RAPID: Checking API Usage for the Cloud in the Cloud

Michael Emmi
Amazon Web Services
USA

Liana Hadarean
Amazon Web Services
USA

Ranjit Jhala
Amazon Web Services
USA

Lee Pike
Amazon Web Services
USA

Nicolás Rosner
Amazon Web Services
USA

Martin Schäf
Amazon Web Services
USA

Aritra Sengupta
Amazon Web Services
USA

Willem Visser
Amazon Web Services
USA

ABSTRACT

We present Rapid, an industrial-strength analysis developed at AWS that aims to help developers by providing automatic, fast and actionable feedback about correct usage of cloud-service APIs. Rapid's design is based on the insight that cloud service APIs are structured around *short-lived* request- and response-objects whose usage patterns can be specified as value-dependent type-state automata and be verified by combining *local* type-state with *global* value-flow analyses. We describe various challenges that arose to deploy Rapid at scale. Finally, we present an evaluation that validates our design choices, deployment heuristics, and shows that Rapid is able to quickly and precisely report a wide variety of useful API misuse violations in large, industrial-strength code bases.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; • **Theory of computation** → **Program analysis**; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

software security, API usage checking, static analysis in the cloud

ACM Reference Format:
Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. RAPID: Checking API Usage for the Cloud in the Cloud. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473934

1 INTRODUCTION

Modern software applications make heavy use of *services* hosted in the cloud, *e.g.* to securely and efficiently handle sensitive data. Cloud providers expose these services to clients via APIs — *e.g.* for lookup tables, encryption, storage, queueing, and event-driven execution — that let developers compose the services to engineer robust client applications. For example, millions of Amazon Web Services (AWS) customers use Java APIs for Amazon SQS to implement queues, Amazon S3 to implement persistent storage, AWS KMS for crypto key management, Amazon DynamoDB for lookup tables, and AWS Lambda for executing functions in a virtualized environment.

Cloud service APIs accommodate the widely varying *requirements* of different clients by exposing a variety of methods and parameters that developers can use to customize how their application interacts with the services. However, for ideal results, these methods and parameters must be used according to particular *patterns* that correspond to the best practices for using the underlying services. These practices are at best articulated informally, *e.g.* in the API documentation, and hence, easy to get wrong. For example, incorrect use of the APIs that access the results of storage lookup queries can lead to severe *performance* bottlenecks, such as large datasets being delivered all at once, instead of more efficiently in a "paginated" manner. Similarly, improper use of crypto or data sanitization APIs can lead to serious *security* vulnerabilities in client code, such as untrusted data ending up directly on client pages, leading to potential XSS vulnerabilities [5].

Goals We present Rapid, an industrial-strength analysis developed at AWS that aims to help application developers by providing feedback about correct usage of cloud-service Java APIs. We designed Rapid to meet three constraints that must be satisfied for practical deployment. First, the feedback must be *automatic*, without requiring developers to configure which checks should run or provide annotations, as we want the checks to run on all client applications, regardless of the developers' familiarity with such analysis tools. Consequently, we require that Rapid be *expressive* enough to let analysis designers easily specify a wide variety of API usage patterns, so that the checker can be robustly applied to review a multitude of client applications using different service APIs. Second, the feedback must be *fast*, in the order of low tens of minutes, so that it can be part of continuous integration (CI) or used during code review, where developers can be quickly alerted to the best practices, and fix their code before it is merged in and deployed. In practice, this goal requires Rapid to *scale* up to quickly analyzing hundreds of different usage patterns on large code bases. Third, developers will quickly stop using the analysis if it is not immediately clear from an error message how to

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473934

fix or suppress an issue. Successful deployment requires that the feedback be *actionable*, without requiring the developer to wade through many false alarms, and giving them constructive advice on how to modify their code to adhere to best practices for the pertinent APIs.

Design We designed Rapid to address these challenges via two key insights. Our first insight is that service APIs are structured around *short-lived* request- and response- objects that are used to communicate with a *long-lived* cloud services as exemplified by the standard patterns for OpenAPI or Swagger APIs [12]. Thus, most interesting and important API properties pertain to operations on the short-lived objects before they get sent to the client or the short-lived responses that are returned by the service. The common practice is to *not* persist responses from such API services in fields, but rather, store them in local variables in a method. We demonstrate how the short-lived patterns for these APIs can be expressed as type-state rules which make precise and scalable analysis feasible by *focusing* the analysis to only locally track object-sharing, instead of a global, field-sensitive, whole-program analysis that would preclude our low-latency requirement. However, for many APIs, correct usage depends not just on the sequences of methods as considered by classic type-state analyses, but also on the *values* of particular parameters used, *e.g.* to configure the service request objects. Our second insight is that in practice, these parameters are idiomatically set, not by some complex computations over strings which would require a complex value-flow analysis, but by choosing one of a small set of literals. Thus, we combine these two insights by composing a *local*, field-insensitive, type-state analysis that tracks the allowed method sequences, with a *global* on-demand, field-sensitive inter-procedural reaching-definition analysis that refines the type-states using parameter values, yielding an automatic analysis that provides fast and actionable feedback.

Deployment Challenges We found that in addition to carefully architecting the structure of the analysis, we had to address two unexpected engineering challenges to deploy Rapid at scale. First, Rapid provides two usage modes, *repository analysis*—scans all the code in a repository branch, and *code-review*—scans all code relevant to a pull-request. The user can trigger an analysis by push or pull-request events via simple configuration files. Since the properties checked by Rapid require inter-procedural reasoning, in practice, each run has to analyze a whole repository spanning tens of packages, hundreds of jars or thousands of classes. Rapid addresses this challenge via a simple strategy that *partitions* the classes into clusters that can be analyzed in an "embarrassingly" parallel manner. Our evaluation also suggests that, for large inputs, a simple clustering strategy is surprisingly effective and has little impact on the precision of the analysis. Second, we do not wish to burden users with having to configure or specify the particular sets of patterns that should be checked. Hence, the analysis does not know *a priori* which patterns are relevant to a given code base, and so it must be able to efficiently check potentially hundreds of specification rules at each analysis. Rapid addresses this challenge via a novel *specification fingerprinting* scheme can quickly eliminate lines of code that are irrelevant to the properties being checked.

Evaluation We have implemented and deployed Rapid at AWS and externally as a cloud-based service called CodeGuru [1, 2]. We present an empirical evaluation on open-source software comprising a representative subset of 100 packages from Maven Central [11], selected after analyzing properties of 26000 packages, that assesses our design choices. First, we demonstrate that Rapid is expressive by using it to specify 90 API usage patterns spanning 98 Java classes and APIs. Next, we demonstrate that Rapid provides actionable feedback by precisely pinpointing API usage anti-patterns without overwhelming users with false-alarms. An Amazon wide internal deployment of Rapid on code review pipelines shows a 76% user acceptance on recommendations, validating our design choices. Finally, we show that clustering and fingerprinting enable Rapid to scale up to provide fast feedback for large code-bases with a median analysis latency of 239 seconds, meeting the stringent low-latency requirements for code review.

```
void getBatchItems(DynamoDB client,
                   Map<String, KeysAttribs> items) {
  Request req = new BatchGetItemRequest(items);
  Result result = client.batchGetItem(req);
  if (!result.getResponses().isEmpty())
    println(result.getResponses());
  if (!result.getUnprocessedKeys().isEmpty())
    // Some retry logic.
}
```

Figure 1: Abbreviated example of how to perform a batch request with DynamoDB.

2 MOTIVATION

We start with an overview that illustrates the requirements of checking API usage and showing how to efficiently decompose the analysis into *local* type-state checking and *global* reaching definitions.

2.1 Local Type-State Checking

The first example in Figure 1 shows how to send a batch of requests to the NoSQL database DynamoDB. The documentation states that not all requests in a batch may complete in the first request depending on the size of the individual responses, and that it is up to the caller to check and retry the unprocessed keys. The API usage checker should detect if the result is retrieved but the unprocessed keys are not checked.

Request API Requirement We can ensure that the caller actually reads the unprocessed keys by specifying the requirement as a *type-state* property on the `Result` object. First, we can specify that, when an object of type `Result` is returned by the `batchGetItem` method, it is in an *unchecked* type-state. Next, when `getUnprocessedKeys` is called on an *unchecked* object, the receiver moves into a *checked* state. Finally, the object transitions to an *error* state, if it reaches its *end-of-life*, while remaining in the *unchecked* state. We use this simple property for brevity: in practice, the property may have to be a bit more complex to avoid false alarms on exceptional paths. Unfortunately, generic type-state analyses capable of checking the above property can be hard to scale to real-world code bases. In particular, knowing when an object reaches end-of-life is hard for the general case, as it requires expensive sub-analyses, *e.g.* for precise aliasing detection and call-graph construction.

RESTful APIs Rapid is based on the key insight that our use-case focuses on checking uses of RESTful APIs, which have *short-lived* response objects, a pattern that is not unique to DynamoDB or AWS,

```
private final String p1 =
  "http://.../external-general-entities";
private static final String p2 =
  "http://.../external-parameter-entities";
public Element createParserSecure(String payload)
throws Exception {
    DocumentBuilderFactory
     docBuilder = new DocumentBuilderFactory();
    docBuilder.setFeature(p1, false);
    docBuilder.setFeature(p2, false);
    // Parse raw data into it's components
    payload = payload.replaceAll(...);
    val builder = docBuilder.newDocumentBuilder();
    val is = new InputSource(new StringReader(payload));
    val doc = builder.parse(is);
    return doc.getRootElement().getChild("name");
}
```

Figure 2: Abbreviated example of how to configure and parse an input payload using a DocumentBuilder.

but instead is ubiquitous in the APIs of many cloud services. In particular, the documentation for DynamoDB states that API *clients* (*e.g.* `DynamoDB client`) should be reused, *i.e.* should be members of a class, but that the *responses* returned by a `client` (*e.g.* `Result result`) are short-lived and *should not* be persisted in fields, but should, instead, be local variables in a method.

Localizing Typestate Analysis The RESTful short-lived requirement greatly simplifies property checking. First, we can restrict the alias analysis to track sharing in an interprocedural but field-insensitive manner. Second, we can simply consider a type-state variable to have reached its end-of-life at the point where its *scope* ends. We need not trust that developers adhere to the best-practices for short-lived responses. Instead, we add a *malpractice* state to the type-state property and stipulate that the response object enters this state if it is assigned to a field or added to a collection that is part of another object. Being in the *malpractice* state means our analysis is not equipped to further reason about the object, and we do not issue any warnings for such objects. However, we show that the *malpractice* state occurs so infrequently that we could even issue a warning when it occurs, to alert the developer to a possible improvement in their code.

2.2 Global Reaching Definitions

A purely local type-state analysis is insufficient for APIs where the type transitions depend on the *values* of API method arguments.

Builder API Requirement Consider the code in Figure 2 from the OWASP cheatsheet series [15] on how to build XML readers resilient to XXE attacks. We wish to ensure that the `DocumentBuilderFactory` object that is used to construct the `builder` *disables* the features `external-general-entities` and `external-parameter-entities` before being used to parse a string. We can specify this requirement as a type-state property comprising four states, which represent whether *none*, *first*, *second* or *both* of `external-general-entities` and `external-parameter-entities` are disabled. Every newly constructed `DocumentBuilderFactory` object starts off in the *none* state. Dually, the object transitions into *error* if the `parse` method is called on `builder`, derived from `docBuilder`, in any state other than *both*. Invocations of the `setFeature` method trigger transitions into the *first*, *second* and *both* states. However, to correctly track which properties have been disabled, we need the *values* of the call parameters.

Global Reaching Definitions We find that for many API properties these values can be obtained from a separate *backward reaching-definitions* analysis which determines which literals may reach a given variable. This is significantly more efficient than a full-blown value-flow analysis which determines the possible values of this variable. Computing reaching-definitions is considerably cheaper because we do not need to model the semantics of operations (*e.g.*, string concatenation), and we do not need to compute a fixed point. A reaching-definitions analysis lets us quickly determine the values of `p1` and `p2` and hence that the corresponding `setFeature(p1, false)` and `setFeature(p2, false)` calls transition the `docBuilder` object first from `none` to `first` and then to the `both` state thereby validating the invocation of `parse`. § 3.4 describes how different type-states, such as `DocumentBuilderFactory` and `DocumentBuilder`, are composed. As shown even in this simple example, we require the reaching definitions analysis to be global, (*i.e.* inter-procedural), and field-sensitive (*i.e.* separating the literals that reach different fields like `p1` and `p2`).

Thus, many interesting API usage properties can be checked by combining an inexpensive local type-state analysis with an on-demand, field-sensitive, inter-procedural reaching-definitions analysis. Next, we detail how we specify properties for our analysis, how we implement the analyses, and show the result is scalable and precise on real-world code bases.

3 PROPERTY SPECIFICATION LANGUAGE

Rapid allows users to specify a wide variety of API properties using an expressive *guarded typestate automaton* language. This language has two main components: a *typestate automata* language to express temporal type-state properties and a *predicate* language that guards the automata transitions. The two components correspond to the two key elements of our analysis: the local typestate analysis tracks the automaton state, while the global reaching definitions are used, on-demand, to evaluate the guard predicates. Next, we describe Rapid's property specification language, by presenting the typestate automaton and predicate languages. We start by describing the automata in § 3.1, and then detail the predicate language in § 3.2. We then describe two important features that greatly simplify writing modular, readable and extensible specifications: extending typestates with a key-value store § 3.3, and a mechanism to compose automata § 3.4.

3.1 Typestate Automata

Rapid's typestate automaton language is parameterized by a language Φ of predicates that guard automata transitions.

Property Automata A property is described by an automaton $P \doteq (S, s_0, E, T, \phi_0)$ comprising five elements: (1) S is a finite set of *type-states*; (2) $s_0 \in S$ is the *initial* state of the automaton; (3) $E \subseteq S$ is the set of *error* states whose reachability results in an issue reported to the user; (4) $T \subseteq S \times \Phi \times S$ is the set of *guarded transitions*, where each transition t is a triple $t \doteq (s, \phi, s')$ comprising *source* and *target* states s, s' and a *predicate* $\phi \in \Phi$ that must be satisfied before the automaton transitions from s to s'; and (5) $\phi_0 \in \Phi$, an *instantiation* predicate that specifies the program expressions at which new instances of the type-state automaton P should be created.

Instantiated Automaton An instance of P is created at each program expression e at which the instantiation predicate ϕ_0 holds. An *instantiated automaton* is a pair $P_e \doteq (P,e)$ comprising the automaton P and the expression e where the instance was created.

3.2 Guard Predicates

The predicate language Φ comprises boolean combinations (*i.e.* conjunctions, disjunctions, negations) over a fixed set of *atomic* predicates over program expressions. Each predicate is either a *syntactic* check, *e.g.* that the expression corresponds to a call of a method with given name or type, or a *semantic* condition, *e.g.* that the run-time values passed as parameters in the call match some criteria. In essence the predicate language allows Rapid to *factor out* "global" concerns like value-flow from "local" ones which related to the temporal order of API invocations. We discuss in § 4 how the predicates themselves are implemented via a plugin architecture that allows each predicate to invoke its own (global) sub-analysis on-demand. Next, lets see how Rapid supports a variety of API specifications by populating Φ with a diverse and extensible set of 52 atomic predicates.

Calls Predicates CallsMethod(e, `sig`) evaluate to true for expression e that contains a call to a method that matches the signature `sig`. The instantiation condition ϕ_0 for Figure 1 can be specified as:

$$\phi_0 \doteq \mathsf{CallsMethod}(e, \texttt{DDBClient.batchGetItem}) \vee \mathsf{CallsMethod}(e, \texttt{DDBClient.batchWriteItem})$$

which evaluates to true for any expression e in which `batchGetItem` or `batchWriteItem` is called on any `DDBClient` object. A new instance of the automaton P is associated with the receiver of the method call, for each expression at which ϕ_0 evaluates to true. Thus, in the example from Figure 1, an instance of the property automaton is associated with the receiver variable `result`.

Calls-Instance Predicates CallsInstMethod(P_e, e', `sig`) evaluate to true for any expression e' and instantiated automaton P_e if (1) CallsMethod(e', `sig`), *i.e.* e' contains a call to a method whose signature matches `sig`, and (2) the base of the call is an *alias* of the receiver in e *i.e.* is received by the object associated with the instance automaton P_e. We can use this predicate to specify the transition t_1 from the unchecked to the checked type-state described for the example from Figure 1 as $t_1 \doteq (\text{unchecked}, \phi_1, \text{checked})$ where

$$\phi_1 \doteq \mathsf{CallsInstMethod}(P_e, e', \texttt{getUnprocessedKeys})$$

Note that evaluating this predicate requires us to decide if there is an alias between the receiver in expression e and the base of the method call in e'. It is up to the implementer of the predicate to decide how precise this alias analysis should be.

Is-Garbage Predicates IsGarbage(P_e) evaluate to true at any program point where (objects aliased to) the expression e are no longer *live*, *i.e.* will no longer be *used* in the future. This predicate helps specify properties that require that some operation is performed *at least once*. For example, in the example from Figure 1, we specify that the API client must check the results by stipulating the following transition t_2 from the unchecked to the error typestate:

$$t_2 \doteq (\text{unchecked}, \phi_2, \text{error}) \qquad \phi_2 \doteq \mathsf{IsGarbage}(P_e)$$

This transition states that our automaton goes into an error state if the unprocessed keys have not been checked yet (via t_1 described above) but the object associated with e is no longer live.

To implement the IsGarbage($\cdot$) predicate, we need an analysis that can decide when an object is used for the last time. For method-local variables we can find this point using standard compiler optimization techniques. However, for the general case, the problem would require a full-program alias analysis. In § 4 we describe why a local analysis suffices with minimal loss of precision.

Parameter-Value Predicates ParamValue(P_e, e', `sig`, `idx`, `rgx`) evaluate to true if CallsInstMethod(P_e, e', `sig`) is true and additionally, *any* of the possible values of the argument at position `idx` of the method call match the regular expression `rgx`. This predicate lets us write the rule for the example in Figure 2 that transitions from initial to firstSet where `setFeature` sets the `external-general-entities` parameter of the `DocumentBuilderFactory` object:

$$t_2 \doteq (\text{initial}, \phi_2, \text{firstSet}) \quad \texttt{rgx}_2 \doteq \texttt{.*external\_general\_entities}$$

$$\phi_2 \doteq \mathsf{ParamValue}(P_e, e', \texttt{setFeature}, 0, \texttt{rgx}_2) \wedge \mathsf{ParamValue}(P_e, e', \texttt{setFeature}, 1, \texttt{false})$$

Informally, the guard ϕ_2 says the transition is triggered at any expression e' that is an invocation of `setFeature` where the 0^{th} configuration parameter matches $\texttt{rgx}_2$ and 1^{st} parameter is `false`.

Evaluating Predicates All the predicates are evaluated in the same way: the analysis has access to the current statement or expression and to the instantiated automaton P_e (or just the automaton in the case of ϕ_0 predicates). The implementer of the predicate decides what technique to use when evaluating the predicate. The key challenge that Rapid solves when implementing the parameter-value predicate is to identify the *possible values* that flow into the method argument, in a way that balances precision and performance (§ 4).

3.3 Stores and Updates

In practice, many properties require checking that a *set of* actions have occurred prior to some event. It can get quite cumbersome to directly encode such properties with typestates as the number of required states grows exponentially with the number of actions as we need a separate typestate to represent each *subset* of actions that have already occurred. For example, recall the example from § 2.2 that illustrates how to build XXE-attack resistant XML readers by disabling *two* features in the `docBuilder`, which required four different typestates (*none*, *first*, *second* and *both*). This example is a simplification: the actual property requires correctly setting *five* parameters [15]. The resulting automaton would have thirty two typestates which is very tedious to specify, audit, and understand.

States We address this problem by *factoring* the typestate specifications into an *explicit* control state and an *implicit* key-value store. Let K and V respectively denote sets of *keys* and *values*. Let M be the set of finite maps from K to sets of values V. We can then factor the set of states S into pairs of control states $C \doteq C \times M$ where maps $M \doteq K \mapsto 2^V$. We write $m(k)$ to denote the set of values bound to k in m, which is empty when the key is not defined in m.

Updates A transition can change the map via an update u which comprises a (possibly empty) *sequence* of *store* operations $store(k,v)$ each of which insert a new binding from k to v in the implicit store. Let $m[k \mapsto V]$ denote the map that binds each k' to $m(k')$ if $k' \neq k$ and to V otherwise. We define $u(m)$ to denote the new store resulting from applying the updates u to m as $\emptyset(m) \doteq m$ and $(store(k,v);u)(m) \doteq u(m[k \mapsto \{v\} \cup m(k)])$.

Transitions We refactor the set of transitions as $T \subseteq C \times \Phi \times U \times C$ where each transition t is of the form (c, ϕ, u, c') which compactly describes the *set of* transitions from states (c, m) to $(c', u(m))$. Further, we extend the language of predicates with primitives that guard transitions using the current store, *e.g.* by checking for the presence of particular keys, or the total number of store bindings.

Example The left pane in Figure 3 shows how key-value stores can be used to compactly specify the XXE construction property. The specification contains three control states `initial`, `allSet` and `notEnough`. The first transition – from `initial` to `allSet` – is triggered at when the `build` method is invoked, *only if* we have already correctly configured the values of two or more properties (regardless of their order). The second transition – from `initial` to `notEnough` – is instead triggered at the `build` invocation, when fewer than two properties have been properly configured. The third transition – from `initial` back to itself – is triggered when `setFeature` is invoked to configure the value of any of the two important parameters, and uses isValueStored(*key*, `sig`, `idx`) predicate to access the key-value store. As before we use `$0` and `$1` to name the *values* that can flow into the corresponding index of parameters at the invocation sites. The *pre-condition* checks that the relevant parameter has not been set, and the *effect* updates the store appropriately. The above mechanism easily scales up to account for all the five different configuration parameters required by the full XXE specification. In essence, we need only change the 2 to 5 in the first two pre-condition clauses, and add the extra three string parameter names to the `isArgumentInSet` predicate.

3.4 Composing Automata

Many real-world specifications are too complicated to express within a single typestate automaton. First, these specifications span method invocations of *multiple* objects. Second, we found it useful to factor out common parts of specifications to enable *reuse* across different end-to-end properties. For example, recall the XXE example from § 2.2. The actual error occurs when the `parse` method is invoked on a `DocumentBuilder` instance obtained from an incorrectly configured `DocumentBuilderFactory` instance. Hence, while the `xxeDocumentBuilder` specification on the left in Figure 3 shows how to track the typestate for each `DocumentBuilderFactory` instance, in real codebases we must connect that information with subsequent invocations of `parse` on a *different* (derived) object.

Dependencies We address this problem by introducing a mechanism for *composing* multiple automata by allowing each automaton's specification to *depend upon* upon other specifications, illustrated on the right in Figure 3. The specification includes a `DependsOn` clause that indicates that the `xxeParserUse` specification "mixes-in" the `xxeDocumentBuilder` specification on the left.

Automaton Predicates We connect the specification of one automaton to others by using special *dependency predicates* that check which states the other automata are in at that particular program point. For example, the predicate isAutomatonInState(*a*, *s*) holds if there is an instance of the *a* automaton in (control) state *s* at that point. So isAutomatonInState(`xxeDocumentBuilder`, `allSet`) in the first transition on the right in Figure 3 ensures that any methods *other than* `parse` transition the automaton to a "safe" state as long as the `DocumentBuilderFactory` from which the object was derived, was properly configured, *i.e.* was in the `allSet` typestate which the specification on the left establishes when the relevant sanitization parameters have been properly configured. The condition on the second transition moves the automaton into the `error` state if the originating `DocumentBuilderFactory` was *not* properly configured.

Stratified Analyses Rapid uses the dependency clauses to *stratify* the specifications into a directed acyclic graph (we prohibit cyclic dependencies). The analysis then processes the specifications in a topologically sorted order, *i.e.* performing the analysis for each automaton *after* generating the results for each of that automaton's dependencies, thereby using memoized results and allowing us to quickly analyze the composed specifications.

4 DESIGN AND IMPLEMENTATION

At a high level, Rapid operates in three phases. In the first phase (§ 4.1), Rapid (a) *groups* the set of specifications and (b) *partitions* the customer-supplied code to parallelize the downstream analysis. In the second phase, Rapid analyzes each specification group and code partition using a combination of a local typestate analysis (§ 4.2) and a global demand-driven reaching definitions analysis (§ 4.3). In the third phase, Rapid collects results from the analysis processes and provides feedback: reports of each violation found (paths into automata error states) and recommendations on how to fix them.

4.1 Preprocessing

Rapid takes as input Java source *and* jar files. Only class files with accompanying source code are analyzed, so that Rapid can provide source-level recommendations after analyzing the bytecode.

Class Partitioning Analysis time is correlated with the number of classes analyzed *simultaneously*, which influences the sizes of the call- and control-flow graphs, but also with the sizes of the analysis domains for both the forward and the on-demand backward analyses. Early versions of Rapid failed to scale to large codebases, running out of memory while loading the classes or incurring drastic slowdowns in call-graph construction, inter-procedural control-flow graph construction, and data-flow passes. Without some splitting mechanism, there would be a hard limit to the tractable codebase and analysis domain sizes. To maximize the number of recommendations provided within the latency requirements, we partition the input classes into partitions that can be analyzed independently by separate worker processes. Our partitioning traverses class directories and creates partitions via greedy bin-packing. This heuristic keeps, as much as possible, classes from the same Java package in the same partition, preserving method dependency locality.

In principle, partitioning can lead to false negatives due to potentially separating callers from callees. However, in practice, for the properties checked by Rapid, the performance-soundness tradeoff is highly beneficial. In § 6 we show that, on average, a partition size of 1,000 yields a 85% reduction in latency and 55% reduction in compute cost with only a 2% loss of recommendations.

Specification Fingerprinting Before the analysis commences, we must find the automata relevant to a codebase. An early implementation used reflection to determine which automata initializers *matched* each line of bytecode. This was prohibitively expensive and did not scale to checking hundreds of rules on multimillion-line codebases. Instead, Rapid precomputes fingerprints of all rules by creating a hash-table that maps the automaton-initializing class and method

```
Name: xxeDocumentBuilder

QuickMatchConstructor:
 "javax.xml.parsers.DocumentBuilderFactory":"newInstance"

States: [ initial, allSet, notEnough ]

InitialState: initial

ErrorStates: []

Transitions:
  - Source: initial
    Predicate: isCalled('build')
    Pre: getValuesSizeForKey('xxeconfig') == 2
    Target: allSet

  - Source: initial
    Predicate: isCalled(‘build')
    Pre: getValuesSizeForKey('xxeconfig') < 2
    Target: notEnough

  - Source: initial
    Predicate: >
      (isCalled('setFeature') &&
       isArgumentInSet('setFeature', $0,
      {'.../external-general-entities',
       '.../external-parameter-entities' })
       && isArgumentInSet('setFeature', $1, { false }))
    Pre: !isValueStored(‘xxeconfig', 'setFeature', $0)
    Target: initial
    Effect: store('xxeconfig', 'setFeature', $0)
```

```
Name: xxeParserUse

QuickMatchConstructor:
 “javax.xml.parsers.DocumentBuilderFactory": "newDocumentBuilder"

DependsOn: ["xxeDocumentBuilder"]

States: [ initial, createdFromSafe, error ]

InitialState: initial

ErrorStates: [ error ]

Transitions:
  - Source: initial
    Predicate: >
      !isCalled('parse') &&
      isAutomatonInState('xxeDocumentBuilder', 'allSet')
    Target: createdFromSafe

  - Source: initial
    Predicate: >
      isCalled('parse') &&
      (isAutomatonInState('xxeDocumentBuilder', 'initial')
       || isAutomatonInState('xxeDocumentBuilder', 'notEnough'))
    Target: error
```

Figure 3: A simplified specification for the XXE sanitization property described in § 2.2.

names — *e.g.* as specified via the value of the `QuickMatchConstructor` key in Figure 3 — with the corresponding automata specifications. Then, Rapid can make a fast linear pass over the customer's bytecode to eliminate irrelevant code using the hash-table, to determine relevant automata specifications and their instantiation locations.

Specification Dependency Resolution Recall that Rapid lets users modularize specifications by splitting them into smaller automata, typically one per type, that depend on each other (§ 3.4). For example, Figure 3 shows two automata, for types `DocumentBuilderFactory` and `DocumentBuilder`. We resolve dependencies between relevant specification automata to divide the set of specifications into *specification groups* comprising automata that can be analyzed independently. For example, given five specifications A–E with the dependencies $\{B \to A, C \to A, D \to B, E \to C\}$, dependency resolution groups them as $[\{A\}, \{B,C\}, \{D,E\}]$ where the specifications in each group can be analyzed independently *after* we have computed the results for the specifications in preceding groups.

Intermediate Representations Rapid loads classes into memory and builds an intermediate representation amenable to whole-program analysis. Java bytecode is first transformed to Soot's [45] typed, three-address Jimple representation. Next, Rapid performs a *Class Hierarchy Analysis* (CHA) [29] to build a call graph: based on the declared type of each receiver, it consults the class hierarchy to determine its possible runtime types and find out the set of concrete methods that may be called at the virtual call site. Then Rapid builds an inter-procedural control-flow graph (CFG). The latter captures exceptional edges for both explicit `throw` statements and implicit *unchecked* exceptions (*e.g.* array-bounds exceptions), which are modeled using Soot's `ExceptionalUnitGraph`.

4.2 Typestate Analysis

In the next phase, Rapid analyzes the IR for each class partition against each specification group by combining a *local* typestate analysis with a *global* reaching definitions analysis which is used to evaluate value-dependent predicates guarding the automata transitions.

IFDS/IDE Analysis Framework Soot's [23] Heros framework [10] simplifies the implementation of flow- and context-sensitive IFDS/IDE analyses [38]. An *Interprocedural Finite Distributive Subset* (IFDS) analysis has three inputs (G, D, F) where: G is the inter-procedural CFG, D is a finite analysis domain, and F represents flow functions which, for each kind of program statement, map each input flow-fact from D to a set of facts in 2^D. Each flow function $f \in F$ is *distributive*, *i.e.* $f(X \cup Y) = f(X) \cup f(Y)$, which allows point-wise data-flow propagation. An *Interprocedural Distributive Environment* analysis, IDE, extends IFDS with a bounded lattice L and *edge functions* $E : L \to L$ that describe how a particular lattice element is updated by each program statement, to compute, for each program point, a mapping $M : D \to L$ from data-flow facts D to elements of L.

Typestate Analysis using IFDS/IDE Rapid's typestate analysis is implemented as an IDE instance where: (1) the dataflow facts D correspond to pairs (v,P) of program variables v and (specification) automata P, and (2) the lattice L corresponds to the different automaton *states*, and edge functions are computed using the (guarded) automaton transitions. Rapid's IFDS instance creates "seed" facts (*i.e.* identifier-automata pairs) by iterating over program statements and checking if the statement has a method call that matches any of the precomputed (hashed) class- and method-pairs that initialize the automata (c.f. the `QuickMatchConstructor` in Figure 3). The IFDS framework then propagates these facts across assignments, branches, joins, calls and returns to compute the automata instances that reach each program point. Rapid's IDE instance uses $M : D \to L$, composes edge-functions over IFDS edges, and propagates the IDE values in L to compute the state of each automaton that reaches that program point. Finally, recall that Rapid analyzes each specification group *after* analyzing its dependencies, thus allowing the predicates (transitions) for each automaton to depend on those of its dependencies, enabling modular analysis (§ 3.4).

Summarizing Methods Rapid may not always analyze data-flow into a method: the bytecode may be absent from the class-path or partition, and some library methods (*e.g.* calls into methods from

`String` or `Collection`) may be abstracted to preclude their analysis. IFDS/IDE provides context-sensitive analysis using summaries. Additionally, Rapid uses a manually constructed and extensible set of 66 summaries to avoid analyzing common library methods.

4.3 Reaching Definitions Analysis

In the IDE analysis, the edge-functions $E : L \rightarrow L$ are computed for program locations where a subset of the transitions in the corresponding property automata are enabled, *i.e.* when the value-dependent predicates *guarding* the transitions evaluate to `true` (S 3.2). We evaluate such predicates via a reaching definitions analysis instantiated *on-demand* whenever we need to evaluate a guard. For example, a ParamValue(P_e, e', `sig`, `idx`, `rgx`) predicate uses an on-demand analysis to determine if the constants reaching `idx` match `rgx`.

Reaching Definitions via Boomerang Rapid's demand-driven, field-, flow-, and context-sensitive reaching definitions analysis is built atop the open-source Boomerang pointer-analysis framework [39, 40]. Boomerang provides aliases for a *query* on a program variable or field reference. We adopt Boomerang's backward queries, used to locate the allocation site of a variable or field reference, to locate a *source*—which, for Rapid, can be definitions of constants, annotated taint sources, or any other custom sources. A Boomerang backward query can be resolved recursively via multiple backward and forward queries. Instead of computing heap-aware aliases with forward queries, Rapid computes reachable sources up to the point of the triggering location, *e.g.* at the point of IDE edge-function resolution in the forward type-state analysis. For efficiency, in the backward analysis, we use the aforementioned summaries of libraries that are also used in the forward IFDS/IDE analysis. Boomerang uses IFDS internally, and fits naturally into our implementation.

Forward Typestate vs. Backwards Definitions When extending its predicate language, Rapid allows analysis designers to separate different kinds of information into forward and backwards analyses, which can be more efficient depending on the property being analyzed. For example, a predicate that checks if the current expression may be a hardcoded credential is best implemented as a backwards reaching-definitions analysis; this avoids unnecessary work, as most strings in a program are not credentials. Further, for credentials, a reaching-definitions analysis is sufficient: we do not care about the credential value, but only about whether it is hardcoded or not. In contrast, for properties like command injection or cross-site scripting, we want predicates that tell us if the current expression can carry tainted information. This information can also be efficiently computed via a forward data-flow analysis since the number of tainted sources, typically a predifined set of APIs, is smaller compared to all hardcoded strings. In the implementation, however, to have field sensitivity, we used demand-driven backward queries, for both these categories.

5 CASE STUDIES

5.1 File IO Streams

A common mistake is to forget to close `java.util.stream` objects returned by the `java.nio.file` APIs. These omissions can cause resource leaks that lead to denial-of-service attacks [8]. Rapid checks that for all APIs in `java.nio.file.Files` that return streams, the streams are eventually closed. The check also ensures that the error is flagged only when the stream is actually used.

Specification We specify this check using an automaton that places each created `java.util.stream` object in a typestate that denotes how the object was instantiated by the different `stream` constructing APIs *e.g.* `newDirectoryStream`, `newByteChannel`, `walk` *etc.*. The specification then uses an IsGarbage(·) predicate to ensure that the object does not *die* (*i.e.* go out of scope) in a *used* state, but must instead only do so in a *closed* state.

Findings The code below shows a typical use of `Files.walk` that creates a `stream` to select files with a given extension `ext`

```
Files.walk(path, FileVisitOption.FOLLOW_LINKS)
  .filter(Files::isRegularFile)
  .filter(f -> f.toString().endsWith(ext))
  .collect(Collectors.toSet());
```

Rapid reports a finding on the above, as the developer has forgotten the call to `close` on the `stream` created by `Files.walk`. Instead, Rapid reports the code as safe if she either added an explicit `close()` or, more idiomatically, wrapped the usage in a `try`-with-resources block which generates bytecode that closes the `stream`.

5.2 S3 Bucket Encryption

`AWS S3` is a distributed storage service [4] thats gives clients access to a scalable and reliable data store where the persisted values may optionally be encrypted depending on the client application. `S3` clients issue `PutObjectRequest`s to add an object to a container *bucket*. We use Rapid to check that if the programmer has specified that we perform server-side encryption (SSE) (before storing the object) along *some* path, then she should specify SSE on *all* paths, or she may be insecurely storing a sensitive object.

Specification The above requirement is a *relational* property that connects the behavior of the program on *different* paths [28]. We specify the above requirement by using automaton dependency (§ 3.4). First, we specify a three-state automaton for the `PutObjectRequest` objects, which represent the *requests* themselves. The request object starts off in the initial—*sse-not-set* state, and then transitions to *set-sse* or remains in *sse-not-set* depending on whether SSE is explicitly *enabled* or *disabled* on the request object. Second, we specify a typestate automaton for `S3Client` objects that checks that `putObject` method invocations are given `PutObjectRequest` arguments that are suitably configured *i.e.* in exactly one of the *sse-set* state *or* in the *sse-not-set* state, via an isAutomatonInState(·, ·) predicate (§ 3.4) and transition the `S3Client` to an error if the arguments can be in *both* states.

Findings The following illustrates use of the S3 `putObject` API [3].

```
PutObjectRequest putReq = new PutObjectRequest(...);
if (kmsKeyId != null) {
  var keyParams =
    new SSEAwsKeyManagementParams(kmsKeyId);
  putReq.setSSEAwsKeyManagementParams(keyParams);
} else { // SSE not set }
this.s3Client.putObject(putReq); // finding
```

The code initializes the request `putReq` and then, in one branch, calls `setSSEAwsKeyManagementParams` to transition `putReq` to the *set-sse* typestate indicating that SSE should be done for this object. However SSE is *not* set in the `else` branch and so, at the call to `putObject` the `putReq` object is in both *set-sse* and *sse-not-set* states thereby allowing Rapid to flag a violation.

5.3 Cryptographic Key Generation

Recall that Rapid is tailored toward checking API usage properties on *short-lived* requests and response objects. When a reference is stored in the heap, Rapid moves the type-state into a *malpractice* state (§ 1 and § 2). This feature lets Rapid flag potentially insecure uses of general purpose Java cryptographic libraries like `java.security.*` and `java.crypto.*`. For example, suppose we want to check that API clients (1) only use a whitelisted subset of *compliant* cryptographic algorithms `AES`, `HmacSHA224`, `HmacSHA256`, `HmacSHA384`, `HmacSHA512`; (2) use secure keys whose size is *compatible* with the selected cryptographic algorithm, *e.g.* if the library is instantiated to use the `AES` then the keys must be of size `{128, 192, 256}` in the initialization step; and (3) correctly complete the API call sequence, *e.g.* do not forget to call `generateKey()` either after instantiation and initialization.

Specification The requirements (1) and (2) can be specified using parameter value predicates that restrict the values of arguments passed into the calls that construct the cryptographic instance and the key instantiation respectively. The requirement (3) can be encoded with a guard that uses IsGarbage(·) to stipulate that the instance is correctly configured before it goes out of scope.

Findings Rapid flags a warning in the code below where the developer instantiates a `javax.crypto.KeyGenerator` and writes it to the heap via the `aesKeyGen` field which is used later to complete the initialization with an *incompatible* 244-bit key.

```
public CmpResponder(...) throws NoSuchAlgoException {
  aesKeyGen = KeyGenerator.getInstance("AES"); // finding
}
synchronized (aesKeyGen) {
  aesKeyGen.init(224);  // wrong key size
}
```

Java best-practices recommend storing security-sensitive values like the `KeyGenerator` instance in short-lived stack allocations and not on the heap where they may remain ripe for stealing indefinitely until they are overwritten or reclaimed by garbage collection [6]. Rapid reports an error where the `KeyGenerator` instance is assigned to `this.aesKeyGen` as after that assignment, the instance goes to the heap, and hence the *malpractice* state.

5.4 Hard-Coded Credentials

Rapid's backward analysis lets us check that code does not use hard-coded passwords. Java's `javax.crypto.SecretKeyFactory` is an API for creating encryption ciphers. We use parameter value predicates to check two properties of clients: (1) the `SecretKeyFactory` is only instantiated using an element from a whitelisted set of compliant algorithms; and (2) the `generateSecret` method is only invoked with `PBEKeySpec` keys that are *not* derived from hardcoded passwords, as such passwords may be intercepted by a debugger or heap dump.

Findings The following snippet shows how Rapid finds a property violation where a secret `key` may be a constant at least along one path. In `MDfromHTMLUtils` line 583 we have a path where a constant defined in `MDfromHTMLConstants` is returned and flows into `key`, triggering the warning at line 165 in `MDfromHTMLCrypto`.

```
// MDfromHTMLCrypto line 165
var secretKey = SecretKeyFactory
  .getInstance("PBEWithMD5AndDES")
  .generateSecret(new PBEKeySpec(key));
```

```
// MDfromHTMLUtils line 583
if (pwKey == null || pwKey.trim().length() == 0) {
  pwKey = System.getProperty(..);
  if (pwKey == null || pwKey.trim().length() == 0)
    return Constants.PROPERTY_FILE_KEY;
}
```

6 EVALUATION

Next, we present an evaluation of Rapid along three dimensions: the checker's *performance* (§ 6.1), the importance of *partitioning* (§ 6.2), and the overall *precision* of the generated warnings (§ 6.3).

6.1 Performance

First, (how) can Rapid efficiently analyze large code bases within the latency requirements needed for code review?

Subject Selection We ran Rapid in repository analysis mode on a selection of subjects from Maven Central [11]. We started with a set of 26,142 subjects, randomly selecting a version of each package among Java projects with both source code and bytecode (jars). We then selected a representative subset, in the order of hundreds of subjects, for detailed analysis, by binning all the subjects along three dimensions: (1) size in lines of code; (2) number of APIs relevant to properties checked by Rapid; and (3) number of field writes for types that are relevant to the properties. We used stratified sampling on each distribution to get 100 subjects per distribution. For each distribution, we randomly selected 10 of the 100 subjects, resulting in 30 subjects that represent a wide range of values from the different strata in each of the three dimensions.

Experimental Setup We devised different configurations of Rapid and executed them on the Maven analysis subjects, including: (1) Rapid with input partitioning (§ 4.1) with different fixed partition sizes; and (2) Rapid with per-property partitioning (§ 4.1), analyzing the input with one property at a time. Experiments were run on an Intel Xeon system with a 2.50 GHz, 64-bit 48-core processor, and 192 GB RAM, running Linux 5.4.95, with hyper-threading. Each configuration described above is "embarrassingly" parallel, so we present both the total sequential and the shortest possible parallel runtime.

Results Table 1 summarizes Rapid's performance on Maven subjects selected to cover a range of package sizes, APIs and properties analyzed (cf. § 6.1). The analyzed packages span in size from 5.6 KLOC to 1.8 MLOC. Column *APIs analyzed* shows that a property can be instantiated at multiple code locations and that total property instantiations in a subject range from 1 to at most 132 in Apache CXF. *Distinct properties* shows that each subject instantiates between 1 and 17 different properties during the analysis. *Analysis domain size* is the size of the analysis domain of the forward type-state analysis. *Total queries* is the number of on-demand backward analysis queries, *Analysis Time* is the total time of the forward and backward analysis.

Prevalence of Malpractice Recall that Rapid's division of labor between a local typestate and global reaching definitions analyses (§ 2) is predicated on the assumption that the code uses the pattern of *short-lived* response objects which should not be persisted in fields. We validate this assumption by measuring *Malpractice states*: the number of code locations where applications write values of property-dependent types to the heap. This number is zero or low for most subjects. For a few subjects, such states are reached a few dozen times: a small fraction of the analysis domain, and crucially,

Table 1: Run-time performance measured when using Rapid to analyze a representative subset from 26,142 open-source packages.

Package name	Lines of code	APIs analyzed	Distinct properties	Analysis domain size	Total queries	Errors found	Malpractice states	Java heap usage (MB)	Analysis time (s)	Max worker time with fixed-size partitioning (s)
QuickFIX/J 2.1.1	1,817,376	21	7	40	5,680	15	0	8.024	679	40
Kuali Coeus	1,440,633	14	5	34	462	8	0	6,967	239	31
HAPI HL7 FHIR	430,964	25	10	31	138	11	0	6,102	210	74
Infinispan Embedded Query	413,431	27	8	55	2,256	19	4	5,588	1,417	75
Kill-Bill Billing, Dwolla Plugin	376,813	65	17	151	4,117	25	17	6,679	1,199	69
Apache Servicemix Bundle	349,458	18	6	57	728	6	1	2,171	165	28
Yugabyte Java Driver Core	323,976	45	12	250	1,103	6	2	3,167	271	44
Daisy Saxon-HE	188,153	33	8	98	5,426	14	2	3,697	755	100
Apache CXF Bundle	148,449	132	16	298	10,452	30	21	1,803	1,017	489
Glassfish Web-Core	58,609	82	16	334	11,782	34	23	1,247	690	690
IDPF EPUBCheck	23,953	33	8	44	1,578	16	1	227	120	117
MyBatis 2	14,581	13	5	15	333	4	0	115	14	12
Apache Felix Webconsole	9,556	30	6	149	124	3	1	82	7	5
AgNO3 Java dbus-core	7,176	10	3	39	268	2	2	83	9	6
Mil-OSS FGSMS Data Access	5,599	1	1	2	526	1	0	81	30	28

less frequent than the (small) number of *Errors found*. This shows such field writes are indeed rare, validating our design choice.

Running Time Table 1 shows Rapid analysis time without partitioning. As expected, analysis time is a function of the cost of forward and backward analysis. The largest analysis times are incurred on subjects like Apache CXF and Kill-Bill, with analysis domain sizes in the hundreds, and with thousands of on-demand queries. Call-graph construction and class loading time are typically a small fraction of analysis time, but can go up to 18% of it in a few cases. The last column of Table 1 shows the longest analysis time taken by any worker to analyze a partition when the same subject is analyzed *with* partitioning. A fixed partition size of 1,500 classes is used, as also in production.

6.2 Partitioning

Next, we evaluate the effectiveness of our partitioning strategies: are they needed to make the analysis efficient, and does this efficiency come at the cost of missing property violations?

Experimental Setup We evaluate our partitioning strategy, as described in § 4.1, by running the 10 largest subjects from Table 1 with partition sizes ranging from 20,000 to 1,000 classes. Each partition is analyzed by a single worker, and we set a timeout of 10 minutes for each worker. Figure 4 summarizes the results, by showing, for each partition size: (1) number of errors reported by Rapid (bars, left Y-axis); (2) total analysis time across all partitions; (3) parallel analysis latency *i.e.* maximum execution time of a partition.

Results: Code Partitioning Our results show that partitioning is essential. Partitioning too coarsely (or not partitioning at all) can result in losing all (or most) findings, as the partitions are too large to be efficiently analyzed. Instead, smaller partitions are readily digested by Rapid and, thanks to the locality of the properties, rarely result in a loss of findings. These trends are illustrated in Figures 4a–4c which exemplify recurrent patterns. For Infinispan (Fig. 4a), a partition size of 20,000 results in zero findings: all partitions time out. A size of 5,000 yields just one finding, as most partitions time out. A more moderate partition size of 4,000 yields all 19 findings. The same pattern can be observed in 4 of the 10 largest subjects. Interestingly, even the *total* compute effort decreases with smaller partition sizes. Since the time complexities of the forward and backward analyses are cubic in the analysis domain size, we attribute this to the fact that the sum of the cubes is smaller than the cube of the sums, as observed in all subjects. In a few cases like Coeus (Fig. 4c), very small partition sizes cause the initialization overhead to become significant and increase the total cost, although the best latency still decreases.

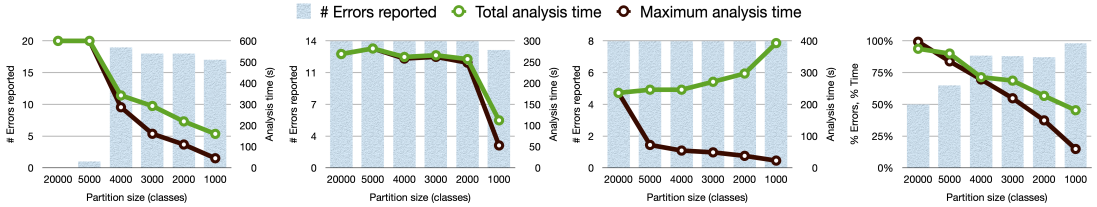
Of course, reducing the partition size can also entail some loss of findings, as the analysis misses certain important value flows. This can be seen in Figs. 4a and 4b. However, such loss is modest relative to the gains in latency and cost, as shown in Figure 4d, which shows the average over all 10 largest subjects. We believe this stems from the fact that most of the property violations are fairly local. Consequently, Rapid currently runs in production with a partition size chosen by the same methodology.

Results: Property Partitioning We also evaluated a configuration of Rapid that analyzes a subject by focusing on particular property specification groups at a time. Focusing on one property reduces the size of the analysis domain, and hence the analysis time. A parallel implementation can exploit this to incur only the latency of the most expensive property. On our analysis subjects, assuming an implementation where the running time is approximately that of the slowest worker, Rapid with property partitioning provides a speedup of 40% (ratio of geometric mean on all subjects) over the configuration used in Table 1. The results imply that in future, based on customer traffic, we can combine input and property partitioning effectively.

6.3 Precision

Finally, we get to the ultimate proof of the pudding: how *precise* or *useful* do developers find the results produced by Rapid?

Internal Deployment Results We deployed Rapid on Amazon's internal code review system, used daily by thousands of developers. Rapid runs on a code review request once the build artifacts for the code change are generated. Currently, Rapid scans the whole package on each code review, and meets the latency requirements. During code review, the user has the option to mark the recommendation as *useful*, *not useful*, or *not sure*. We define the *acceptance* metric for a given week as the fraction of recommendations marked useful out of all those marked either useful or not-useful during that week. We find that averaging over all properties, Rapid's recommendations



(a) Infinispan (b) Daisy (c) Coeus (d) Average

Figure 4: Plots (a), (b), and (c) illustrate on three subjects the main insights that characterize the tradeoffs of partitioning. Plot (d) shows the average of the normalized results for this experiment (the top 10 largest subjects from Table 1). Three costs are shown: total errors found, total analysis time (*i.e.* computational cost), and maximum analysis time for any worker (*i.e.* smallest possible latency).

Table 2: Rapid results on OWASP security vulnerability categories.

Category	CWE	TP	FN	TN	FP	Score %
Command Inject.	CWE-78	126	0	27	98	21.60
Cross-site Scr.	CWE-79	246	0	95	114	45.45
Insecure Cookie	CWE-614	36	0	31	0	100.00
Ldap Injection	CWE-90	27	0	12	20	37.50
Path Traversal	CWE-22	133	0	42	93	31.11
Sql Injection	CWE-89	272	0	80	152	34.48
Weak Encrypt.	CWE-327	130	0	116	0	100.00
Weak Hashing	CWE-328	129	0	107	0	100.00
Xpath Injection	CWE-91	15	0	3	17	15.00
Total		**1114**	**301**	**831**	**494**	**62.29**

have an acceptance rate of 76% over a period of four weeks, indicating that developers find the vast majority of the reports useful.

RAPID produced recommendations for 64 distinct properties on entire repositories, and for 34 distinct properties on changed lines of code. 329 distinct developers participated in these code reviews. For the recommendations on changed lines of code, 85% of the properties had 50% or higher acceptance rate, 63% of the properties had 75% or higher acceptance rate, and 57% of the properties had 100% acceptance rate from the developers who authored the changes.

OWASP Benchmark Results Table 2 shows Rapid's results for the OWASP benchmark [13]. Each row is associated with a *common weakness* from the CWE [7] taxonomy. We report on the 9 (of 11) benchmark categories for which we have property specifications. The two excluded categories are CWE 501, which is non-exploitable and will soon be removed [14] by OWASP; and CWE 330, which we disabled in production despite a 100% TPR because too many developers marked it *not useful* in code review feedback. The table shows the difference between true positive rate (TPR) and false positive rate (FPR), called the *score*. On these 9 categories, Rapid has a 100% TPR. The taint tracking properties, and pure type-state properties such as Insecure Cookies are detected with total recall. Rapid has a 37.71% FPR on these OWASP categories because the analysis is (1) *not* path-sensitive and sometimes reports tainted flows in infeasible paths; and (2) *not* precise w.r.t. arrays and collections. Fortunately, our deployment shows this imprecision is rare in real-world code, so we eschew complicating Rapid's analysis solely for OWASP.

7 RELATED WORK AND CONCLUSIONS

Rapid builds on a large literature on deploying static code analyses.

Code linters like LCLint [32], Prefix [25] and Meta-level Compilation [31] focussed on finding suspicious anti-patterns in C/C++ code bases. Recent Java linters include Findbugs [33] which led to many others that are collected under easy to reuse umbrella-frameworks like Error-Prone [9] and the Software Assurance Marketplace which provides error-reporting plugins for various VCS and IDEs [16].

Typestate Specifications Rapid's mechanism for writing typestate-based [41] API specifications is related to the specification languages used to specify kernel API protocols for model checkers [21, 22]. The PQL system [36] introduced query languages to describe error patterns for Java security flaws or resource leaks that could be found by dynamic analysis. CRYSL [35] provides a higher level specification language which allows writing typestate properties that pertain to correct use of Java cryptographic APIs [34]. Rapid aims to support a wider class of properties, and supports automata dependencies, and key-value stores to enable modular, reusable and compact specifications, with reasonable property-development time.

Analysis Algorithms and Frameworks Rapid uses the IFDS/IDE analysis paradigm introduced by [38], as implemented in the Heros library [20, 23] in the Soot Java bytecode analysis framework [44]. The Wala [18] framework underlies scalable *taint* analyses like Taj [43] and Andromeda [42]. Infer provides a compositional framework for analyzing heap-sensitive properties of Java programs using abstract domains based on separation logic [26, 27]. *Declarative* frameworks specify pointer analyses as logic programs that are then executed using BDD [46] or Datalog [19, 24] backends. The checker Framework [17] is a flexible way to build plugin type checkers for Java but requires expertise to implement each check while Rapid simplifies the specification of typestate properties for restful APIs.

Conclusions Rapid follows many recent applications of program analysis in industry, *e.g.* Tricorder which composes multiple linters and analyses under an umbrella used for CI inside Google [37], and the Infer and Zoncolan systems used to check C, Java and PHP code bases at Facebook [30]. In contrast, we designed Rapid to meet the latency constraints particular to our use case: allowing third-party API clients to check their own library usage, and show this is possible by combining local and global analysis and using a *malpractice* type-state to sidestep heavyweight reasoning about the heap with minimal loss of precision.

References

[1] [n.d.]. Amazon CodeGuru. https://aws.amazon.com/codeguru.
[2] [n.d.]. Amazon CodeGuru Reviewer. https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html.
[3] [n.d.]. Amazon S3 PutObject API. https://docs.aws.amazon.com/AmazonS3/latest/API/API_PutObject.html.
[4] [n.d.]. Amazon Simple Storage Service (Amazon S3). https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html.
[5] [n.d.]. Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss/.
[6] [n.d.]. CWE-244: Improper Clearing of Heap Memory Before Release ('Heap Inspection'). https://cwe.mitre.org/data/definitions/244.html.
[7] [n.d.]. CWE: Common Weakness Enumeration. https://cwe.mitre.org.
[8] [n.d.]. Denial of Service (DoS) attack. https://owasp.org/www-community/attacks/Denial_of_Service.
[9] [n.d.]. Error Prone. https://errorprone.info/.
[10] [n.d.]. Heros IFDS/IDE Solver. https://github.com/Sable/heros.
[11] [n.d.]. Maven Central Repository. https://maven.apache.org/.
[12] [n.d.]. OpenAPI Specification. https://swagger.io/specification/.
[13] [n.d.]. OWASP Benchmark: The OWASP Benchmark Project. https://owasp.org/www-project-benchmark.
[14] [n.d.]. OWASP Benchmark: Trust Boundary Violation (CWE 501) Not Exploitable. https://github.com/OWASP/Benchmark/issues/43.
[15] [n.d.]. OWASP XXE: OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html.
[16] [n.d.]. SWAMP: Software Assurance Marketplace. https://continuousassurance.org/swamp-in-a-box/.
[17] [n.d.]. The Checker Framework. https://checkerframework.org/.
[18] [n.d.]. T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.
[19] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *PLDI*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. https://doi.org/10.1145/3385412.3386026
[20] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299
[21] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2999)*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-540-24756-2_1
[22] Dirk Beyer, Adam Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. The Blast Query Language for Software Verification.. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3148)*, Roberto Giacobazzi (Ed.). Springer, 2–18. https://doi.org/10.1007/978-3-540-27864-1_2
[23] Eric Bodden. 2012. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (Beijing, China) *(SOAP '12)*. Association for Computing Machinery, New York, NY, USA, 3–8. https://doi.org/10.1145/2259051.2259052
[24] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108
[25] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exp.* 30, 7 (2000), 775–802.
[26] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33
[27] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700
[28] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393
[29] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101.
[30] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112
[31] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, Michael B. Jones and M. Frans Kaashoek (Eds.). USENIX Association, 1–16. http://dl.acm.org/citation.cfm?id=1251230
[32] David Evans, John Guttag, James Horning, and Yang Meng Tan. 1994. LCLint: A Tool for Using Specifications to Check Code. *SIGSOFT Softw. Eng. Notes* 19, 5 (Dec. 1994), 87–96. https://doi.org/10.1145/195274.195297
[33] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 132–136. https://doi.org/10.1145/1028664.1028717
[34] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 931–936. https://doi.org/10.1109/ASE.2017.8115707
[35] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *ECOOP' (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10
[36] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 365–383. https://doi.org/10.1145/1094811.1094840
[37] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 598–608.
[38] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1-2 (1996), 131–170.
[39] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290361
[40] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali 0001, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22
[41] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929
[42] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *FASE (Lecture Notes in Computer Science, Vol. 7793)*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15
[43] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 87–97. https://doi.org/10.1145/1542476.1542486
[44] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13. https://dl.acm.org/citation.cfm?id=782008
[45] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *Proceedings of the 9th International Conference on Compiler Construction (CC '00)*. Springer-Verlag, Berlin, Heidelberg, 18–34.
[46] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859