Persisting and Reusing Results of Static Program Analyses on a Large Scale

Johannes Düsing
TU Dortmund
Dortmund, Germany
johannes.duesing@tu-dortmund.de
ORCID: 0000-0002-9367-2206

Ben Hermann
TU Dortmund
Dortmund, Germany
ben.hermann@tu-dortmund.de
ORCID: 0000-0001-9848-2017

***Abstract*—Static Program Analysis (SPA) has long been established as an important technique for gaining insights into software systems. Over the last years, analysis designers increasingly produced analyses that are compositional, collaborative, or incremental in nature - thus relying on some form of existing results to increase performance or even precision. However, systematic result reuse is still rare in this field even though the analyzed software is mainly composed of reusable software components.**

For this work, we study 40 state-of-the-art SPA implementations and find that there is a tremendous potential for reusing analysis results. We attribute this to the fact that there is no systematic process in place for persisting and sharing analysis results and propose such a process here to fill this gap. In this paper, we present SPARRI, a prototype implementation providing an HTTP API to publish, search, and reuse SPA results. Our evaluation shows that reusing existing results with SPARRI can improve analysis performance by up to 92%. Furthermore, we see potential in applying it to other research areas like empirical software studies, benchmark creation, and artifact evaluation.

***Index Terms*—Static Program Analysis, Modular Analysis, Result Reuse, Repository Mining**

I. Introduction

The practice of reusing existing software artifacts has been shown to aid overall software quality and increase productivity [1]. Popular software package repositories like Maven Central host millions of packages [2], which are openly accessible to anyone and used by thousands of software developers every day.

Despite its apparent popularity, software reuse also comes with some distinct disadvantages. Recent incidents like *Log4Shell* [3], as well as the intentional compromise of *colors.js* and *node-ipc* underline the enormous impact that a single flawed software artifact may have on its ecosystem.

In each of these cases, thousands of software packages and projects became either vulnerable or stopped working altogether. As a result of this growing impact of third-party software, researchers have turned their attention to analyzing the usage [4], quality [5], and security of such artifacts [6].

Such inspections are often implemented via *Static Program Analysis* (SPA), which enables researchers to soundly over-approximate the runtime behavior and to compute properties of a software system without executing its code. Analyses are used to investigate the propagation of vulnerabilities [7], [8], to prove the absence of certain types of bugs [9]–[11], as well as to quantify the quality of a software package via *Software Quality Metrics* [12]–[14]. Their results are often used to identify vulnerable software libraries, characterize their quality, and derive guidelines for further improvement. Guarantees that are proven via static program analysis can also help in proving the correctness of software.

As a result of those heterogeneous aims, static analyses exist in a variety of different configurations. Those may differ in their purpose, the properties being computed, their type of result, the framework used for implementation, and the target language of the programs being analyzed. Historically, a common factor has been that the final results of static analysis are after being interpreted in their respective domain, either discarded or archived. Consequently, sharing and reusing existing software analysis results, much like reusing software itself, requires a dedicated process and adequate tooling, which do not exist today.

However, some analyses would in fact strongly benefit from reusing existing results, both in terms of the analysis runtime and memory consumption. An example of this is the use of callgraphs. Callgraphs are the foundation for a number of other analyses, including dataflow analyses, abstract interpretation, or reachability analyses. Furthermore, the construction of *whole-program* callgraphs by combining the callgraphs of individual libraries has been shown to improve performance [15], [16]. Another trend supporting this claim is the emergence of *incremental* [17]–[20] and *demand-driven* analyses [21], which reuse results of previous runs or employ lazy computations to optimize performance.

Despite those advantages, the structured reuse of analysis results is a complex problem that we found did not receive much attention in the research community to this date. We hypothesize that this is mainly because of two reasons:

- Analyses use different, often domain-specific data formats to output results. While standardized formats like *SARIF* [22] exist, they do not sufficiently capture every aspect of static program analysis in general. For SARIF, we do not observe widespread adoption in state-of-the-art analysis implementations from the research community.
- There is no common platform for sharing or reusing static analysis results, so reuse would require manual search and

evaluation.

Our goal in this paper is to address this issue and provide researchers with means to build atop existing SPA results or share their results with others. To this end, we make the following contributions:

- We inspected 40 analysis implementations and found that 38% do not persist their results at all, and for another 50% the authors do not provide any information on how results are stored.
- We propose a novel system for computing, publishing, and reusing the results of SPA. It adopts a *Blackboard Architecture*, which is inspired by existing analysis frameworks like OPAL [23], but lifted to the scale of software component ecosystems.
- We implemented this system (named *SPARRI*) based on Scala and provide a RESTful Web-API for accessing results.
- We evaluated compositional analyses using SPARRI and found that our approach can be up to 92% faster than whole-program analyses. We also show that SPARRI can help researchers conduct empirical studies on software and share their results.

II. State of the Art

In this section, we systematically analyze the current state-of-the-art in applications of static program analysis. Our goal is to understand what results are produced, and how researchers deal with (potential) reuse of such results. We aim to answer the following research questions (RQs):

- What results are produced by SPA applications? (**RQ1.1**)
- How are results of SPA applications structured? (**RQ1.2**)
- By what means are SPA results persisted? (**RQ1.3**)
- To what extent are SPA results reused? (**RQ1.4**)

To answer those questions, we investigate a total of 40 publications on SPA applications and employ an open card-sorting approach [24] to identify relevant categories.

We build a corpus of relevant publications following the guidelines for Structured Literature Reviews proposed by Stapić et al. [25]. In that, we define four *Search Terms* (**T1** - **T4**) that we use to query the interfaces of the *ACM Digital Library*, *IEEExplore* and *ScienceDirect*, which are relevant indexing services for scientific publications [26]. Furthermore, we define *Inclusion Criteria* (**I1** - **I8**) and a *Quality Checklist* (**Q1** - **Q4**). Most notably, we restrict ourselves to publications of the last ten years (**I8**). All terms and criteria are presented in detail in our artifact[1].

For each RQ, we manually assign text labels to every publication in our corpus. We then group similar labels into categories and assign category names. Finally, we use the size of those categories (i.e. the number of publications) to judge their relevance and answer our RQs.

[1]In case of acceptance we will publish a research artifact via Zenodo. For now, we provide an anonymized version: https://anonymous.4open.science/r/sparri-artifact-3C98

A. Results

For each RQ we highlight the categories obtained via card-sorting and their prevalence in the set of publications. An overview of those results is presented in Table I.

1) RQ1.1: What results are produced?: For this RQ, we investigate the semantic meaning of SPA results and their distribution in our dataset. The most prevalent category of results semantics is *Control- and Dataflow* (**C1.2**) information, which is attributed to 40% of all publications. Here we observe that Control-Flow Graphs (CFGs) are often enriched with additional dataflow information, which is why the category spans both. In particular, edges are often annotated [27] or nodes inserted [28]. Other approaches represent dataflow with nodes for program states and transitions between them [9], [10], [21].

23% of all publications belong to category **C1.1**, which describes so-called *Summary-Based* dataflows. Here we found SPA applications computing summarized dataflows for certain structural entities like methods or functions. This is often done to compose local analysis results into global ones:

> *"The procedural transfer function [...] summarizes its points-to side-effects between its formal-in parameters and formal-out parameters [...]. When analyzing a call site, the analysis reasons [...] by applying its procedural transfer function [...]."* [11]

Most applications summarize per-method (or function) [11], [29]–[34], but depending on the domain, other granularities like per-thread [20] or per-intent (Android) [35] are possible.

Other categories are *Points-To Information* (**C1.3**) and *Security and Safety* (**C1.4**), the latter containing analyses focused on system calls [36], fault tolerance [37], general program termination [38] and more. Least prevalent, we find two publications concerned with emitting *Program- and Bytecode* (**C1.5**) [39], [40], as well as two publications calculating *Descriptive Characteristics* (**C1.6**) like metrics values [13] and roles of program variables [12].

2) RQ1.2: How are results structured?: We now present our results on the general formal structure of SPA results. The most popular structures we observed are *Graph Structures* (**C2.1**) with 35% of all publications, and *Maps* (**C2.5**) with 28%. For **C2.1** we see that oftentimes graphs are used to describe control- and dataflows [10], [17], [21], [27], [28], [41]–[44], optionally using edge labels, node labels or partitions to further annotate information. We also observe trees being used to summarize control flow [31]. As to **C2.5**, we report that program variables are predominantly used as keys for maps, often mapping to sets of allocation sites [18], [33], [45]–[47], but also to access permissions and mutability information [48]–[50]. Maps are also used to associate program locations to abstract states [51], [52].

Less prominently, authors also use *Traces* (**C2.2**) to report taint- and dataflows, general *Function Summaries* (**C2.4**) and plain *Algebraic Structures* (**C2.3**), including vectors [12], *Hoare Triples* [38], and numbers [13]. Four publications do not fit into any of the previous categories (**C2.6**), which use

TABLE I
CATEGORIES RESULTING FROM OPEN CARD-SORTING WITH THEIR RESPECTIVE NUMBER OF PUBLICATIONS CONTAINED

RQ1: Semantics of SPA Results		**RQ2:** Structure of SPA Results		**RQ3:** Technical Storage Solution		**RQ4:** Usage of Existing Results	
Category	**Papers**	**Category**	**Papers**	**Category**	**Papers**	**Category**	**Papers**
C1.1 Summary-Based Dataflow	23% (09)	**C2.1** Graph Structures	35% (14)	**C3.1** In-Memory Only	37% (15)	**C4.1** Summaries	18% (07)
C1.2 Control- and Dataflow	40% (16)	**C2.2** Traces	13% (05)	**C3.2** File-Based	10% (04)	**C4.2** Points-To Data	10% (04)
C1.3 Points-To Information	15% (06)	**C2.3** Algebraic Structures	7% (03)	**C3.3** Structured	3% (01)	**C4.3** Execution Paths	7% (03)
C1.4 Security and Safety	12% (05)	**C2.4** Function Summaries	7% (03)	**C3.4** No Information	50% (20)	**C4.4** Semantic Graphs	13% (05)
C1.5 Program- and Bytecode	5% (02)	**C2.5** Maps	28% (11)			**C4.5** No Usage	42% (17)
C1.6 Descriptive Characteristics	5% (02)	**C2.6** Misc	10% (04)			**C4.6** Previous Results	7% (03)
						C4.7 Metric Values	3% (01)

structures that are either very specific (Bytecode [40], Sandbox Whitelist Policies [36], Java Program Slices [39]), or formats that are not reported at all [53].

3) RQ1.3: By what means are results persisted?: Throughout the entire set of publications, we find that technical details of SPA result storage is an aspect rarely discussed. For 50% there is no information (**C3.4**) on this topic, another 37% do not persist the results of SPA at all, instead keeping them in-memory only (**C3.1**). 10% of publications used one [9], [36], [40] or multiple [29] files (**C3.2**) for storing SPA results:

> *"The partial summaries [...] are stored in a .res file on the machine where the analysis is performed. This .res file needs to be transferred to the target machine [...]."* [29]

We found that one publication uses a structured storage solution (**C3.3**), namely a database, for storing SPA results [12].

4) RQ1.4: To what extent are results reused?: We investigate if and how analyses make use of existing results. While for this aspect we derive the largest number of categories, many analyses still do not incorporate any existing results at all (**C4.5**). However, those analyses often compute generic inputs like points-to information [44], [45], [54] or callgraphs [11], [46] themselves in every analysis run. In fact, some authors explicitly acknowledge the fact that such computations are indeed a time-consuming necessity:

> *"[...] to decompose the expensive cost of the exhaustive points-to analysis, we [...]. The points-to analysis is performed along with a thread call graph in a bottom-up fashion [...]."* [11]

Other publications go one step further and enable the reuse of points-to data (**C4.2**) [19], [27], [47], [55] or semantic graphs (**C4.4**) [17], [18], [21], [28], [49] like callgraphs. A similar observation can be made for categories **C4.1** (*Summaries*) and **C4.6** (*Previous Results*), where external or previous results are reused to speed up analyses.

The latter includes domain-specific results like termination-certified modules [38], program modules with associated dependencies [20], or specific abstract states from previous iterations [52].

10% of all publications consume program slices or traces (**C4.3**) [39], [53], [56], while one publication consumes externally computed metric values for individual program components in order to aggregate a whole-system metric value [13].

We note that there are only two analysis implementations [10], [13] that reuse results produced by *other external analyses*. All other reuse happens for data produced in the context of the same publication, i.e. by preprocessing steps or via incremental computations.

B. Discussion

Our analysis of state-of-the-art SPA implementations highlights some interesting aspects regarding the handling of results.

First, we note that 55% of SPA results are of a very general nature (**C1.2**, **C1.3**) and potentially applicable to many different research contexts. Adding to that, the structure of SPA results is often well-defined and easy to process, with graphs (**C2.1**), general algebraic structures (**C2.3**) and maps (**C2.5**) making up 70% of all publications.

Observation 1: Many SPA results are of general nature (55%) and have a well-defined structure (70%).

Furthermore, we observe that authors seem to understand the benefits of SPA result reuse, with 58% of publications reusing some form of result. Even for analyses without reuse, we see authors re-implementing common functionality to obtain points-to sets or callgraphs, although sophisticated and well-tested implementations exist for those purposes. However, reuse almost exclusively happens within the context of a single publication, we notice that only two publications reuse externally computed results. As we observed many SPA implementations producing generally applicable results of straightforward structures (Observation 1), we hypothesize that this fact is due to a lack in supply: There is no straightforward way to reuse existing results for SPA in a structured manner.

Observation 2: Most analyses (58%) consume some pre-computed results, but rarely from external sources (5%).

Contrary to the potential for reuse we identified earlier, we find that a large majority of publications are not concerned with persisting analysis results longer than a single execution. In fact, we see a majority of publications that either do not persist their results (37%) or do not describe the handling of results altogether (50%).

The ones that do often use files to store data, without providing any information about the concrete data format used for serialization. This implies that the reuse of those results is

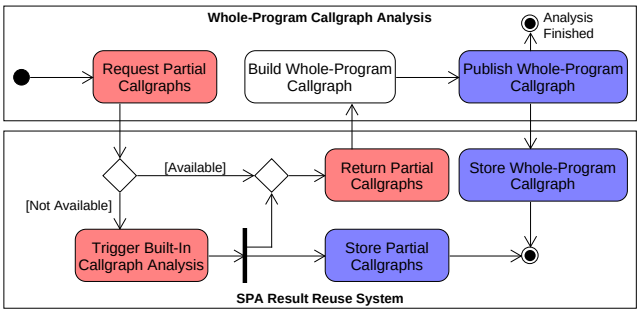

Fig. 1. Example for an SPA result reuse process

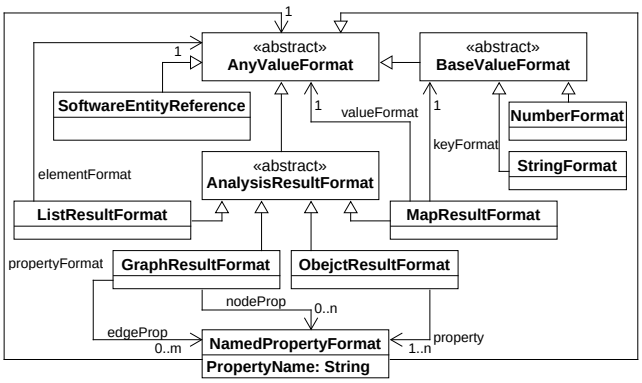

Fig. 2. Data model for SPA result formats

tedious for others, as it requires manual reverse-engineering of the data format used.

Observation 3: Only 13% of publications mention persistent storage of results, but even those cases do not provide further information on the concrete data format.

Overall, we made what seem to be contradictory observations: There is a sufficient supply of generally-applicable SPA analysis results, as well as an apparent benefit to reusing them. However, researchers rarely reuse external SPA results and do not discuss persisting results.

We argue that this is mainly because no structured process for SPA result reuse exists. To address the problem, in the following, we propose such a process based on our findings.

III. Design

A process for reusing SPA results must define ways to *publish* results to a *persistent* storage and to provide *structured access* to them. Furthermore, in order to foster reliability and traceability, process participants must be able to judge the *context and quality* of results. Finally, to design a process that is relevant to current SPA researchers, all major result structures and data formats that we observed in state-of-the-art SPA implementations must be supported.

In this section, we propose a process that implements the above-mentioned goals. Figure 1 exemplifies how this process looks like for the example of whole-program callgraph construction. We, first, present a data model that captures the most prominent SPA result formats, followed by an in-depth description of the entire process.

A. Result Format Definition

Providing *structured access* to SPA results requires a data model defining the set of supported result formats. There are two conflicting goals for such a model: It must be *broad* in a sense that it should cover all major result structures observed in Section II, but also *not* allow for *arbitrary* structures, so that result semantics are preserved and clearly communicated to potential consumers.

Based on our findings from Table I, we define a data model for SPA result formats. The model incorporates *Graph Structures* (**C2.1**), *Maps* (**C2.5**), *Objects* and *Lists*. The latter is used to model vectors (covering **C2.3**) as well as *Traces* (**C2.2**), which can be seen as lists of strings. In total, the model covers 83% of all structures we observed in state-of-the-art implementations.

However, we do not want our process to work on maps, graphs, objects, and lists of *arbitrary data*, as this would greatly reduce expressivity and usability. Instead, our data model also defines the format of basic result elements. We observe that results often reference concrete software entities: Points-to sets refer to *Program Statements*, callgraphs refer to *Methods* or *Classes*, and metrics refer to entire *Programs* or *Libraries*. Consequently, we introduce the notion of *Software Entities* to our data model.

A UML representation of our model is given in Figure 2. It defines the format of an analysis result to be either a `ListResultFormat`, a `GraphResultFormat`, a `MapResultFormat` or an `ObjectResultFormat`. The basic result elements may be formatted as references to software entities (`SoftwareEntityReference`), basic values like `StringFormat` and `NumberFormat`, or any cascaded analysis results. Every format definition can be annotated with optional descriptions for contained formats so that the semantics of elements are preserved.

This model enables process participants to publish *Maps* where the keys are *Strings* (e.g. variable names) and the values are *Lists* of references to concrete software statements (e.g. potential allocation sites), thus creating a very common format for points-to information. By distinguishing references to software entities from plain strings, this model enables bidirectional links that greatly improve the querying of results.

B. Process

Besides software entities and SPA results themselves, our reuse process involves additional entities that are introduced in Figure 3.

The diagram shows the definition of a `SoftwareEntity`, which always belongs to a single repository and programming language. Also, those entities are of a certain kind (`SoftwareEntityKind`), and may have any number of

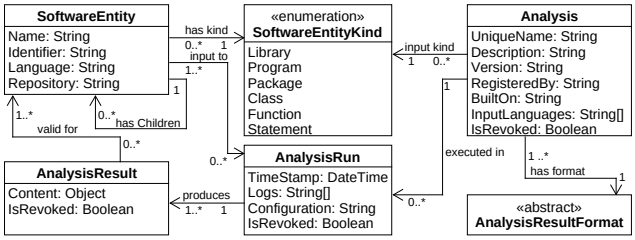

Fig. 3. Core entities of the reuse process

Child-Entities, which typically have a more precise kind than their parents.

An `AnalysisResult` is produced as a result of an `AnalysisRun`. Because an analysis run may for example be executed on a given *Program*, but produce results for each individual *Method*, we distinguish between entities that are input to an analysis run, and those for which the corresponding analysis result is valid. Finally, an `AnalysisRun` is the concrete execution of an `Analysis`. Every analysis has a fixed result format, for which detailed definitions have already been given in Figure 2. As individual analysis implementations evolve over time, they may also be *versioned* via the `Version` attribute.

A number of attributes in Figure 3 are defined so that process participants can judge the context and quality of SPA results, and enhance the reproducibility of results:

- Every analysis has a `Description`, information about the technology it is built upon (`BuiltOn`) as well as a reference to the process participant registering it.
- Analysis runs provide information about the concrete configuration that was used to instantiate the analysis, as well as the logs produced during execution.
- Results, analysis runs or entire analyses may be *revoked* if the corresponding implementation or configuration turns out to be incorrect. Revoking an analysis run automatically revokes all results associated with it, the same is true for analyses and their analysis runs.

Furthermore, the fact that analyses (and results transitively) are attributed to a user acts as an *incentive* for publishing results. It allows process participants to reference their SPA results online and encourages reuse in other research projects.

Based on our definitions we identify four necessary process steps, which are described in the following sections.

1) Acquiring Software Entity Information: Since analysis results are linked to software entities in multiple ways, one first needs to gather information about those entities before any results can be published. This can be seen as an *indexing phase*, where information about target programs, classes, methods, and statements is acquired and persistently stored. This information may either be computed *ad-hoc* for a given set of software components (e.g., all contents of a repository), or *on-demand* whenever an analysis run is about to process software.

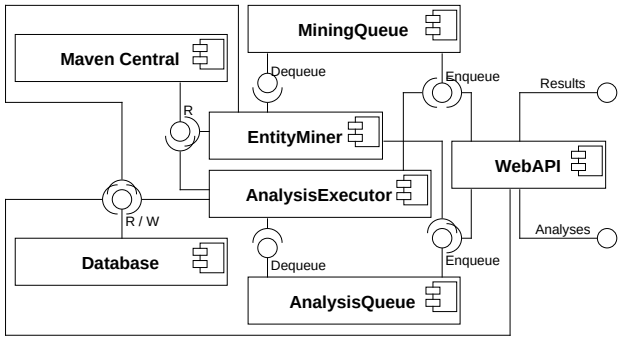

Fig. 4. Components of our prototype implementation

2) Registering Analyses: When process participants want to publish results, they first need to register corresponding *analysis information* in a global registry. As mentioned before, this is necessary so that other participants can judge the context in which subsequent results have been generated, as well as potential limitations or threats to validity. This also defines the format of results that are produced by the analysis.

3) Producing Results: Once entity information and analysis data are present, SPA results can be published and stored in a database. As seen in Figure 3, this also includes information about the analysis run in which the results have been produced.

4) Consuming Results: All process participants may retrieve information on SPA results or software entities from the database. The definitions in Figure 3 enable us to query results in different ways: Based on their respective analysis, the analysis run with created them, or based on the entities that the results are valid for. This makes it possible to for example retrieve all SPA results associated with program methods that have a specific name.

IV. Implementation

We implement *SPARRI*, a prototype for our design presented in Section III using *Scala*. Our implementation is available on GitHub[2]. For this prototype, we restrict ourselves to processing software written for the JVM and published via *Maven Central*, although the process and data model work with arbitrary software artifacts and repositories. We further add the concept of *built-in* analyses to SPARRI. This allows for certain types of results (including callgraphs and dependencies) to be calculated on-demand without having to implement an analysis first.

Figure 4 provides an UML component diagram highlighting the architecture of SPARRI. Most notably, we employ *Message Queues* to buffer different kinds of system tasks and decouple components from each other.

The interface for interacting with our system prototype is realized in the *WebAPI* component. It hosts a RESTful HTTP

[2]For review purposes we provide an anonymized version here: https://anonymous.4open.science/r/sparri-artifact-3C98

API for accessing all system functionalities. Specifically, the API can be used for:

- **Registering Analyses**: Users can register information for new analyses at the API. All data is directly written to the underlying database. This implements a mandatory step of our reuse process, as described in Section III-B2.
- **Publishing and Accessing Results**: Analysis results for a given set of software entities can be posted to and retrieved from the WebAPI component via the JSON data format. Similar to registering analyses, this is implemented as a direct database write or read, respectively.
- **Triggering Analysis Runs**: Our prototype provides a number of built-in analyses, which can be triggered for arbitrary input entities via the WebAPI. If those entities are not yet known to the system, a `MiningTask` is created and inserted into the *MiningQueue*. If the entities are known, a `AnalysisTask` is created and inserted into the *AnalysisQueue*. Both queue inserts are done with *high priority*, as they are the result of a user request and should therefore be prioritized over automatically generated background tasks.

The *EntityMiner* processes `MiningTasks` from the corresponding queue and collects data on the requested entities. This is done by querying the HTTP interface of Maven Central. For each Maven program, we collect data on packages, classes, methods, and statements, which we then store in the database. To do this, we download the corresponding JAR file, process it using the OPAL framework [23] and then discard it afterward. Optionally, the `MiningTask` can specify a built-in analysis to trigger once mining is completed. In this case, upon completion, the EntityMiner queues a corresponding `AnalysisTask` at the *AnalysisQueue*.

`AnalysisTasks` are consumed by the *AnalysisExecutor* component. This component is responsible for executing *built-in* analyses on-demand. Before an analysis is executed, the component validates a number of prerequisites. In particular, it checks that the supplied input entities are of the correct kind, that the exact same run has not been executed before (otherwise it would be redundant), and that data on the input entities has been mined beforehand. If the last check fails, a corresponding `MiningTask` is queued, thus effectively postponing analysis execution until the required data is present. Currently, SPARRI comprises three built-in analyses:

- **Dependencies**: An analysis implementation that extracts all direct dependencies for entities of kind `Program`, i.e. Maven software artifacts. It can be configured to also extract transitive dependencies.
- **Change Frequency**: For a given entity of kind `Library` (i.e. *G-A-Tuples* in Maven), this analysis calculates the change frequency of all classes contained in any library release. The result is a mapping of *Fully-Qualified-Classnames* to tuples (n_δ, n), where n_δ represents the number of unique binary occurrences of the class, and n the total number of occurrences.
- **Callgraphs** This analysis creates partial callgraphs for entities of kind *Program*. To do so, it downloads JAR files from Maven Central, loads them in the OPAL analysis framework, and builds a callgraph using the XTA construction algorithm. The analysis can be configured to use different construction algorithms (e.g., *CHA*, *RTA* or others) and can optionally also load implementations of the Java Runtime Environment when building the callgraph.

Once execution succeeded, the component creates corresponding objects of type *AnalysisResult* and *AnalysisRun* (see Figure 3) in the database.

A. Result Contextualization

SPARRI allows analysis result formats to be annotated with textual descriptions for each individual part of the specification. This can help in understanding the output of an analysis and putting it in the right context before reusing results. To support this, our implementation features an API endpoint that generates a full-text description of result formats based on these individual annotations. Listing 1 shows the description generated for the result format of our built-in *Dependency Analysis*.

```
A List of elements that represent Dependencies (GAV-
    Triple) for this program. The elements are
    formatted as:
 An object containing 2 different properties:
  A property with name identifier, that contains The
      GAV-Triple identifying a dependency, i.e. a
     required Maven library. The property is
     formatted as:
   An object containing 3 different properties:
    A property with name groupId. The property is
       formatted as: A text value
    A property with name artifactId. The property is
        formatted as: A text value
    A property with name version. The property is
       formatted as: A text value
  A property with name scope, that contains The
     Maven scope of this dependency. The property
     is formatted as: A text value
```

Listing 1. Generated description for Maven dependency result format

B. Foundations

For some aspects of SPARRI we rely on existing tools and frameworks. In particular, we use *RabbitMQ* for implementing Message Queues with priorities, OPAL for inspecting bytecode and for executing analyses, as well as *Maven Archeologist*[3] and *Jeka*[4] for resolving dependencies. Furthermore, both the `EntityMiner` and `AnalysisExecutor` make use of *Akka Streams*[5] for the continuous stream processing of tasks, while the WebAPI makes use of *Akka HTTP*[6] to host its HTTP server. Our database is implemented using a predefined *PostgreSQL* Docker image[7]. All individual components are deployed using Docker as well.

[3] github.com/square/maven-archeologist
[4] github.com/jeka-dev/jeka
[5] github.com/akka/akka
[6] github.com/akka/akka-http
[7] hub.docker.com/_/postgres/

V. Evaluation

To evaluate our work, we pose the following research questions:

- Can compositional analysis with SPARRI provide performance benefits over whole-program analysis? (**RQ2.1**)
- Does an infrastructure like SPARRI aid in conducting large-scale empirical research on software? (**RQ2.2**)
- Is our data model for result formats flexible enough to support real-world format definitions? (**RQ2.3**)

In the following sections, we answer those questions in detail, each time presenting the respective methodology and results. All experiments have been conducted on a server with an *Intel Xeon E5-2650* quad-core CPU and 34GB of RAM.

A. RQ2.1: Performance Benefits

We ask whether building compositional static analyses that rely on partial or intermediate results can result in performance benefits when compared to "traditional" whole-program analysis. Recent research indicates that this is indeed the case not only in theory but in practice [15], [57], with modular constructions being up to 95% faster when partial results are available [57].

To verify these results for our own design and prototype, we compare execution times for whole-program analyses with those of compositional ones for two problems of different complexity. Each time, we record the average execution time of whole-program analysis runs, as well as the initial and amortized runtime for compositional analyses. We then analyze if and under which assumptions a compositional implementation yields benefits.

For the first analysis, we chose the relatively simple problem domain of calculating the transitive closure of all dependencies for a given program. As a second problem domain, we choose callgraph construction, as it is a very popular example of static program analysis.

1) Transitive Dependencies: We implement two analyses that both calculate the set of all transitive dependencies for a given program identifier (GAV-Triple) in Maven Central:

- The `SimpleTransitiveDependencyAnalysis` is a straightforward whole-program analysis that for a given program collects all transitive dependencies using a library called *Jeka*[8], which parses all `pom.xml` files along the dependency tree.
- The `ReusedBasedTransitiveDependencyAnalysis` relies on SPARRI's API to retrieve partial results - i.e. the set of all *direct dependencies* - for every program along the dependency tree.

When running our compositional analysis implementation, we assume that all required partial results have in fact been precomputed. To ensure a fair comparison, each evaluation run starts with an empty database. We then trigger a computation of all required partial results and record the time it takes SPARRI to make all results available. This initial computation time $t_{R,0}$ is crucial for computing amortized execution times.

[8]https://github.com/jeka-dev/jeka

TABLE II
Execution times for dependency computations

	$\#_{DEPS}$	t_{WP} [ms]	$t_{R,0}$ [ms]	t_R [ms]	$\overline{t_R}$ [ms]	N_B
P1	6	1318	29120	97	494	4
P2	5	1001		86	417	4
P3	1	324		31	97	3
P4	32	5917		439	2557	4
All	44	8561	29120	654	3566	4

We evaluate the performance of both analysis implementations on a small benchmark of four programs listed on Maven Central. To create the benchmark, we started at popular Maven libraries[9] and randomly searched through the list of transitive dependees, to find programs with different dependency counts. The rationale behind this is that the number of dependencies corresponds to the number of partial results that need to be created and consumed, thus allowing us to judge whether this affects the overall performance. For each program, both analyses are repeated ten times. Each time, we check that the compositional and the whole-program approach yield the same results.

Table II reports our key results, namely the average whole-program execution time t_{WP}, the initial time for precomputations $t_{R,0}$ and average execution time for the compositional analysis t_R. Finally, $\overline{t_R}$ captures the amortized runtime for ten repetitions of the compositional analysis, and N_B the point of *Break-Even*, i.e. the number of runs after which the compositional approach is faster (on average) than the whole-program analysis.

We can see that for all individual programs, as well as overall, the compositional approach is faster than the whole-program computations, with raw execution times being reduced by over 92%. However, the compositional analysis requires initial computations that took a total of 29 seconds. Taking this into account, it takes four runs (N_B) for the compositional analysis to be faster - on average. After ten runs ($\overline{t_R}$), the runtime for compositional analyses is already reduced by over 58% compared to whole-program analyses.

2) Callgraphs: We implement two different approaches to calculating callgraphs using the Rapid Type Analysis (RTA) algorithm. `DirectCallgraphAnalysis` is a whole-program analysis that leverages bytecode parsing and the implementation of RTA from the OPAL framework [23]. To do so, it first downloads JAR files for all dependencies. In contrast, the `ReuseBasedCallgraphAnalysis` uses partial callgraphs registered at SPARRI and stitches them together.

As before, we start with an empty database and record the initial computation time for partial results $t_{R,0}$. Furthermore, we capture the execution times for the whole-program approach (t_{WP}) and the compositional analysis (t_R), each time averaging over five runs. We execute our analysis on three programs with a varying number of dependencies.

[9]https://mvnrepository.com/open-source

TABLE III
EXECUTION TIMES FOR CALLGRAPH COMPUTATIONS

	$\#_{DEPS}$	t_{WP} [ms]	$t_{R,0}$ [ms]	t_R [ms]	$\overline{t_R}$ [ms]	N_B
P1	5	9528	27283	3661	4479	2
P2	12	5861		3160	4933	7
P3	0	245		31	167	7
All	17	15634	27283	6851	9579	4

Table III presents the key results for the callgraph performance evaluation. Similar to our observations for dependencies, we see that the raw execution times are smaller for the compositional approach, on average they are reduced by 56%. Depending on the size of the program's dependency set, break-even is achieved after two to seven runs, with initial computations taking over 27 seconds in total.

B. RQ2.2: Aid to Empirical Research

Since our design essentially provides a large-scale index for arbitrary analysis results with built-in facilities to reason about libraries, programs, and other entities, we see an opportunity for using SPARRI to simplify large-scale empirical studies on software. To validate this claim, we show how existing empirical studies on Java software can be replicated using SPARRI, without having to collect any software artifacts or process any bytecode.

In the following, we present three empirical studies on software artifacts by other researchers and explain how we re-implemented their respective methodologies using our prototype implementation. Our implementations for those studies can be found in our GitHub repository.

1) API Breaking Change Analysis: In their 2016 study titled *"Semantic versioning and impact of breaking changes in the Maven repository"* [58], Raemaekers et al. investigate what kind of changes are introduced in new releases of a software component and how they are related to semantic versioning. The study uses a benchmark of over 100,000 JAR files to collect change information, which is computed on JVM bytecode using a tool called *Clirr*. It detects a total of 20 breaking change types, 13 of which can immediately be computed using the information collected by SPARRI:

Method removed (MR), Class removed (MR), Parameter Type Changed (PTC), Return Type Changed (MRC), Interface Removed (IR), Number of Arguments Changed (NPC), Method added to Interface (MAI), Removed from Superclasses, Method Accessibility Decreased, Method now Final, Abstract Method Added, Added final Modifier, Added abstract Modifier

The remaining seven change types operate on *Fields*, which are currently not reflected in our prototype implementation. However, adding them is a straightforward operation within the boundaries of our current data model (cf. Figure 3).

To validate our claim, we implement a `BreakingChangeAnalysis` in SPARRI. Given a Maven library identifier (GA-Tuple) and two releases of that library, it computes the number of occurrences for each of the thirteen change types using only SPARRI's API. We note that in this case, there is no need to pre-compute any partial results, as all computations can be done strictly on the entity structure of the two programs.

2) Dependency Graph Generation: In 2019, Benelallam et al. introduce the *Maven Miner* [59], a tool that builds a dependency graph for the entirety of Maven Central. It iterates over all programs (GAV-Triples), each time extracting dependencies from the respective `pom.xml` files, and stores the resulting graph in a *Neo4j* graph database for subsequent analyses.

We claim that the same result can be achieved using SPARRI, without having to explicitly download and parse any files. This requires *direct dependencies* - a type of partial result for which we provide built-in analyses - to be pre-computed for every artifact in Maven Central.

To verify this claim, we implement the `DependencyGraphGenerator`. It builds a full dependency graph for Maven Central and writes it to a dedicated Neo4j Graph Database. As in the original study, nodes are GAV-Triples and edges represent dependencies. This is done by only querying the SPARRI API, no calls to Maven Central itself are necessary. The following HTTP request is used to retrieve all Maven programs (GAV-Triples):

```
GET /entities?kind=Program&language=Java
```

For each such program, the direct dependencies are retrieved by invoking the following HTTP request:

```
GET /entities/<GAV>/results?analysis=mvn-
                dependencies:1.0.0
```

The required information, namely the dependency GAVs, can be directly extracted from the resulting JSON response. Since no XML parsing and dependency resolution is required, the overall implementation of the `DependencyGraphGenerator` requires less than 210 lines of code.

3) Library Evolution Metrics: Constantinou and Stamelos developed a set of software metrics that describe the stability and evolution of an evolving software system [60]. They do this by computing numeric values - named *Stability* and *Evolution* - based on two subsequent releases of a library. The authors evaluate their metrics by conducting an empirical study on five real-world software systems with a total of 220 versions.

The metrics are computed using a simplified structural model of the target software. Specifically, all metrics are computed using only the sets of packages and classes, as well as usage relations between classes. As specified by the study authors, any direct mention of one class by another is considered a *usage*, including subtyping, invocations, and field accesses.

We note that SPARRI would also allow us to compute *potential usages* based on an actual callgraph, instead of relying on the declared type of invocations and field accesses. This would merely require us to compute partial callgraphs beforehand, as illustrated for our performance evaluations in

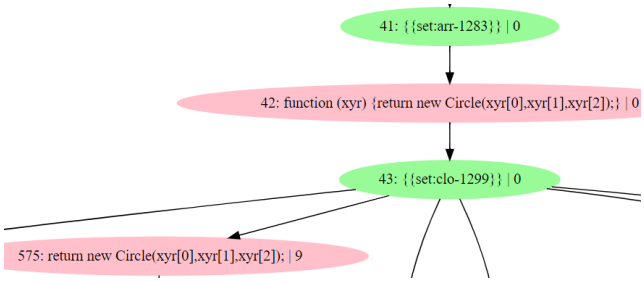

Fig. 5. Rendering of a JIPDA flow graph [61]

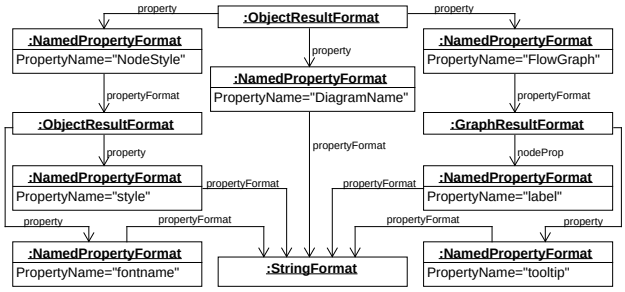

Fig. 6. Representing JIPDA flow graphs in our prototype

Section V-A. However, since this was not done in the original study, we stick to structurally derived usages here.

```
def calculateEvolutionMetrics(ga: String,v1: String,
  v2: String): Try[EvolutionMetricsReport]
```

Listing 2. Interface of the `EvolutionAnalyzer`

We implement a Scala class called `EvolutionAnalyzer` that computes the Stability and Evolution metrics for two releases of a given Maven library. Listing 2 shows its interface, requiring only a GA-Tuple to identify the library and two subsequent versions `v1` and `v2`. For both resulting GAV-Triples, we retrieve the entire program entity structure (packages, classes, methods, and instructions) using a single call to the SPARRI API: `GET /entities/<GAV>` Based on this data, we compute the sets of packages (P_i and P_{i+1}) and usage relations ($REL_i \in P_i \times P_i$ and $REL_{i+1} \in P_{i+1} \times P_{i+1}$) as specified by Constantinou and Stamelos. In the final step, we use those sets to compute the metric values for Stability and Evolution, returning them in a container class called `EvolutionMetricsReport`.

C. RQ2.3: Flexible Data Model

Figure 2 presented the data model we propose for describing the formats of analysis results. To evaluate its flexibility, we ask whether this model can be used in real-world SPA applications and, therefore, is an appropriate choice for our use case. We evaluate this question by providing example mappings of real-world analysis result formats to our own format. The code-based format definitions can be found in our artifact.

First, we investigate JIPDA flow graphs used by Nicolay et al. to detect vulnerabilities in JavaScript applications [10]. Such graphs are exported as a *dot graph*, an example of which can be seen in Figure 5.

As seen here, graph nodes contain labels (and additionally tooltips), as well as a unique integer id. Edges are defined as tuples of source- and target node ids, without additional properties. Additionally, the graph may have a name and node styling properties. Figure 6 shows how this format is replicated using our data model (as seen in Figure 2).

A similar mapping (also relying on the `GraphResultFormat`) can be used to describe *Static Summary Trees* as used by Chen et al. [31] for their Thread Escape Analysis. This only requires an additional node property called `Type`, in order to differentiate node types like *Read*, *Write*, *Controlflow*, and others.

The *auxiliary points-to sets* used by Hardekopf and Lin to perform flow-sensitive pointer analysis [47] can be represented in our format using Maps where keys are variable names (strings) and values are sets of program locations. Those locations can be represented as either text (e.g. statement content), numbers (program counters), or composite objects containing additional information.

VI. Discussion

Our study on state-of-the-art applications of SPA showed that reusing existing or pre-computed results is becoming increasingly popular among analysis developers. However, this reuse happens almost exclusively in the context of one research project or application, not across projects or domains. We argue that many analyses (and empirical studies) would benefit from persisting, reusing, and sharing such SPA results, but as of now lack the means to do so. Thus we presented SPARRI, a prototype implementation of our result reuse process.

Our evaluation showed that compositional static analyses based on SPARRI can in fact yield performance benefits compared to a whole-program approach. This is in line with the findings of other researchers on reuse in specific domains [15], [57]. We see that the initial overhead of computing partial results can often be compensated after only four analysis runs, in one case even after two runs. Amortized execution times for ten runs can be less than half of those for whole-program analyses. Given that SPARRI aims to persist partial results indefinitely for an entire repository, initial computation times can almost be neglected in the long run, which would yield a net reduction in execution time of almost 90% for dependencies, and 56% for callgraphs.

However, it is important to note that our evaluation only confirms that speedups *are possible*, we cannot assume that this generalizes to all programs or other analysis domains (aside from dependency analysis and callgraphs). Some program analyses are inherently non-compositional, and thus can not be expected to benefit from persisting and reusing results.

We further find that a system like SPARRI vastly simplifies empirical studies on software. Our evaluation illustrated that it is possible to replicate existing studies by relying only on

SPARRI's HTTP API, without any frameworks for processing JVM bytecode or accessing Maven Central. This common abstraction could be especially useful for comparative studies between different repositories and programming languages, which Decan et al. [62] identified as necessary for understanding structural differences in repositories.

The result format we propose (by construction) covers 83% of the top-level data structures we saw in state-of-the-art implementations. Our evaluation showed that it can be used to capture multiple result formats from real-world SPA implementations. We aim to improve result format comprehension by allowing descriptive annotations, which consumers can use to better understand the syntax and semantics of existing results (cf. Listing 1).

There are additional benefits to indexing and reusing analysis results, which we think will be especially beneficial to the software engineering research community. A system like SPARRI can be used to share results in the context of artifact evaluations for research artifacts. In particular, this would allow reviewers to access results and judge their quality while allowing researchers to understand how and why their results are used in other research contexts. Additionally, a large-scale index of code entities (like SPARRI) facilitates the creation of domain-specific software benchmarks, where researchers can query and filter for distinct features that they would like to be represented in their benchmark.

With our work, we have shown that persisting and reusing the results of static program analyses - like the reuse of software itself - can yield benefits in multiple dimensions. This includes the design and implementation of new analyses, the runtime complexity of such analyses, as well as other research areas like empirical studies on software and artifact evaluations. A downside to sharing results is the organizational and developmental overhead it introduces. With SPARRI, we provide a system that addresses those issues, thus reducing the overhead for publishing SPA results.

A. Adopting SPARRI

Based on our evaluation results, we anticipate that SPARRI can yield significant benefits when adopted by SPA implementations. We foresee two distinct aspects that analysis designers may want to address:

1.) Analyses may utilize SPARRI to publish their results and make them accessible to other analyses, researchers, or developers. This can be done for both existing and new analysis implementations with relatively low effort. It requires an additional (potentially optional) post-processing step, which converts the analysis results to our result format and uploads them.

2.) Analyses may use existing results provided via SPARRI to build on top of them. This is a fundamental design choice, and can only be achieved for existing analysis implementations if they already rely on precomputed or partial results. In this case, a pre-processing step that retrieves those results and converts them to the format expected by the analysis needs to be implemented. For new implementations, analysis designers could directly rely on our format when consuming their inputs.

Both aspects can be adopted independently, meaning that analyses may publish to SPARRI, consume from SPARRI, or both at the same time. In the future, we plan to provide a client-side API that reduces the implementation effort for converting proprietary result formats to our format and vice versa.

B. Limitations and Threats to Validity

Internal Validity. For *RQ2.2*, we faithfully re-implemented tools for empirical studies based on their publications. We did not reproduce those studies to the original extent, as our goal was to prove that all required input data can be provided via SPARRI. Furthermore, SPARRI is a prototype implementation missing features like authentication and authorization. While implementing those features may introduce a slight overhead in performance, we are confident that this does not influence our observations for *RQ2.1* in any significant way.

External Validity. While we showed that some analyses can benefit from reusing results, this is by no means guaranteed for all inputs to those analyses or other types of analysis. It is to be expected that some non-distributive analysis problems either cannot reuse previous results at all, or do not benefit from doing so. On the other hand, previous research has shown that there can be performance benefits even in non-distributive cases [16]. We further note that studies requiring results that cannot be provided by SPARRI's built-in analyses can only benefit from using our process if others provided the required data beforehand.

VII. Related Work

To the best of our knowledge, only a few publications have dealt with the topic of reusing analysis results, none of which provide a general approach. Therefore, we present related work that focuses on partial aspects similar to our design.

A. Large-Scale Software Analysis

In 2021, Maj et al. presented *CodeDJ*, an infrastructure for querying git repositories for content and metadata [63]. The system consists of a persistent datastore and an in-memory database offering a Rust-based query interface. It uses a language-agnostic data model to store information about projects, commits, users, and metadata, as well as file contents. CodeDJ emphasizes reproducibility by allowing queries on older states of the data store. Similar to our work, Maj et al. capture an extensive amount of project-specific metadata. However, unlike SPARRI, their system does not investigate the source code itself and thus offers no facilities to query for entities like classes, methods, or instructions.

Dyer et al. propose *Boa*, a system that enables researchers to query software projects semantically [64]. The authors provide infrastructure to parse Java files and conduct parallel queries on a large data set generated from SVN repositories. Similar to SPARRI, Boa provides built-in static analyses that can extract graph-based representations of individual Java Methods, including control-flow graphs, program-dependence graphs, and

others [65]. However, Boa does not handle any non-graph-based or inter-method results, and cannot be extended with data from analyses that are not built-in by the developers.

GitHub Code Search allows users of GitHub to search for symbol definitions in software, including class- and method-definitions [66]. Similar to SPARRI and its indexing feature, this enables users to conduct studies on code and built domain-specific benchmarks based on semantic filtering criteria.

Upadhyaya and Rajan present a different approach to reducing the runtime for large-scale program analysis [67]. They cluster programs based on their *"analysis specific similarity"* so that it is sufficient to analyze one candidate to produce valid results for the entire cluster. The authors rely on Boa to implement their analyses and observe speed-ups of up to 69% compared to their baseline approach.

Pauck and Wehrheim combine existing Android taint analyses into a single framework called *CoDiDroid* [68]. They show that combining the respective results of those individual analyses on-demand yields benefits in terms of precision and scalability for a given analysis problem.

B. SARIF

The OASIS *Static Analysis Results Interchange Format* (SARIF) defines a format for persisting the results of static analysis tools that assess the quality of a program [22]. One goal is to provide *"an overall picture of program quality"* [22, p.16] by vastly simplifying the exchange and aggregation of results of an individual qualitative aspect.

One central concept of SARIF are *rules*. Analysis tools define a set of rules that they can validate, and an *analysis run* produces results for a given program, which consist of a number of rule violations. The SARIF specification requires that *"each result is produced by the evaluation of a rule"*, meaning that all results semantically correspond to the violation of a software quality rule or guideline.

```
"results": [{
  "ruleId": "C2001",
  "ruleIndex": 0,
  "message": { "id": "default"  },
  "locations": [{
   "physicalLocation": {
      "artifactLocation": {
        "uri": "src/collections/list.cpp",
      "uriBaseId": "SRCROOT"
    }
   },
   "logicalLocations": [
   {"fullyQualifiedName":"collections::list::add"}
   ]
  }]
}]
```

Listing 3. Sample `results` definition in SARIF [22]

Listing 3 shows an example of a rule violation against a rule named `C2001` in SARIF. It highlights the amount of additional information that can be attached to a finding, including a physical and logical location, as well as messages.

We choose to not use SARIF as our data format because the conceptual goal of our design is not to store violations of some rules or guidelines, but also general properties (e.g. callgraphs) of a given program. These properties are not derived by checking whether a given rule holds and are also not directly connected to some aspect of a program's quality. In fact, while SARIF aims to provide a broad range of information to program developers directly, our goal is to make general program properties reusable for other analyses as well.

However, our format definition (see Figure 2) allows us to also represent findings as specified in SARIF, where we can use `SoftwareEntityReference` objects to model `logicalLocations`, and general `ObjectResultFormats` to represent further information about the physical location of a violation. It would therefore be feasible to import results specified via SARIF into SPARRI. On top of that, it would be possible to represent the entire SARIF specification using our format definitions, thus allowing not only the direct upload of SARIF files but also their download when accessing the results.

Using SARIF has some advantages over our data format. It is an industry-backed, fully standardized format that is supported by some existing analysis implementations, as well as a variety of IDE integrations and GUI tools. SARIF also supports localization for multiple languages, as well as hierarchical views and taxonomies for rule violations. However, its main goal is to provide program developers with a way of managing, aggregating, and inspecting violations of software quality rules. With our work, we provide a less restrictive result format, where any computed property of a program can be stored, inspected and ultimately be used in other analysis implementations. This way we open the possibility to reuse intermediary results.

VIII. Conclusion

For this work, we evaluated 40 state-of-the-art SPA implementations. We analyzed if and how existing results are reused, what new results are produced, and how they are persisted. We found that while most analyses reuse some form of partial result, this rarely happens across tools or research projects (5% of publications). Although many analyses produce general results, only 13% of publications talk about persisting results in any form.

Motivated by those findings, we proposed a structured process for publishing, indexing, and reusing SPA results. We implemented SPARRI, a prototype supporting our reuse process for Maven Central. SPARRI is a distributed application featuring multiple built-in SPA implementations that can be triggered on-demand, as well as an HTTP API for accessing analysis results and software information.

Our evaluation of SPARRI illustrated that reusing SPA results can result in significant runtime reductions, in some cases over 92%. We further found that other research areas also benefit from persisting analysis results. We replicated multiple empirical studies on software using only SPARRI's HTTP API, thus drastically reducing implementation complexity. We further hypothesize that tasks like benchmark creation and artifact evaluation can be supported using SPARRI.

References

[1] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 1994.

[2] Sonatype, "Statistics for the central repository," https://search.maven.org/stats, 2023, accessed: 24.04.2023.

[3] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch, "The race to the vulnerable: Measuring the log4j shell incident," 2022.

[4] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Top Productivity through Software Reuse*, K. Schmid, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 207–222.

[5] H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes, "Is Popularity a Measure of Quality? An Analysis of Maven Components," ser. ICSME '14. USA: IEEE Computer Society, 2014, p. 231–240. [Online]. Available: https://doi.org/10.1109/ICSME.2014.45

[6] A. Ikegami, R. G. Kula, B. Chinthanet, V. Maeprasart, A. Ouni, T. Ishio, and K. Matsumoto, "On the Use of Refactoring in Security Vulnerability Fixes: An Exploratory Study on Maven Libraries," in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, ser. EASE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 288–293. [Online]. Available: https://doi.org/10.1145/3530019.3535304

[7] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, p. 102, May 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10071-9

[8] T. Litzenberger, J. Düsing, and B. Hermann, "Dgmf: Fast generation of comparable, updatable dependency graphs for software repositories," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.

[9] W. Koch, A. Chaabane, M. Egele, W. Robertson, and E. Kirda, "Semi-Automated Discovery of Server-Based Information Oversharing Vulnerabilities in Android Applications," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 147–157. [Online]. Available: https://doi.org/10.1145/3092703.3092708

[10] J. Nicolay, V. Spruyt, and C. De Roover, "Static Detection of User-Specified Security Vulnerabilities in Client-Side JavaScript," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–13. [Online]. Available: https://doi.org/10.1145/2993600.2993612

[11] Y. Cai, P. Yao, and C. Zhang, "Canary: Practical static detection of inter-thread value-flow bugs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1126–1140. [Online]. Available: https://doi.org/10.1145/3453483.3454099

[12] A. Taherkhani, A. Korhonen, and L. Malmi, "Recognizing Algorithms Using Language Constructs, Software Metrics and Roles of Variables: An Experiment with Sorting Algorithms," *The Computer Journal*, vol. 54, no. 7, pp. 1049–1066, 05 2010. [Online]. Available: https://doi.org/10.1093/comjnl/bxq049

[13] J. Pande, C. J. Garcia, and D. Pant, "Optimal component selection for component based software development using pliability metric," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 1, p. 1–6, jan 2013. [Online]. Available: https://doi.org/10.1145/2413038.2413044

[14] M.-C. Lee, "Software quality factors and software quality metrics to enhance software quality assurance," *British Journal of Applied Science & Technology*, vol. 4, no. 21, pp. 3069–3095, 2014.

[15] W. Zhang and B. G. Ryder, "Automatic construction of accurate application call graph with library call abstraction for java," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 4, pp. 231–252, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.351

[16] P. D. Schubert, B. Hermann, and E. Bodden, "Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Møller and M. Sridharan, Eds., vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14045

[17] S. Arzt and E. Bodden, "Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes," ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 288–298. [Online]. Available: https://doi.org/10.1145/2568225.2568243

[18] T. Szabó, S. Erdweg, and G. Bergmann, "Incremental whole-program analysis in datalog with lattices," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–15. [Online]. Available: https://doi.org/10.1145/3453483.3454026

[19] B. Liu and J. Huang, "Sharp: Fast incremental context-sensitive pointer analysis for java," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527332

[20] J. Van der Plas, Q. Stiévenart, N. Van Es, and C. De Roover, "Incremental flow analysis through computational dependency reification," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 25–36.

[21] B. Stein, B.-Y. E. Chang, and M. Sridharan, "Demanded abstract interpretation," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 282–295. [Online]. Available: https://doi.org/10.1145/3453483.3454044

[22] OASIS Open, "SARIF Version 2.1.0," https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.pdf, 2020, accessed: 05.05.2023.

[23] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in opal," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–196. [Online]. Available: https://doi.org/10.1145/3368089.3409765

[24] J. R. Wood and L. E. Wood, "Card sorting: Current practices and beyond," *J. Usability Studies*, vol. 4, no. 1, p. 1–6, nov 2008.

[25] Z. Stapic, E. García, A. García-Cabot, C. De, M. Ortega, and V. Strahonja, "Performing systematic literature review in software engineering," 07 2022.

[26] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, 2007, software Performance. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412120600197X

[27] A. Mishra, A. Kanade, and Y. N. Srikant, "Asynchrony-aware static analysis of Android applications," in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2016, pp. 163–172.

[28] Y. Jin, H. Wang, T. Yu, X. Tang, T. Hoefler, X. Liu, and J. Zhai, "Scalana: Automating scaling loss detection with graph analysis," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[29] M. Thakur and V. K. Nandivada, "Pye: A framework for precise-yet-efficient just-in-time analyses for java programs," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, jul 2019. [Online]. Available: https://doi.org/10.1145/3337794

[30] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[31] Q. Chen, L. Wang, and Z. Yang, "Heat: An integrated static and dynamic approach for thread escape analysis," in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, vol. 1, 2009, pp. 142–147.

[32] Q. Stiévenart and C. D. Roover, "Compositional information flow analysis for webassembly programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 13–24.

[33] J. Krainz and M. Philippsen, "Diff graphs for a fast incremental pointer analysis," in *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS'17. New York, NY, USA: Association

for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3098572.3098578

[34] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 127–137. [Online]. Available: https://doi.org/10.1145/3324884.3416558

[35] W. Klieber, L. Flynn, W. Snavely, and M. Zheng, "Practical Precise Taint-Flow Static Analysis for Android App Sets," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3230833.3232825

[36] S. Pailoor, X. Wang, H. Shacham, and I. Dillig, "Automated policy synthesis for system call sandboxing," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428203

[37] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 150–157.

[38] F. He and J. Han, "Termination analysis for evolving programs: An incremental approach by reusing certified modules," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428267

[39] J. Liu, Y. Li, T. Tan, and J. Xue, "Reflection analysis for java: Uncovering more reflective targets precisely," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017.

[40] P. Ferrara, A. Cortesi, and F. Spoto, "From cil to java bytecode: Semantics-based translation for static analysis leveraging," *Science of Computer Programming*, vol. 191, p. 102392, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642320300034

[41] N. Allen, P. Krishnan, and B. Scholz, "Combining type-analysis with points-to analysis for analyzing java library source-code," ser. SOAP 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2771284.2771287

[42] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1299–1308. [Online]. Available: https://doi.org/10.1145/3132847.3132954

[43] H. Cai and R. Santelices, "Method-level program dependence abstraction and its application to impact analysis," *Journal of Systems and Software*, vol. 122, pp. 311–326, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216301960

[44] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[45] J. Feichtner, D. Missmann, and R. Spreitzer, "Automated binary analysis on ios: A case study on cryptographic misuse in ios applications," ser. WiSec '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 236–247. [Online]. Available: https://doi.org/10.1145/3212480.3212487

[46] F. Angerer, H. Prähofer, R. Ramler, and F. Grillenberger, "Points-to analysis of iec 61131-3 programs: Implementation and application," in *2013 IEEE 18th Conference on Emerging Technologies and Factory Automation (ETFA)*, 2013, pp. 1–8.

[47] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. 289–298.

[48] A. Sadiq, Y.-F. Li, S. Ling, and I. Ahmed, "Extracting permission-based specifications from a sequential java program," in *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2016, pp. 215–218.

[49] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in SE for Blockchain (WETSEB)*, 2019.

[50] T. Roth, D. Helm, M. Reif, and M. Mezini, "Cifi: Versatile analysis of class and field immutability," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[51] B. Burgstaller, B. Scholz, and J. Blieberger, "A symbolic analysis framework for static analysis of imperative programming languages," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1418–1439, 2012, special Issue: Agile Development. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211002974

[52] L. Nichols, M. Emre, and B. Hardekopf, "Fixpoint Reuse for Incremental JavaScript Analysis," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 2–7. [Online]. Available: https://doi.org/10.1145/3315568.3329964

[53] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," vol. 24, no. 1, oct 2014. [Online]. Available: https://doi.org/10.1145/2629536

[54] A. Rountev and D. Yan, "Static Reference Analysis for GUI Objects in Android Software," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 143–153. [Online]. Available: https://doi.org/10.1145/2544137.2544159

[55] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, mar 2019. [Online]. Available: https://doi.org/10.1145/3293606

[56] J. Nicolay, V. Spruyt, and C. De Roover, "Static Detection of User-Specified Security Vulnerabilities in Client-Side JavaScript," ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 3–13. [Online]. Available: https://doi.org/10.1145/2993600.2993612

[57] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–41. [Online]. Available: https://doi.org/10.1145/3460319.3464836

[58] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300243

[59] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 344–348.

[60] E. Constantinou and I. Stamelos, "Identifying evolution patterns: a metrics-based approach for external library reuse," *Software: Practice and Experience*, vol. 47, no. 7, pp. 1027–1039, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2484

[61] J. Nicolay, "Jipda test: Graph.dot," https://github.com/jensnicolay/jipda/blob/master/test/graph.dot, October 2019, accessed: 04.04.2023.

[62] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks: A comparison of three programming language ecosystems," in *Proccedings of the 10th European Conference on Software Architecture Workshops*, ser. ECSAW '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2993412.3003382

[63] P. Maj, K. Siek, A. Kovalenko, and J. Vitek, "CodeDJ: Reproducible Queries over Large-Scale Software Repositories," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, 2021, pp. 6:1–6:24.

[64] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 422–431.

[65] Iowa State University of Science and Technology, "The Boa Programming Guide - Program Analysis," https://boa.cs.iastate.edu/docs/program-analysis.php, 2019, accessed: 25.07.2023.

[66] GitHub, Inc., "About GitHub Code Search," https://docs.github.com/en/search-github/github-code-search/about-github-code-search, 2023, accessed: 25.07.2023.

[67] G. Upadhyaya and H. Rajan, "Collective program analysis," ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3180155.3180252

[68] F. Pauck and H. Wehrheim, "Together Strong: Cooperative Android App Analysis," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 374–384. [Online]. Available: https://doi.org/10.1145/3338906.3338915