between the two API nodes corresponding to an anomaly which results in an employee being counted in the raised total salary amount but not receiving a raise. Figure 5 shows a witness to this anomaly. Operations 5 and 7 are in different transactions, making this is a scope-based anomaly.

The self-loop cycle on API call `add_employee` between operations 2 and 3 corresponds to a violation of the uniqueness property on employee names. However, since these operations are in the same transaction, we must consider the isolation level of the database, as discussed in the next section.

Runtime. Given a trace containing p operations, the abstract history contains $O(p)$ operation nodes and $O(p^2)$ edges, so DFS requires $O(p^2)$ time. Analyzing all pairs of operations requires $O(p^4)$ time. However, this worst case runtime is rarely reached. If we divide the operations into p_r reads and p_w writes, then there are at most $p_w^2 + p_w p_r$ edges, since each edge requires at least one endpoint to be a write. If $p_w \ll p_r$, this is approximately $p_w p_r$, or $O(p^2 p_w p_r)$. For the applications we analyze in Section 4, the number of edges is roughly $O(p)$, not $O(p^2)$. Furthermore, it is often unnecessary to examine all pairs of operations for anomalous behavior. We can leverage user-provided input to focus the search on anomalies involving specific tables and columns and achieve interactive runtimes (Section 4.2.3). With this optimization, our prototype 2AD analysis tool implemented in Python completed within 10 seconds for every application we analyzed.

3.1.4 Witness Refinement

Thus far, our 2AD analysis has operated under the assumption that all expansions of traces can be achieved via concurrent execution of the corresponding API requests. In effect, this corresponds to an application that *i.*) executes under a database with no isolation, *ii.*) can execute API calls with arbitrary (well-typed) values, and *iii.*) is able to reliably generate the exact same read-write transaction activity from concurrent API calls as in the input trace. In practice, these properties may not hold. First, databases provide *weak isolation* guarantees that do not guarantee serializable execution but nevertheless restricts allowable concurrent executions [17]. Second, 2AD's use of traces treats applications as black-box transaction generators: while some applications can reliably (re-)generate concurrent transactions, others may have more complex read-write logic and/or application-level concurrency control mechanisms that restrict the space of achievable schedules. As a result, 2AD's cycle generation may produce false positives, or anomalous witness traces that are not actually possible to produce under concurrent execution.

To reduce these false positives, thereby improving the soundness of 2AD analysis, we introduce an optional *witness refinement* step. In this step, we encode additional knowledge about the space of achievable histories in the form of restrictions on witnesses. There are two main sources of knowledge we consider:

1.) Isolation-Based Refinement. Different isolation levels allow different types of anomalies. For example, Read Uncommitted isolation disallows witnesses consisting only of write-write conflicts. Therefore, if the database operates under Read Uncommitted isolation, we can modify the 2AD cycle generation protocol to ignore cycles consisting only of undirected write edges. We can in fact capture the entire theory of weak isolation including common models such as Snapshot Isolation [17] via witness refinement by encoding the corresponding restrictions on the witness histories. To avoid enumerating all cycles (potentially exponential in number), we modify the DFS to memoize refinement information.

As a concrete example of isolation-based refinement, again consider Figure 4 and the cycle in API call `add_employee` between operations 2 and 3. Since these operations are in the same transaction, we must consider the isolation level of the database. Operation 2 is a predicate read, thus this anomaly will still be possible under Read Uncommitted, Read Committed, and Repeatable Read isolation levels. Serializable isolation would disallow this anomaly.

Performing refinement of this type requires knowledge of the isolation level at which the application will be run, as well as database schema information. The schema information allows 2AD to distinguish reads on unique keys from predicate reads (as the two are treated differently under RR and SI).

2.) Application-Level Refinement. We can also perform witness refinement given information about the application and execution environment. For example, if we know that the application is deployed in an environment that limits the number of concurrent API requests to N (e.g., due to web server configuration such as process pool size), we can ensure that cycles in 2AD witnesses span at most N API calls. In addition, 2AD's abstract histories are value-agnostic and do not account for control flow within a program; in effect, 2AD's abstract history construction process assumes that each variable read and written can assume arbitrary values. However, there are often dependencies (e.g., $y = x + 1$) between the values that variables assume. In general, analyzing and encoding *all* program logic into the 2AD refinement step is highly challenging, and, in the limit, requires static analysis of the source program.

In our experimental study, it was faster to attempt to trigger a reported anomaly and then find the associated program logic preventing the vulnerability than to preemptively add refinements. For the web applications we seek here—many of which have simple Create-Read-Update-Destroy (CRUD) semantics—complex application-level refinement was not necessary to detect our target anomalies.

3.2 2AD Overview and Discussion

Benefits. In the parlance of programming languages, 2AD is a dynamic analysis [50], in that it uses traces from live applications as the basis of analysis. This is a natural fit for database-backed applications: it is a simple engineering exercise to collect query logs, and a relatively straightforward task to correlate log entries with API calls for many of the frameworks we study. Database schema information is similarly easy to collect. Although we have performed our analyses in a test environment (Section 4.2.1), 2AD is amenable to execution over production traces as well.

2AD is both language agnostic—allowing it to analyze many different applications, and database agnostic—requiring only that the database allow for command logging and support a SQL-like query language. This has proved useful in practice (Section 4).

Soundness and Completeness. As discussed in Section 3.1.3, 2AD is complete with respect to the trace. 2AD is as sound as its refinements; it will only report false positives based on isolation or application information it does not know about. As described in Section 4.2.5, a basic 2AD implementation was sufficiently sound to assist in finding vulnerabilities in real applications.

Limitations. 2AD analysis has several fundamental limitations. As 2AD only operates over database logs, it does not account for any program logic that enforces serializability or expansions unachievable due to constraints on values. As a result, 2AD may result in false positives; for example, a developer could use a global variable to lock a critical section of code instead of wrapping it in a transaction. To avoid this false positive, we would have to encode this information during trace refinement (e.g., via static analysis).

Moreover, 2AD analysis is only as thorough as the provided traces. If a given API call is not in the input trace, 2AD cannot check for anomalies involving the call. 2AD does not account for program behavior such as internal control flow that is not observable from traces. Thus, 2AD is well-suited to finding latent errors in common-case application behavior, but it will miss anomalies corresponding to rare or exceptional behavior not found in input logs.

In addition, 2AD only finds *anomalies*, not vulnerabilities. It is up to the programmer or an additional tool to ascertain whether a given anomaly may result in an ACIDRain attack. We discuss this process at length in the next section.

Extensions. There are a number of promising extensions to 2AD that we believe can capture more sophisticated transaction usage patterns. For example, under mixed isolation modes (e.g., one transaction running at Read Committed and another at Snapshot Isolation), we can annotate transaction nodes with allowable isolation guarantees, then propagate these labels during trace refinement (e.g., a transaction allowed to execute in SI but not RC will disallow Lost Update phenomena). In addition, by adding "sub-transaction" nodes (similar to nesting transaction nodes inside of API nodes) and modifying the detection procedure, we can extend 2AD to nested transactional (and, respectively, nested API call) models.

Summary. 2AD is a cross-language dynamic analysis that uses database traces to search for potential level and scoping anomalies under concurrent execution. Our choice to focus on database traces was motivated by our desire for a portable, lightweight tool that can analyze database-backed applications written in arbitrary languages. The decision to focus on database-level activity also allowed us to adapt decades of theory on weak isolation in detecting anomalies. Developing automated techniques for incorporating additional knowledge of application structure into trace refinement will allow more fine-grained analysis and is a worthwhile area for future work. However, despite its limitations, 2AD has proven a useful tool in analyzing real applications—the subject of the next section.

4. ACIDRain IN THE WILD

Having described how to use database traces to identify possible anomalies, in this section we describe how to use these this approach to detect vulnerabilities and subsequently perform ACIDRain attacks. We apply a prototype 2AD analysis tool to a suite of 12 eCommerce applications, identifying 22 new ACIDRain attacks. Section 4.1 describes how to produce vulnerabilities from anomalies, and Section 4.2 details our experience finding vulnerabilities in self-hosted eCommerce applications.

4.1 From Anomalies to Vulnerabilities

Isolation guarantees are a means towards protecting application integrity, or invariants over data. Provided transactions (resp. API calls) maintain application invariants in a serial execution, a serializable execution will also preserve those invariants. However, an anomalous execution *could* violate invariants and corrupt application state. When does this corruption actually occur?

For a given anomaly, there exists some application for which the anomaly violates an invariant [45]. Intuitively, if anomaly *a* occurs in a history *H*, we can create a new application whose transactions are the same as those in *H* and whose sole invariant is that "anomaly *a* never occurs." However, for a *given* application, the anomaly may or may not influence the application invariants. Thus, to use 2AD in an ACIDRain attack, we must establish a correspondence between potential anomalies and invariant violations for a given application. This is challenging to do in general: for example, describing all program invariants is notoriously difficult and burdensome for programmers [33].

Shifting from the theoretical to the practical, identifying security-related invariants is less onerous than it may immediately seem. An attacker will likely target particular data records of value such as bank account balances, store inventory, tax records, and/or access control policies. Therefore, a security officer's role is to identify and ensure adequate protection of these critical assets. Thus, 2AD's

ability to highlight anomalies that affect particular data items (e.g., a table containing account balances) and determine the API calls that may trigger them (e.g., two concurrent withdrawal requests) allows users to determine which anomalies affect key program invariants. In the next section, we describe this attack process for three critical invariants found in popular eCommerce applications.

4.2 Attacking Self-Hosted eCommerce

To understand the prevalence of ACIDRain attack vulnerabilities across a range of applications, we turned to self-hosted eCommerce platforms (i.e., eCommerce platforms users deploy on their own servers, in contrast to hosted offerings like Shopify). Over 60% of the top 1M eCommerce sites are backed by these platforms [4]. Moreover, analyzing a particular class of application (eCommerce) allowed us to check the same invariants across a range of codebases, helping identify trends in vulnerability and prevention patterns.

4.2.1 Target Application Corpus

We selected a set of 12 eCommerce applications written in four languages based on popularity measures including GitHub stars and references in popular articles (Table 1). Stores use these applications by building a custom front end for customers while relying on the application for tasks such as catalog management and payment integration. This is similar to how a WordPress user might create a blog (and, indeed, our most popular application, WooCommerce, is actually a plugin for WordPress). Each application provides functionality for managing an online store, allowing users to browse a store catalog and place orders. Each application maintains inventory, a ledger of orders, and tracks order status. Store owners can view this data and perform administrative actions via separate interfaces. Table 1 summarizes the applications used, their deployments, and popularity on GitHub. While we could not find deployment numbers for all of the platforms chosen, according to builtwith.com [4], this set covers over 55% of eCommerce sites on the Internet. WooCommerce alone accounts for 39% of all online stores.

We chose eCommerce sites in part because they are among the most popular widely-deployed self-hosted web applications and also because they deal with money. (In contrast, we did not find any popular self-hosted banking applications.) We did not censor our selection but instead selected for prevalence and popularity alone. We report results from every application we tested.

The eCommerce sites' feature sets ranged considerably, but, as Table 5 (Appendix C) shows, most had common functionalities including functionality to track products and inventory, record customer activity, and set up promotions—functionality we target in the next section. Most importantly, they all shipped with sample store that was easy to configure and represented a basic first deployment that exercised core application functionality.

Applicability of results. In our analysis, we study application codebases that allow us to gather traces and verify vulnerabilities without performing attacks on sites in the wild, thereby avoiding committing criminal offenses under a variety of jurisdictions. This methodology leaves two questions unanswered:

First, if installed and configured according to directions, real online stores using each application we study will use the same functions and functionality described here, thus exposing themselves to the vulnerabilities we report. However, it is possible that none of the vulnerabilities we report actually exist in real sites—the 2M+ site operators using these applications may have fixed or otherwise mitigated these vulnerabilities. For example, each store owner could *i.*) modify the application code to properly encapsulate vulnerable functionality in transactions, *ii.*) make sure to only deploy their stores on databases that support serializability, and *iii.*) upgrade the isolation level of their databases from the non-serializable default isolation levels to serializability. Combined, these three actions would defend against attacks, as the correctly-scoped application transactions would exhibit serializable behavior.

We believe it is unlikely that all 2M+ sites running this code in the wild performed such modifications, especially as none of the above were mentioned any of these modifications in their documentation. However, we have not attempted to verify this fact and instead only report on application usage as directed.

Second, we only analyzed self-hosted eCommerce applications. According to builtwith.com [4], the majority of the remaining, prominent eCommerce application platforms are hosted; that is, popular platforms such as Squarespace and Shopify provide eCommerce-as-a-service. These hosted applications do not expose database access directly but instead surface application APIs to the public Internet. Thus, it is possible that these hosted eCommerce offerings are subject to the exact same vulnerabilities that many of their self-hosted peers exhibit in our study. One could attempt an attack on these hosted offerings by performing concurrent requests to a store hosted on a platform like Squarespace or Shopify using public-facing APIs. However, we have not attempted to do so and only report on self-hosted applications here.

4.2.2 Target Application Invariants

From this corpus of applications, we extracted a set of three critical invariants as targets for potential ACIDRain attacks. These three invariants by no means represent the entire set of eCommerce invariants that may be subject to attack, but this set applied to almost all applications in the corpus and served as a useful basis for a systematic study. The exact invariant depended on the specific semantics of each application but fell into one of three broad categories:

1.) Inventory Invariant. Each eCommerce site maintains its own bookkeeping of store inventory. Each product has an associated stock value (i.e., count of product remaining) that is decremented upon order completion to record that the associated stock is accounted for. We consider the invariant that a product's stock must be non-negative and that an item's final stock count reflect the orders placed for that item.[4] We selected this invariant due to its ubiquity and also because of its close correspondence to the canonical textbook example of an integrity violation due to concurrent bank account withdrawals resulting in corrupted or negative balances [42].

2.) Voucher Invariant. Nine out of twelve applications allowed administrators to create gift *vouchers* (i.e., gift cards), which have monetary value and/or a limit on the number of times the voucher can be used. We targeted the invariant that vouchers should not be used more than their specified limit. Violating this invariant amounts to overspending a voucher, effectively stealing from the store. The applications all process these vouchers internally, using database backed state instead of third-party payment processors.

3.) Cart Invariant. Each application exposed a shopping cart functionality, into which users place items and subsequently pay for them as part of an order. We target the invariant that the total amount charged for an order should reflect the total value of the goods in the order. While this invariant may seem obvious, we found that in several of the applications it was possible to add an item to the cart concurrent with checkout, resulting in the user paying for the original total of items in the cart, but placing a valid order including the new item as well. This allows users to obtain items for free. For example, a user might buy a pen and add a laptop to their cart during checkout, paying for the pen but placing an order for the pen

---

[4]Some applications allowed backorders, but we disabled that functionality.

App Name	Language	Web Deployments	GitHub Stars as of 3/21/17	Lines of Code	SQL Trace Size (Lines)
OpenCart [7]	PHP	298,399	3247	136544	1699
PrestaShop [10]	PHP	230,501	2287	189812	1422
Magento [6]	PHP	245,680	4198	1161281	801
WooCommerce [16]	PHP	1,979,504	3227	100098	1006
Spree [15]	Ruby	45,000	8268	56069	768
Ror_ecommerce [11]	Ruby	–	1106	17224	218
Shoppe [14]	Ruby	–	835	4062	152
Oscar [8]	Python	–	2427	31727	769
Saleor [12]	Python	–	828	8614	401
Lightning Fast Shop [5]	Python	–	423	25163	563
Broadleaf [3]	Java	–	889	163012	374
Shopizer [13]	Java	–	507	59014	845

Table 1: Summary of applications analyzed. Deployment information provided by builtwith.com [4] for all but Spree, where information is provided by the SpreeCommerce website. We were unable to find deployment numbers for the other applications. All Ruby applications were built using Rails, all Python applications were built using Django, and all Java applications were built using Spring as their respective frameworks. Lines of code only includes the lines in the target language, excluding other files such as Javascript or HTML.

and laptop. Unless an application operator specifically looks for mismatched order totals, this may be problematic, especially when order fulfillment is automated. Thus, violations of this invariant essentially allow customers to steal items from the store.

Table 3 (Appendix C) provides example formal statements of these invariants, sample traces showcasing how these anomalies manifest in the wild, and an example abstract history graph that might arise from a simplified eCommerce application.

4.2.3 Prototype 2AD Analysis Tool

We implemented a prototype 2AD analysis tool in Python following the approach in Section 3.[5] The prototype accepts SQL logs and a schema description and analyzes them via 2AD for potential anomalies. Given the traces and a database schema, the analysis tool outputs a list of tables, columns, and API calls for which 2AD indicates there is a potential anomaly (either level-based or scope-based).

Workflow, False Positives, and Targeted Analysis. 2AD generates a potentially large number of witnesses; for example, if an application fails to use transactions entirely, every read and write to the same column will result in a potential anomaly. Therefore, in our analysis, we took a targeted approach: in addition to outputting all potential anomalies (which can be large), the tool allows filtering by target columns. For a specific invariant, we first identified the relevant columns (e.g., `vouchers.usage` for the voucher invariant). Subsequently, we passed these columns into the 2AD tool, which reported potential witnesses for further inspection. This dramatically reduced the overhead of finding and verifying vulnerabilities. In our traces, the 2AD tool returned a median of 726 vulnerable pairs of anomalous operations per application before filtering. After filtering, the median was 37 witnesses. Via the above schema-driven targeted exploration, by the end of our study, we could perform the trace 2AD analysis in under half an hour per invariant (with most of the time spent identifying table and column names); in contrast, triggering and verifying each attack (e.g., crafting concurrent HTTP requests) took approximately two hours.

Running time. Table 4 (Appendix C) provides a summary of the size of the graphs and the corresponding runtimes. The tool completed in under ten seconds for all traces.

Tool Limitations. Our prototype currently does not support several SQL language constructs such as nested queries, views, and user-

---
[5]https://github.com/stanford-futuredata/acidrain

defined functions. However, the prototype was able to find all of the vulnerabilities described below. Thus, while providing the tool a richer understanding of SQL would improve its ability to accurately find anomalies, this basic implementation proved to be powerful.

4.2.4 Experimental Methodology

We configured each application to run on an Intel i5-430M processor with 4GB RAM running Ubuntu 14.04. Due to application compatibility, we deployed the two Java applications on MySQL Server v5.5.53 and the rest on MariaDB v10.1.10. We subsequently generated database traces by interacting with each site via the public HTTP interface (e.g., placing items in a cart, completing checkout).

Recall that our target invariants are independent of the 2AD analysis; 2AD only finds anomalies, and an 2AD user must relate those anomalies to invariants. Therefore, to detect vulnerabilities, we used our prototype 2AD analysis tool to highlight potential anomalies relevant to the corresponding database tables under MySQL's default isolation level.[6] We subsequently verified each by attempting an attack on the vulnerability by concurrently executing vulnerable API calls via the user interface on our test deployments. When attacks succeeded, we further ensured that each behavior was indeed unexpected by verifying the attack was not possible under a serial execution. To avoid configuring a custom HTTP request generator for each application, we reproduced all the anomalies manually, via rapid, successive HTTP requests (sometimes in separate browsers). For eight (of 22) successful attacks, we introduced additional network delay of 200ms between the application server and database using a pass-through proxy. We have provided instructions for reproducing each vulnerability in the form of publicly accessible bug reports issued against each application (Section 4.2.7).

4.2.5 Analysis Results

Across the 12 applications, we identified 22 vulnerabilities to ACIDRain attacks (Table 5, Appendix C). We discuss developer responses to these vulnerabilities in Section 4.2.7.

Which vulnerabilities occurred? We identified nine inventory vulnerabilities, eight voucher vulnerabilities, and five cart vulner-

---
[6]Because MySQL purports to provide Repeatable Read isolation, MySQL should not allow Lost Updates. However, we were surprised to trigger Lost Updates under MySQL Repeatable Read anyway; that is, MySQL "Repeatable Read" does not provide *PL-2.99*. MySQL uses lock-free multi-versioned reads for all updates except those that specifically specify otherwise (e.g., via `FOR UPDATE`). Thus, MySQL behaves as Read Committed instead. For a detailed discussion of this phenomenon, see https://github.com/ept/hermitage.

abilities. This prevalence inversely correlates with severity: the inventory vulnerability can simply corrupt store inventory—an annoyance, but not necessarily a loss of revenue. The voucher vulnerability allows users to double-spend store credit, but typically receiving the store credit requires the user to purchase the credit at least once. The cart vulnerability is perhaps most severe, allowing potentially unlimited addition of items to a user's order—for free.

Two of the cart vulnerabilities deserve special mention. For both Broadleaf and Shopizer, our tool reported a potential vulnerability that we verified. However, further inspection revealed that the values being written for the order total actually came from request headers, thus making the vulnerability across API scope and thus technically out of scope of our study. However, since we successfully triggered these vulnerabilities and the prototype reported them due to other reads in the checkout API call, we include them here.

Were particular applications more likely to contain vulnerabilities? Only one application (Lightning Fast Shop) contained all three vulnerabilities, and only one application (Spree) contained *no* vulnerabilities (we discuss Spree's application programming patterns that defend against attacks below). In contrast, six applications contained the voucher and inventory vulnerability, and four contained the inventory and cart bug. Shopizer was the only application with just one vulnerability. Thus, with the exception of Spree, these vulnerabilities are widespread in our sample and are not localized to a given set of applications. However, the exact manifestation of each vulnerability reflects the project's coding style and idioms (e.g., [dis]use of transactions; see below).

Database	Level-Based Anomalies Allowed		Remaining
	Default Isolation	Maximum Isolation	
MySQL	5 (RC)	0 (S)	17
Oracle	5 (RC)	1 (SI)	17
Postgres	5 (RC)	0 (S)	17
SAP HANA	5 (RC)	1 (SI)	17

Table 2: RC = Read Committed, SI = Snapshot Isolation, S = Serializability. Summary of how many anomalies would still be observable under the default and maximum isolation levels of some popular databases. As described in Section 4.2.4, MySQL purports to provide Repeatable Read isolation by default but actually provides Read Committed.

What types of anomalies caused vulnerabilities? Of the 22 vulnerabilities, five were level-based, meaning that the default weak isolation level led to the anomalies behind the vulnerabilities. The remaining 17 were scope-based, meaning that the database accesses were not properly encapsulated in transactions and concurrent API requests could trigger the vulnerability independent of the level of isolation provided by the database backend.

Potential level-based anomalies depend on the isolation level permitted by the database and the access pattern. The five that arose from level-based anomalies resulted from both Lost Update (4) behavior and Phantom Reads (1). Thus, under Read Committed (Adya *PL-2*), all five are possible, while only the Phantom Read anomaly should be possible under Repeatable Read (Adya *PL-2.99*) and Snapshot Isolation (Adya *PL-SI*). Table 2 provides an overview of which popular databases expose applications to attacks.

The remaining 17 vulnerabilities were due to scope-based anomalies. In line with [20], several applications failed to use transactions entirely. Some, like Ror_ecommerce (which had both a scope-based vulnerability and a level-based vulnerability) used them sparingly. Two applications had no logged transactions, although one of these two had user level concurrency control in the form of PHP session

locks that prevented one of the vulnerabilities (OpenCart, see below). Many transactions appear to be automatically generated by Object Relational Mapping (ORM) calls instead of manually specified by application programmers, making it difficult to distinguish when transaction usage was intentional. In either case, the prevalence of scope-based vulnerabilities even in the presence of transactions indicates that either programmers, ORMs, or both find it difficult to properly use transactions to encapsulate critical operations.

Were there false positives? As described in Section 4.2.3, we utilized the 2AD prototype's schema-targeted interface to focus on anomalies that pertain to critical columns in the database. We encountered four vulnerabilities that 2AD reported that were not actually triggerable, for one of two reasons. The first class of false positives were due to the use of user-level concurrency control (discussed at length in the next section); for this reason, the witnesses produced by 2AD did not trigger the cart vulnerability in OpenCart and Broadleaf. However, surprisingly, Broadleaf was still vulnerable to the cart exploit due to an error in control flow (i.e., reusing a previous session value). The second class of false positives were due to anomalies that were in fact triggerable but were handled by other program logic and thus rendered benign. The cart vulnerability for Magento and Spree as well as the voucher vulnerability for Spree fell into this category: while we were able to trigger read-write anomalies, these applications used extra database accesses to repeatedly read data and verify invariants at the application level, thus preventing the attack. False positives of the former type could be mitigated by more detailed refinements. The latter require additional information about application control flow.

Our focus on targeted 2AD analysis produced a small set of witnesses pertaining to target columns. Out of curiosity, we investigated a handful of witnesses that were unrelated to the columns of interest to our invariants. Some were merely variations on a vulnerability discussed above. Others were more benign: multiple applications allowed a Lost Update to the user's shopping cart before the checkout process completed, resulting in the user observing an inconsistent cart total in an intermediate step but providing no opportunity to receive inventory for free as in the "true" cart vulnerabilities we report above. Several others were not observable by external users due to internal control flow.

4.2.6 Avoiding ACIDRain Attacks

When weren't applications vulnerable? There were a range of reasons why applications were not vulnerable to all attacks. Three applications (Shoppe, Ror_ecommerce, and Shopizer) lacked the concept of a voucher and so were automatically protected from the voucher vulnerability. One application, Saleor, backed its cart via a session variable instead of the database and was therefore out of scope of this study. Broadleaf appears to have inadvertently rendered its *community edition*'s site inventory management functionality inoperable and we instead found an existing bug report for this broken functionality (and thus were unable to confirm the vulnerability). Shopizer required integrating with a shipping service to exercise its inventory management code, so we do not report on it.

We identified several patterns for avoiding these vulnerabilities. Not all of these patterns appear to have been implemented deliberately to avoid anomalies, as evidenced both by the comments from the developers and the fragility with which some of them manage to prevent a vulnerability:

SELECT FOR UPDATE Appending FOR UPDATE to the end of a SELECT query prevents the data read from being modified until the end of the transaction [42]. This can be used to prevent Lost Update

(i.e., simple Read-Modify-Write) anomalies [17]. Only one of the applications, Spree, used this functionality correctly to prevent the inventory vulnerability. Another application, Ror_ecommerce, used it correctly to prevent the inventory vulnerability when inventory is low. However, Ror_ecommerce is still vulnerable to an attacker as it does not guard the stock management when the inventory is above a user-specified threshold. A third application, Magento, attempted to use SELECT FOR UPDATE to lock the database row before writing it. However, since the read used in the inventory check was made outside of the transaction, Magento was still vulnerable.

In 2AD, accounting for SELECT FOR UPDATE corresponds to a witness refinement limiting allowable witnesses. When looking for a cycle between o_1 and o_2 in the same transaction, with U representing the set of rows locked by SELECT FOR UPDATE after o_1 is executed, this refinement prevents the inclusion of any operation in the witness that conflicts with U.

User level concurrency control. A few applications used user-level locking to prevent concurrent execution of a section of code. PHP automatically performs "session locking" on session files preventing concurrent calls in the same session [9]. This prevented the cart vulnerability in OpenCart. Broadleaf attempted to prevent the cart vulnerability by implementing a mutex in the database. However, while the mutex was correct, the checkout functionality was implemented incorrectly, and a version the cart invariant was still vulnerable (Section 4.2.5, false positives).

In 2AD, user-level concurrency control could correspond to a refinement rule in the abstract history that is derived from application logic. We found it simpler to test the anomaly than search for and encode such refinements.

Single read of data. Some vulnerabilities, like the cart vulnerability, stem from an invariant that certain values in the database obey a given relationship (e.g., the sum of the prices of the items in the order equals the the total charged for the order). A natural way to enforce this constraint during checkout is to read the cart *once*, then compute both the order total and order items from that read. This implementation does not allow an anomaly to occur that violates the constraint, as the data items are both computed from a single input. Oscar, PrestaShop, and WooCommerce avoided the cart vulnerability in this manner. In contrast, vulnerable applications calculate the order total and order items from different reads of the cart table.

In 2AD, a single read of the cart table will cause there to be only one read operation mentioning the cart table in the "checkout" API call, and thus there can be no non-trivial abstract cycle for the cart table starting from this call. When the cart table is read more than once, this creates the opportunity for non-trivial abstract cycle between two read operations in the "checkout" API call and a write operation in the "add to cart" API call.[7]

Multiple validations. Spree avoided the voucher vulnerability by validating that the usage is under the limit multiple times: it checked both before and after marking the voucher as used, as well as a third time near the end of checkout. This pattern allowed anomalies between the checks, but no vulnerability as all anomalies resulted in unsuccessful checkouts. Similarly, both Spree and Magento read from the cart table multiple times but prevent the cart vulnerability by recalculating the cart total after each read.

Multiple validations result in the second type of false positive—triggerable anomalies that do not compromise the application.

---

[7]Some applications did not have a separate order table but instead had to write back to the corresponding row in the cart table to mark the order completed. In these cases, a similar cycle existed in the abstract history.

4.2.7 Response and Discussion

Potential fixes. The anomaly type and access patterns in Table 5 dictates the actions that could be taken to prevent each vulnerability. For level-based anomalies, simply increasing the isolation level to an appropriate level (if supported) would prevent the corresponding attack. Furthermore, some of the predicate based reads we observed were expected to return at most one result. Marking the column being filtered as unique would allow serializable behavior at a weaker isolation level. For scope-based anomalies, refactoring to properly group operations within transactions is required. In either case, alternative methods discussed in Section 4.2.6 such as SELECT FOR UPDATE or multiple validations could also be used to prevent attacks.

Developer Response. We have reported 18 vulnerabilities to application developers by opening support tickets on each application's GitHub repository or issue tracker (Appendix B). Four vulnerabilities had existing issues filed by other users (due to data corruption, and not explicitly for security-related concerns). Seven reported vulnerabilities have been confirmed thus far. The developers of Ror_ecommerce have proposed performing extra reads to prevent the cart vulnerability. The developers of Oscar have proposed using SELECT FOR UPDATE to prevent the inventory vulnerability. A user of Magento responded to the inventory vulnerability issue describing a similar issue in production: "We set one product to sale and after that we have quantity of product=-14. We use 18 instances of frontend. [sic]" In contrast, the developer of OpenCart responded to the inventory vulnerability by posting a comment—"use your brain! its [sic] not hard to come up with a solution that does not involve coding!"—then closed both the inventory and voucher vulnerability issues and blocked us from responding. Broadleaf considers the voucher vulnerability a feature. That is, the Broadleaf developers responded to a similar ticket, indicating that they would prefer to allow concurrent voucher usage on the grounds that failed checkouts due to voucher overuse would result in poor user experience. It is unclear whether the developers recognize the threat due to malicious abuse of this functionality.

5. RELATED WORK

This research builds upon a long line of work on transaction processing under weak isolation. Originally introduced in 1976 as part of the System R project [41], isolation levels have a colorful history, that includes several efforts to model them by Berenson et al. [23] in the mid-1990s, Adya in the late 1990s [17], and several others today [19, 28, 30]. To date, isolation guarantees remain poorly understood [21]. In particular, our empirical analysis builds upon several recent studies in the database community on the impact of weak isolation:

Jorwekar et al. [44] provide techniques for detecting anomalies in Snapshot Isolation, using SQL logs to analyze the behavior of two benchmarks and two applications in use at IIT Bombay. Our 2AD analysis is inspired by Jorwekar et al.'s use of SQL logs, and Jorwekar et al.'s refinements for SI are directly applicable as refinement rules in 2AD. The work in this paper expands upon Jorwekar et al.'s study by focusing on *API-based security vulnerabilities* in database-backed web applications. We introduce a model that captures both API calls and transactions (and transactions within API calls), requiring non-trivial extension to existing models of weak isolation (including Adya [17]). This extension yields important results: as we have empirically demonstrated, many vulnerabilities exist only at the API level (in our study, 17 of 22 vulnerabilities). In addition, we apply 2AD to isolation levels beyond SI (including RC, and RR), requiring further work on trace refinement. Perhaps most importantly, we analyze 12 open source eCommerce appli-

cations written in four languages, with a broad install base (over 2M websites) and via transaction traces that are up to $46\times$ larger than the largest reported in this prior work, providing an expanded perspective on transaction usage and anomalies in the wild.

More recently, Fekete et al. empirically measured conflicts under non-serializable transaction isolation by crafting a synthetic workload and measuring the occurrence of anomalies in concrete execution histories. Our focus here is on predictive analysis. Most recently, Bailis et al. [20] study a corpus of Ruby on Rails applications to determine the susceptibility of Rails applications to invariant violations, in the form of violations of assertions regarding database-backed state appearing in the code (i.e., *validations*). Thus, while Bailis et al. study invariants that programmers explicitly specify across a range of applications, we study a specific class of invariants that are implicit in eCommerce applications and that are not captured by Bailis et al.'s study. As a result of our focus on implicit invariants, we developed 2AD to check for potential invariant violations from database traces using Adya's theory of transaction isolation [17]; in contrast, Bailis et al. use the theory of invariant confluence [18] to check invariants directly via static analysis of Ruby code.

There are a range of other studies profiling weakly consistent databases including Amazon's SimpleDB [59] and S3 [24] databases and providing online algorithms for detecting violations of linearizability, and serializability [38, 62], and various bounded staleness models [22, 36, 61]. Our focus here is on detecting and exploiting weak isolation anomalies and analyzing their impact on database-backed applications as deployed on the public Internet.

Several other works study web security. Most techniques designed for the database setting focus on detecting and preventing database manipulation such as SQL injection [47, 57], which are not our focus. Model-based intrusion detection flags anomalous program executions based on a learned or provided programming model [29, 37]. These tools monitor a running application and enforce invariants at various program points, unlike our tool which does not require instrumenting a running program. These tools also differ in that they cannot reason about concurrent executions. In contrast, Yang et al. [60] predict the growing threat of concurrency attacks and analyze some existing attacks; our work builds upon theirs by defining a new class of concurrency attacks along with studying the prevalence of these attacks in real applications.

Data race detection is a popular topic in program analysis. Most dynamic techniques either search for inconsistent locksets [53, 58] or use Lamport's happens-before relation [46] to find two accesses that are unordered with respect to each other [35, 54, 56]. Static techniques based on the lockset algorithm, type systems, or model checking are also used [27, 31, 49, 52]. There are two key differences between this shared memory setting and the database setting:

First, database analyses must consider weak isolation levels as opposed to weak memory models. These differ substantially in nature; weak memory models are traditionally non-transactional, and their semantics are more influenced by the particulars of hardware cache coherence protocol design than database systems, which historically owe their semantics both to relaxations of two-phase locking [41] and convenient implementation in a multi-versioned concurrency control subsystem [23]. Second, transaction activity is performed at a much higher level of semantic granularity than low-level memory accesses, making it difficult to trace race conditions back to application code. Further adapting data race detection techniques to the database setting is a promising area for future work.

6. CONCLUSIONS

For decades, the transaction concept has played a central role in database research and development. Despite this prominence, trans-actional databases today often surface much weaker models than the classic serializable isolation guarantee—and, by default, far weaker models than alternative, "strong but not serializable" models such as Snapshot Isolation. Moreover, the transaction concept requires the programmer's involvement: should an application programmer fail to correctly use transactions by appropriately encapsulating functionality, even serializable transactions will expose programmers to errors. While many errors arising from these practices may be masked by low concurrency during normal operation, they are susceptible to occur during periods of abnormally high concurrency. By triggering these errors via concurrent access in a deliberate attack, a determined adversary could systematically exploit them for gain.

In this work, we defined the problem of ACIDRain attacks and introduced 2AD, a lightweight dynamic analysis tool that uses traces of normal database activity to detect possible anomalous behavior in applications. To enable 2AD, we extended Adya's theory of weak isolation to allow efficient reasoning over the space of all possible concurrent executions of a set of transactions based on a concrete history, via a new concept called an abstract history, which also applies to API calls. We then applied 2AD analysis to twelve popular self-hosted eCommerce applications, finding 22 vulnerabilities spread across all but one application we tested, affecting over 50% of eCommerce sites on the Internet today.

We believe that the magnitude and the prevalence of these vulnerabilities to ACIDRain attacks merits a broader reconsideration of the success of the transaction concept as employed by programmers today, in addition to further pursuit of research in this direction. Based on our early experiences both performing ACIDRain attacks on self-hosted applications as well as engaging with developers, we believe there is considerable work to be done in raising awareness of these attacks—for example, via improved analyses and additional 2AD refinement rules (including analysis of source code to better highlight sources of error)—and in automated methods for defending against these attacks—for example, by synthesizing repairs such as automated isolation level tuning and selective application of SELECT FOR UPDATE mechanisms. Our results here—as well as existing instances of ACIDRain attacks in the wild—suggest there is considerable value at stake.

Acknowledgements

7. REFERENCES

[1] Flexcoin. https://web.archive.org/web/20160408190656/http://www.flexcoin.com/ (2014).
[2] Michael Stonebraker Turing Award, 2014. http://amturing.acm.org/award_winners/stonebraker_1172121.cfm.
[3] Broadleaf Commerce, 2016. https://github.com/BroadleafCommerce/BroadleafCommerce.
[4] builtwith, 2016. https://builtwith.com/.
[5] Lightning Fast Shop, 2016. https://github.com/diefenbach/django-lfs.
[6] Magento2, 2016. https://github.com/magento/magento2.
[7] OpenCart, 2016. https://github.com/opencart/opencart.
[8] Oscar, 2016. https://github.com/django-oscar/django-oscar.
[9] PHP Session Basics, 2016. http://php.net/manual/en/session.examples.basic.php.
[10] PrestaShop, 2016. https://github.com/PrestaShop/PrestaShop.
[11] ROR Ecommerce, 2016. https://github.com/drhenner/ror_ecommerce.
[12] Saleor, 2016. https://github.com/mirumee/saleor.
[13] Shopizer, 2016. https://github.com/shopizer-ecommerce/shopizer.
[14] Shoppe, 2016. https://github.com/tryshoppe/shoppe.
[15] Spree Commerce, 2016. https://github.com/spree/spree.

[16] WooCommerce, 2016. https://github.com/woocommerce/woocommerce.

[17] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[18] P. Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, UC Berkeley, 2015.

[19] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[20] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[21] P. Bailis, J. M. Hellerstein, and M. Stonebraker. *Readings in database systems*. 3 edition, 2015.

[22] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB*, 2012.

[23] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[24] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *MW4SOC*, 2011.

[25] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[26] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (SDD-1). *ACM TODS*, 5(1):18–51, Mar. 1980.

[27] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[28] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[29] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *International Workshop on Recent Advances in Intrusion Detection*, pages 63–86. Springer, 2007.

[30] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: A unified model for consistency and isolation via states. *arXiv preprint arXiv:1609.06670*, 2016.

[31] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[32] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM.

[33] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[34] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976.

[35] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[36] F. Freitas, R. Rodrigues, et al. Characterizing the consistency of online services (practical experience report). In *DSN*, 2016.

[37] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[38] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *PODC*, 2011.

[39] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[40] J. Gray. What next? a dozen information-technology research goals. page 24, June 1999.

[41] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[42] J. Gray and A. Reuter. *Transaction processing*. Kaufmann, 1993.

[43] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM CSUR*, 15(4):287–317, 1983.

[44] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB*, 2007.

[45] H.-T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, 1979.

[46] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[47] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *ESORICS*, 2002.

[48] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE TKDE*, 2004.

[49] M. Naik, A. Aiken, and J. Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[50] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[51] N. POPPER. A hacking of more than $50 million dashes hopes in the world of virtual currency, June 2016. New York Times DealBook: http://nyti.ms/1UdyDfx.

[52] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. *PLDI*, 2004.

[53] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[54] D. Schonberg. *On-the-fly detection of access anomalies*. 1989.

[55] E. Sirer. Nosql meets bitcoin and brings down two exchanges: The story of flexcoin and poloniex. http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/, 2014.

[56] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.

[57] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140. Springer, 2005.

[58] C. Von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[59] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, 2011.

[60] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *HotPar*, 2012.

[61] K. Zellag and B. Kemme. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *ICDE*, 2011.

[62] K. Zellag and B. Kemme. How consistent is your cloud application? In *ACM SoCC*, 2012.

APPENDIX

A. 2AD THEORY

In this section, we more formally define 2AD ideas and provide proofs of key concepts introduced in Section 3. We adopt the formalism of Adya [17] whenever possible.

A transaction is a totally ordered set of *operations*, each of which is a read or a write to a data item. We model predicate- and set-based operations per Adya [17], where set-oriented operations read and write to predicates. The database contains multiple *versions* of each item; each write to an object returns a new version of the data item, and each read from an object returns a version of the data item. We only consider committed transactions.

A *concrete history* consists of a multiset of transactions T, a partial ordering O on the operations within T, and a set of return values R for each operation appearing in T. We denote this $CH(T, O, R)$.

We say two operations *conflict* if they both operate on the same data item and at least one of them is a write.

The *concrete serialization graph CSG* for $\mathcal{C} = CH(T, O, R)$, denoted $CSG(\mathcal{C})$, is a directed multigraph whose nodes are the transactions in T whose edges are $T_i \to T_j, i \neq j$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in $\mathcal{C}$. Each edge is tagged with the operations that conflict and there is one edge per pair of conflicting transactions.

We similarly can construct an *abstract history* on a set of transactions as defined in Section 3.1.2. For brevity, we consider one transaction per API node here; these results extend to multiple transactions per API node. Given a concrete history $\mathcal{C} = CH(T, O, R)$, we define the abstract history of $\mathcal{C}$ as $AH(\mathcal{C}) = AH(T')$, constructing one API node per transaction.

Lemma 1. A concrete history $\mathcal{C}$ is not serializable if and only if there is a cycle in its CSG [17, 25].

Lemma 2. Every cycle in a $CSG(\mathcal{C})$ contains at least two distinct operations from at least one transaction.

Proof. Each cycle specifies n distinct operations. Sort these operations according to the ordering provided by $\mathcal{C}$. Since the graph has no self edges, $n \geq 2$. By the constraints on the partial order [17], o_1 must precede o_2 and similarly o_2 precedes o_n, so $o_1 \neq o_n$. Since

```
659    set autocommit=0
              ⋮
664    SELECT (1) AS 'a' FROM
              'voucher_voucherapplication' WHERE
              'voucher_voucherapplication'.'voucher_id'
              = 6 LIMIT 1
              ⋮
708    INSERT INTO 'voucher_voucherapplication'
              ('voucher_id', 'user_id', 'order_id',
              'date_created') VALUES (6, 4, 23,
              '2016−11−06')
              ⋮
723    commit
```

Figure 6: Sample logs from Oscar "checkout" API call revealing the voucher vulnerability. We can see that all accesses are properly wrapped in a transaction (setting `autocommit=0` begins a transaction). Oscar checks if a single-use voucher is available by seeing if there are any applications of the voucher. This allows a level-based phantom write anomaly to occur under non-serializable isolation levels such as Read Committed, Snapshot Isolation, and Repeatable Read.

```
559    SELECT 'main_table'.*, 'cp_table'.'type_id'
              FROM 'cataloginventory_stock_item' AS
              'main_table' INNER JOIN
              'catalog_product_entity' AS 'cp_table'
              ON main_table.product_id =
              cp_table.entity_id WHERE
              ('main_table'.'product_id' IN('2048'))
              ⋮
680    START TRANSACTION
681    SELECT 'si'.*, 'p'.'type_id' FROM
              'cataloginventory_stock_item' AS 'si'
              INNER JOIN 'catalog_product_entity'
              AS 'p' ON p.entity_id=si.product_id
              WHERE (website_id=0) AND
              (product_id IN(2048)) FOR UPDATE
682    UPDATE 'cataloginventory_stock_item' SET
              'qty' = CASE product_id WHEN 2048
              THEN qty−1 ELSE qty END WHERE
              (product_id IN (2048)) AND (website_id
              = 0)
683    COMMIT
```

Figure 7: Sample logs from Magento "checkout" API call revealing the inventory vulnerability. While the second access is properly encapsulated in a transaction and use `SELECT FOR UPDATE`, the guard against allowing inventory to become negative uses the value from the first read. This allows the opportunity for a scope-based Lost Update anomaly.

this is a cycle, $T_{o_n} = T_{o_1}$. Therefore, T_{o_1} has two distinct operations in the cycle. $\qquad\square$

We define a *non-trivial abstract cycle* in an abstract history AH as a cycle of API nodes where the cycle contains edges induced by two or more distinct operations residing within a single API node. Note that this definition allows repetition of nodes.

We say that a concrete history $\mathcal{C}$ is *non-serializable in o* if $CSG(\mathcal{C})$ has a cycle containing operation o.

Lemma 3. If there exists a concrete history $\mathcal{C}$ that is non-serializable in operation o, then $AH(\mathcal{C})$ contains a non-trivial abstract cycle containing o.

Proof. By assumption, there is a cycle in $CSG(\mathcal{C})$. Note that there is a surjective mapping from $CSG(\mathcal{C})$ nodes to $AH(\mathcal{C})$ nodes. Since $AH(\mathcal{C})$ allows self edges, every edge in $CSG(\mathcal{C})$ corresponds to an edge in $AH(\mathcal{C})$. Lemma 2 implies that there exists an API node with 2 distinct operations in the cycle in $CSG(\mathcal{C})$, namely the node containing the transaction corresponding to the first operation according to the ordering of $\mathcal{C}$. Starting from that operation, follow the corresponding edges in $AH(\mathcal{C})$ to find a non-trivial cycle in $AH(\mathcal{C})$. $\qquad\square$

A history $\mathcal{C}'$ is in the *expansion* of $\mathcal{C}$ if, for every transaction T_i in $\mathcal{C}'$ there is a corresponding transaction T_j such that the operations in T_i and T_j are identical disregarding concrete values (have identical access patterns over columns).

Lemma 4. Given a concrete history $\mathcal{C}$, if $AH(\mathcal{C})$ contains a non-trivial abstract cycle containing o then there exists an expansion $\mathcal{C}'$ of $\mathcal{C}$ such that $\mathcal{C}'$ is non-serializable in o.

Proof. Consider an abstract history $AH(\mathcal{C})$ with a non-trivial abstract cycle c. $\mathcal{C}$ must contain an API node A_o with two distinct operations o_i and o_j such that o_i and o_j form part of a non-trivial abstract cycle. Consider the following history $\mathcal{C}'$: first, execute all

of the transaction operations within A_o up to and including o_i. Next, follow the cycle c and execute all of the operations of each API node in c in their respective transaction order. If an API node is ever revisited, create a new instance of that API node and its operations. Finally, execute o_j and the remainder of A_o (don't create a fresh API node for o_j). Each transaction in $\mathcal{C}'$ corresponds to a transaction in the set used to create AH, so it can be mapped to a transaction in $\mathcal{C}$ with the same structure. Therefore, it is an expansion of $\mathcal{C}$. Next, consider $CSG(\mathcal{C}')$. Follow the same path of operations as those in the cycle in $AH(\mathcal{C})$ starting from o_i. Because each operation had conflicts in $AH(\mathcal{C})$, their corresponding operations must conflict in $\mathcal{C}'$. There is a cycle of such operations in $\mathcal{C}'$ formed by the counterpart operations, beginning at $\mathcal{C}'$'s counterpart for o_i and ending at o_j. Therefore, by definition, there must be a directed cycle in $CSG(CH(\mathcal{C}'))$ and therefore $\mathcal{C}'$ is a non-serializable expansion. $\qquad\square$

Lemma 5. For each complete concrete history $\mathcal{C}'$ in the expansion of $\mathcal{C}$, $AH(\mathcal{C}') \subseteq AH(\mathcal{C})$.

Proof. Recall $\mathcal{C} = CH(T, O, R)$ and $AH(\mathcal{C}) = AH(T)$. Since each expansion $\mathcal{C}'$ is also a history, it must have also be $\mathcal{C}' = CH(T', O', R')$ and $AH(\mathcal{C}') = AH(T')$. We know that for each $T_j \in T'$, there is a mapping to a $T_i \in T$ such that their structure is the same. As described in Section 3.1.2, an abstract history will collapse API nodes with the same structure. Thus, $AH(\mathcal{C}') \subseteq AH(\mathcal{C})$. $\qquad\square$

Theorem 1 (Formal). For every operation o in a concrete history $\mathcal{C}$, there exists a concrete history $\mathcal{C}'$ in the expansion of $\mathcal{C}$ that is non-serializable in o iff $AH(\mathcal{C})$ contains a non-trivial abstract cycle including o.

Proof. Case non-serializable expansion implies cycle: Call the expansion $\mathcal{C}'$. By Lemma 3, $AH(\mathcal{C}')$ contains a cycle. By Lemma 5, $AH(\mathcal{C}') \subseteq AH(\mathcal{C})$, so the cycle can still be found in $AH(\mathcal{C})$.

```
108   set autocommit=0                            (a)

109   INSERT INTO 'cart_cartitem' ('cart_id',
         'product_id', 'amount',
         'creation_date', 'modification_date')
         VALUES (8, 1, 1, '2016−07−18
         18:35:23.204957', '2016−07−18
         18:35:23.205002')
110   commit
```

```
388   SELECT 'cart_cartitem'.* FROM              (b)
         'cart_cartitem' WHERE
         'cart_cartitem'.'cart_id' = 8 ORDER BY
         'cart_cartitem'.'id' ASC
                        ⋮
402   set autocommit=0
403   INSERT INTO 'order_order' (. . .) VALUES
         (. . .)
404   commit
                        ⋮
438   SELECT 'cart_cartitem'.* FROM
         'cart_cartitem' WHERE
         'cart_cartitem'.'cart_id' = 8 ORDER BY
         'cart_cartitem'.'id' ASC
439   set autocommit=0
440   INSERT INTO 'order_orderitem' (. . .)
         VALUES (8, 100, 100, 0, 1, 1, '1', 'tp1',
         100, 100, 0)
441   commit
```

Figure 8: Sample logs from the Lightning Fast Shop "add to cart" (a) and "checkout" (b) API calls revealing the cart vulnerability. The reads from the cart individually listed all fields and joined to the product table for pricing information, we have simplified for space and clarity. We have similarly simplified the INSERT statements. The order total is calculated from a different read than the one used to specify the order items. This gives an opportunity for a new item to be inserted into the order in between calculating the total and recording the items. The automatically generated transactions that wrap only single operations do not help prevent this anomaly.

Case cycle implies non-serializable expansion: Follows directly from Lemma 4. □

Finally, note that any non-trivial abstract cycle c implies the existence of a non-trivial abstract cycle with at most one node repetition. By assumption, c must contain some o_i and o_j in the same API node A_0. There must be a subpath of c from o_i to o_j that does not visit o_i or o_j except at the endpoints. Starting from this subpath, we can build a new cycle C' by further collapsing any cycles along this path that do not contain the first or last edge of the path. The resulting C' is still a cycle containing two distinct operations o_i and o_j in the same API node, so it is a non-trivial abstract cycle. However, the only API node that could be repeated is A_0 as the other endpoint of one of the edges containing o_i or o_j.

B.   2AD LINKS TO ISSUES

Name	Variables	Invariant
Cart	item i (cost: c_i, qty: q_i), total: T	$\sum_i c_i q_i = T$
Inventory	item i, stock s_i	$\forall i, s_i \geq 0$
Voucher	cost in usage i: c_i, limit: v_{limit}	$\sum_i v_i \leq v_{limit}$

Table 3: Formal statements of target eCommerce invariants; for brevity, we omit the "no Lost Updates" component of the Inventory invariant.

We provide links to each GitHub issue we opened during our investigation below:

https://github.com/opencart/opencart/issues/4811
https://github.com/opencart/opencart/issues/4812
http://forge.prestashop.com/browse/PSCSX-8333
http://forge.prestashop.com/browse/PSCSX-8334
https://github.com/magento/magento2/issues/6363
https://github.com/magento/magento2/issues/6364
https://github.com/woocommerce/woocommerce/issues/12467
https://github.com/tryshoppe/shoppe/issues/403
https://github.com/drhenner/ror_ecommerce/issues/174
https://github.com/django-oscar/django-oscar/issues/2101
https://github.com/django-oscar/django-oscar/issues/2102
https://github.com/mirumee/saleor/issues/543
https://github.com/mirumee/saleor/issues/544
https://github.com/diefenbach/django-lfs/issues/201
https://github.com/diefenbach/django-lfs/issues/202
https://github.com/diefenbach/django-lfs/issues/203
https://github.com/BroadleafCommerce/BroadleafCommerce/issues/1574
https://github.com/shopizer-ecommerce/shopizer/issues/121

C.   2AD APPLICATION VULNERABILITY EXAMPLES

Figures 6, 7, and 8 show real logs we used to detect the voucher, inventory, and cart vulnerabilities respectively. We have highlighted only the relevant log statements, and explain how they exemplify the patterns discussed in the main text.

Table 3 provides sample logical predicates for invariants. Table 4 provides statistics on graph sizes and runtimes. Table 5 summarizes the types of vulnerabilities we found, along with the access patterns and transaction usage allowed for the corresponding anomalies to take place.

Figure 9 shows a sample abstract history corresponding to a simplified eCommerce application. This abstract history contains API calls for an add_to_cart function and a checkout function. This application is backed by a cart_items table storing the products in a user's cart, a stock table storing product stock values, an orders table storing order total information, and a order_items table storing the products bought in each order.

This figure contains two cycles corresponding to invariant violating anomalies. First, there is a scope-based anomaly represented by the path between operations 5, 3, and 7 creating a cycle between the two API nodes. This anomaly could cause a cart vulnerability. Second, there is the path from operation 4 to 9 creating a self-loop cycle on the checkout API call. The corresponding scope-based anomaly could cause an inventory vulnerability. While this graph is quite simplified, it captures the essence of behavior that we saw in real applications.

App Name	Operation Nodes	Txn Nodes	Explicit Txns	API Nodes	Edges	Total Runtime (s)	Parse (s)	Analyze (s)
OpenCart	1575	1575	0	12	7845	9.761	9.458	0.299
PrestaShop	1349	1349	0	9	1745	9.165	8.867	0.270
Magento	653	574	13	7	956	4.117	4.035	0.785
WooCommerce	884	740	1	7	8522	3.232	2.976	0.239
Spree	689	587	22	6	3503	2.565	2.395	0.168
Ror_ecommerce	190	159	3	6	226	0.491	0.483	0.007
Shoppe	126	102	6	6	136	0.335	0.321	0.004
Oscar	469	154	14	8	373	2.465	2.424	0.038
Saleor	226	83	16	9	191	1.035	1.026	0.007
Lightning Fast Shop	350	347	1	6	460	2.320	2.288	0.030
Broadleaf	253	216	11	6	288	5.878	5.860	0.017
Shopizer	183	125	37	5	134	7.366	7.348	0.016

Table 4: Explicit transactions are those with more than one operation that have explicit BEGIN and COMMIT statements. All runtimes only measure the time to find the set of vulnerable API call/table pairs, and are run on an Intel i5-430M processor with 4GB RAM. The runtimes to find an anomaly for all pairs that operate on a specific table are not shown, but were also all under ten seconds.

		Vulnerability								
		Voucher			Inventory			Cart		
Language	Application	V	AP	AT	V	AP	AT	V	AP	AT
PHP	Opencart	yes	phantom	scope	yes	LU	scope	no		
	PrestaShop	yes	LU	scope	yes	LU	scope	no		
	Magento	yes	LU	scope	yes	LU	scope	no		
	WooCommerce	yes	LU	scope	yes	LU	scope	no		
Ruby (Rails)	Spree	no			no			no		
	Ror_ecommerce	NF			yes	LU	level	yes	phantom	scope
	Shoppe	NF			yes	phantom	scope	yes	phantom	scope
Python (Django)	Oscar	yes	phantom	level	yes	LU	level	no		
	Lightning Fast Shop	yes	LU	scope	yes	LU	scope	yes	phantom	scope
	Saleor	yes	LU	level	yes	LU	level	NDB		
Java (Spring)	Broadleaf	yes	phantom	scope	BF			yes*	phantom	scope
	Shopizer	NF			BF			yes*	phantom	scope

Table 5: Summary of vulnerabilities. V = Vulnerable, AP = Access Pattern, AT = Anomaly Type, NF = No Functionality, BF = Broken Functionality, NDB = Functionality that is not database backed, and thus out of the scope of this study. The two yes* correspond to triggerable bugs that were reported by the tool (See Section 4.2.5, "Were there false positives?").
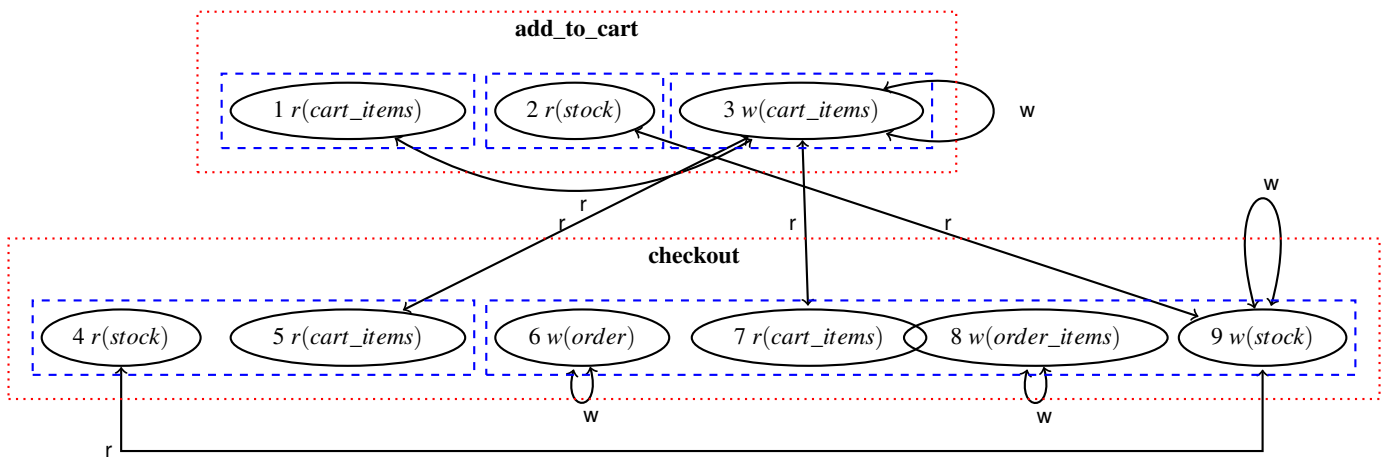

Figure 9: An abstract history corresponding to a sample `add_to_cart` API call and sample `checkout` API call. Solid circles correspond to operations, dashed rectangles to transactions, and dotted rectangles to API calls. Edges are labeled with the type of conflict.