Input Splitting for Cloud-Based Static Application Security Testing Platforms

Maria Christakis
MPI-SWS, Germany

Thomas Cottenier
Amazon Web Services

Antonio Filieri
Amazon Web Services

Linghui Luo
Amazon Web Services

Muhammad Numair Mansur
MPI-SWS, Germany

Lee Pike
Amazon Web Services

Nicolás Rosner
Amazon Web Services

Martin Schäf
Amazon Web Services

Aritra Sengupta
Amazon Web Services

Willem Visser
Amazon Web Services

ABSTRACT

As software development teams adopt DevSecOps practices, application security is increasingly the responsibility of development teams, who are required to set up their own Static Application Security Testing (SAST) infrastructure.

Since development teams often do not have the necessary infrastructure and expertise to set up a custom SAST solution, there is an increased need for cloud-based SAST *platforms* that operate as a service and run a variety of static analyzers. Adding a new static analyzer to a cloud-based SAST platform can be challenging because static analyzers greatly vary in complexity, from linters that scale efficiently to interprocedural dataflow engines that use cubic or even more complex algorithms. Careful manual evaluation is needed to decide whether a new analyzer would slow down the overall response time of the platform or may timeout too often.

We explore the question of whether this can be simplified by splitting the input to the analyzer into partitions and analyzing the partitions independently. Depending on the complexity of the static analyzer, the partition size can be adjusted to curtail the overall response time. We report on an experiment where we run different analysis tools with and without splitting the inputs. The experimental results show that simple splitting strategies can effectively reduce the running time and memory usage per partition without significantly affecting the findings produced by the tool.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

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

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558944

ACM Reference Format:
Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, Muhammad Numair Mansur, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2022. Input Splitting for Cloud-Based Static Application Security Testing Platforms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3558944

1 INTRODUCTION

With the increasing popularity of DevSecOps development practices, Static Application Security Testing (SAST) shifts further to the left in the software development life-cycle and becomes the responsibility of developers rather than security experts. This creates a growing demand for easy-to-use solutions. Many development teams do not have the capacity or expertise to configure and maintain their own static analysis infrastructure and prefer SAST *platforms* that offer a variety of static analyses on demand. Open-source platforms, such as the Software Assurance Marketplace (SWAMP) [30] or ShipShape [49], and their commercial alternatives offer a convenient abstraction. They provide a simple interface through which developers submit code and build artifacts (in their languages of choice) and receive recommendations on how to improve the code. Internally, such cloud-based SAST platforms may employ a variety of static analysis tools, such as [4, 12, 36, 43, 46, 52].

SAST platforms typically are run as a cloud-based service, and the individual analysis tools are containerized and instantiated on-demand on cloud-based machines. Developers expect such a SAST platform to handle inputs (codebases) of arbitrary complexity, and still deliver results within a certain time window. This is especially true for customers that integrate SAST platforms in their continuous integration and deployment (CI/CD) pipelines.

To maintain a predictable response time, SAST platforms face the challenge that they need to be able to scale to different sizes of inputs, and that, every time they add a new analysis tool, they have to ensure that the new tool does not slow down the response time for existing customers.

Vertical scaling by adding more memory or faster machines is not a cost-effective solution to the risk of running out of time or space when analyzing complex inputs. Provisioning machines large

enough to handle the most complex analysis inputs would make the service unnecessarily expensive for customers that analyze smaller and simpler codebases. In the cloud, a large number of small machines is significantly less expensive than a small number of high-performance machines [50]. Moreover, since many SAST tools have superlinear time complexity [36, 44], even the most powerful machine will eventually not suffice. Much research has been conducted on adding various optimizations to improve the scalability of specific analysis engines, such as summarization of method calls [3, 44, 48], caching and reuse of partial results from prior analyses [2, 36], and incremental analysis [13, 55]. However, when operating a SAST platform, modifying the individual tools may not be an option because the tools might be proprietary or maintaining forks with custom modifications may be too costly.

Thus, a horizontal scaling strategy to distribute and balance the analysis load is still needed. Horizontal scaling needs to split up inputs into pieces such that each analysis tool employed by the platform can handle its input within the expected response time. The different pieces can then be analyzed on parallel instances of a given analysis tool. Such a horizontal scaling can be configured per analysis tool, but without modifying the tool itself. More complex tools can be configured to handle smaller pieces of code than lightweight tools to ensure that the overall latency of the platform does not change when a new complex tool gets added.

In this paper, we present an approach to horizontally scale analysis tools in a static analysis platform. Our approach takes as input a program and a bound for the size of code that should be analyzed by each single machine. It then employs a configurable splitting strategy to split the input program into partitions such that the amount of code in each partition is below the provided bound. We evaluate how this splitting process affects the accuracy of different static analysis tools and how the computational cost of analyzing partitions in parallel relates to the cost of analyzing the entire input program.

Splitting code into partitions comes with several challenges. The first challenge is that information may be lost because dependent code fragments are placed in separate partitions. This may impact the precision and recall of static analysis tools. For example, a real defect arising from the interaction between two classes may become a false negative if those classes end up in different partitions. Similarly, the evidence that a vulnerability has been correctly mitigated may become invisible when defect and mitigation split across partitions, yielding a false positive. This leads to our first research question; **RQ1.** What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?

The second challenge when splitting code into partitions is that the complexity of static analysis may not be tied just to the size of the code. For example, data-flow analysis is cubic in the size of data-flow facts that are tracked [45]. That is, if data-flow facts are not evenly distributed across the program, splitting may not reduce the overall time or memory consumption of data-flow analysis if all facts end up in the same partition. Other analysis techniques, such as bi-abduction used by Infer [11], may require a different type of partitioning since their complexity is not tied to data-flow facts. Hence, we cannot guarantee that analyzing a partition uses less time or memory than analyzing the original program. So our second research question is **RQ2.** How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?

A third challenge is to find a splitting strategy that works for different kinds of static analysis tools. Splitting strategies may have different complexities for static analysis tools targeting different languages. E.g., identifying the direct dependencies of a Java class file is roughly constant since it is sufficient to look at the constant pool [9]. In Python, however, one has to iterate over the entire syntax tree of a file to determine its dependencies. A splitting strategy that takes dependencies into consideration is computationally more expensive for Python than for Java. Thus, our third research question is **RQ3.** What kinds of static analysis tools would benefit from splitting strategies discussed in the paper?

To answer these three research questions, we implement a splitting approach that works with two different strategies to create partitions. The first strategy, SizeLimiting, naïvely splits the input program into partitions based on an upper bound S on the number of files (or classes) per partition. Sorted files in lexicographical order are added to a partition until this bound S is reached and then, a new partition is started. The second strategy, SplitMerge, uses dependency information between the files of the input program to create partitions that include the necessary dependencies of a file. In SizeLimiting, all partitions are disjoint, while in SplitMerge, partitions can overlap.

We apply these two splitting strategies to a set of benchmark programs and analyze the resulting partitions with the static analysis tools Rapid [16] and Infer [11]. We evaluate the impact of both splitting strategies over non-splitting on these analysis tools in terms of reported findings and computational performance.

The contributions of this paper are as follows:

- We motivate why input splitting is a relevant problem for SAST platforms and why additional research in this area is required.
- We present experimental results that input splitting can work in practice with different SAST tools.
- We show that with a proper selection of splitting strategy, all evaluated SAST tools can benefit from splitting. Yet finding the right splitting strategy depends on the complexity of the used SAST tool. While tools like Rapid and Infer which perform complex analyses benefit most from dependency-guided-splitting strategy like SplitMerge in terms of reduction in latency, memory consumption and minimizing the loss of findings, for inexpensive linter-like or intra-procedural analyses such as Bandit, a naive strategy like SizeLimiting may be more beneficial.

We do not claim that any of the proposed strategies are optimal, nor that splitting is the only way to increase the maximum tractable problem size. Instead, this evaluation demonstrates how a lightweight splitting strategy can already significantly improve latency, scalability, and cost-effectiveness of cloud-based SAST platforms. For the future, we envision that such strategies can be used to reduce the cost of integrating new static analysis tools into a SAST platform. Moreover, instead of developing and benchmarking explicit splitting strategies for every new tool, a splitting algorithm could be generalized to adjust the splitting strategy based on the number of observed timeouts.

2 MOTIVATING EXAMPLE

We motivate the need for splitting with an example from the OWASP Benchmark[1]. This standard benchmark for Java SAST tools consists of 2,740 test cases for different types of security vulnerabilities. Each test case is a single Java file. The benchmark also has an additional 162 Java files that contain common helper classes which are used by multiple test cases.

The benchmark's public repository also includes score cards that show the performance of different SAST tools, an excerpt of which is displayed in Table 1. For example, the open-source tool FindSecBugs [41] (v1.4.6) analyzes the benchmark in just over two minutes and obtains a score of 39.1% – the OWASP score is based on the precision and recall of a tool's findings, with 100% for finding all and only the (known) vulnerabilities and 0% for only false positives and false negatives. FindSecBugs performs a lightweight analysis based on type propagation and thus scales linearly with the size of the program. For other tools in the benchmark's score cards, we can see that scalability may be an issue. The tools denoted as SAST-01 to SAST-04 in Table 1 have running times ranging from hours to days. Delays of such magnitude might be unacceptable for CI/CD customers.

If we provide a static analysis platform that runs multiple tools as a portfolio, customers would have to wait for the slowest tool to terminate before getting the final results (there are usually post-processing steps, like de-duplication, before the unified results are returned). This makes it harder to add new tools, like SAST-02, to the portfolio. Hence, we would like a mechanism to split the program under analysis into smaller partitions, assuming that the analysis tool that we want to integrate terminates faster on (most) partitions so we can analyze these partitions in parallel and return results without increasing the latency of our analysis platform.

That is, for OWASP, we would like to split the 2,740 test cases into a set of partitions, each of them bounded by some size S that ensures our analysis terminates within an acceptable amount of time. We would also like each partition to contain the subset of the shared 162 classes that are used by any of the tests in that partition. Finally, we would like to minimize the number of partitions, since a very large number of very small partitions would amplify the impact of per-partition overhead, thus decreasing efficiency.

We illustrate the idea of splitting and the different splitting strategies using the listing in Figure 1, a simplified version of the test `BenchmarkTest01025`. The test contains a CWE22 (Path Traversal) vulnerability: the value received from `request.getHeader` is used in a relative pathname without input validation. An attacker could provide an input like `../../etc/passwd` to try to access sensitive data. This test calls helper method `doSomething` in class `Test1`, which is one of the 162 classes that are used by multiple tests. This method is called by a total of 347 tests in the OWASP benchmark, such as `BenchmarkTest01026` and `BenchmarkTest01029`.

[1] https://github.com/OWASP-Benchmark/BenchmarkJava/tree/53878cc8751e348b63de951b91a6d47cf29121d8/

Tool Name	OWASP Score	Total Time
FBwFindSecBugs v1.4.6	39.10%	0:02:02
SonarQube Java Plugin v3.14	33.34%	0:05:30
Commercial SAST-01	16.74%	2:55:20
Commercial SAST-02	30.60%	135:23:38
Commercial SAST-03	24.89%	1:52:00
Commercial SAST-04	32.64%	13:54:20

Table 1: Score (based on precision and recall) and analysis time for several SAST tools on the OWASP Benchmark v1.1. Data taken from the OWASP Benchmark public repository.

```java
public class Thing1 implements ThingInterface {
    @Override
    public String doSomething(String i) {
        String r = i;
        return r;
    }
}

// Simplified version of the OWASP BenchmarkTest 01025
public class BenchmarkTest01025 extends HttpServlet {
    @Override
    public void doPost(HttpServletRequest req,
                       HttpServletResponse response)
                       throws Exception {
        String p = req.getHeader("foo");
        String bar = new Thing1().doSomething(p);
        File fileTarget = new File("./tmp", bar);
        response.getWriter().println("...");
    }
}
```

Figure 1: Simplified version of an OWASP test that uses a shared class. The method *doSomething* is referenced 347 times in different OWASP tests.

Suppose the available compute instances (virtual machines) allow a certain tool to analyze up to 100 files before it risks exceeding the SLA (Service Level Agreement) time limit. This means we must split the OWASP benchmark into a set of partitions, each of them of size at most $S \leq 100$.

Naïve splitting (SizeLimiting). First, we discuss a naïve strategy called SizeLimiting which splits the codebase into non-overlapping subsets of up to S files each. To ensure determinism, the files are sorted in lexicographical order with respect to their names. Splitting is then performed on the sorted files. For the OWASP benchmark, which has 2,740 test classes and 162 shared classes (for a total of 2,902 files), this may produce, for example, 29 partitions of size 100 and one partition of size 2.

Since method `doSomething` in class `Test1` is called by 347 tests, we know these tests will be distributed over at least 4 partitions. That is, all but one of these partitions will not have access to the implementation of `doSomething` when running the static analysis. Depending on the analysis tool and its assumption on missing methods, this may result in a loss of findings, if the analysis under-approximates; or it may lead to false positives, if the analysis over-approximates; or it may crash the tool.

For this example, we need a splitting strategy that is able to create overlapping partitions to reduce the number of unavailable code dependencies in each partition. In the following sections we outline such a strategy, called SplitMerge, and then evaluate its effect on the number of findings compared to the naïve strategy and to not splitting at all. We also evaluate the overhead of computing partitions and possibly reanalyzing code that is shared between partitions.

3 THE SPLITMERGE STRATEGY

We aim to distribute the analysis of a program P, consisting of n files[2] $F = \{f_1, ..., f_n\}$, by splitting the program into partitions $R = \{r_1, \ldots, r_m\}$ (with $m \leq n$) such that each partition r_i contains no more than S files and can be analyzed independently with the target analysis tools. We ensure that the union of all partitions contains all files ($\cup_i r_i = F$). In general, partitions are not required to be disjoint, i.e., the same file may be replicated across multiple ones.

Algorithm overview. SplitMerge consists of three steps. Initially, a partition is created for each file in the codebase, which includes the file itself and its transitive dependencies up to a distance k. The distance k is a parameter of SplitMerge that allows to trade-off the size vs the degree of self-containment of the initial partitions. For the example in Figure 1, for $k = 2$, the initial partitions are {`BenchmarkTest01025`, `Thing1`, `ThingInterface`} and {`Thing1`, `ThingInterface`}.

The second step – *Split* – ensures none of the initial partitions exceeds the maximum size S by splitting any partition exceeding the size limit, while doing its best effort to preserve the dependency relations it contains. This step replicates the nodes with high degree of connectivity in all the split subsets, with the intuition that units with high connectivity are likely to carry semantic information shared by multiple subproblems.

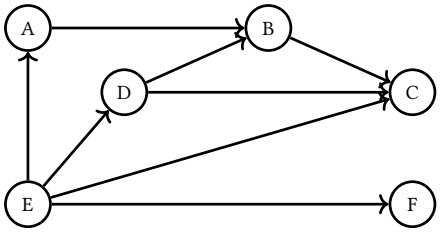
Finally, the third step – *merge* – takes as input a set of partitions of size less or equal than S and performs two tasks: 1) eliminate redundant partitions subsumed by others and 2) merge small partitions into larger ones to balance the load and further increase self-containment. A partition is redundant if it is entirely contained into another. In our example, the partition {`Thing1`, `ThingInterface`} can be dropped since the remaining partitions entirely cover its files and local dependencies. Merging small partitions to maximize the size of their union, constrained by this size being smaller than S, can be framed as a restricted instance of a bin-packing problem [26, 32]. The optimal solution to this problem converges to the smallest number of partitions with approximately uniform size S that cover the input codebase and is expected to balance the analysis load by assigning one partition to each executor.

In the remainder of this section we will detail each step of SplitMerge, with the help of a simplified example.

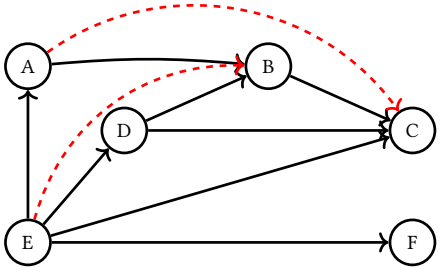
Running example. Consider an example program P containing six files: A, B, C, D, E, F. The dependencies among these files are described in Figure 2a, where a directed edge (x, y) from x to y denotes that x depends on y (symmetrically, that y is a dependency of x). Such dependencies can typically be computed statically in linear time with the size of P, using tools such as JDeps [40] for Java or Snakefood [10] for Python. In the following, we will refer to files and vertices, and dependencies and edges interchangeably via the dependency graph.

Step 1: Initial partitions. This step produces an initial set of partitions of the program P aiming at preserving local dependencies. Given a program P composed of a finite set of files $F = \{f_0, f_1, \ldots\}$ and a neighborhood radius $k > 0$, Alg. 1 constructs for each file a

[2]In this paper we focus on files as the elementary units to partition for analysis, which is a suitable setting for Java and Python. Our splitting strategy can in principle be applied to other language-specific units.



(a) Initial dependency graph of program P.



(b) Dependency graph of P augmented with transitive relations up to radius $k = 2$ (red dashed edges).

Figure 2: Dependency graphs of P.

partition including the file itself and its neighbors up to distance k. A large value for k makes the algorithm more conservative in preserving dependency information. However, it also increases redundancy and the likelihood to produce partitions larger than the size limit S. In Alg. 1, after computing the dependency graph, the first loop augments the dependency relation to include edges linking a vertex to its neighbors up to distance k, while the second loop builds one partition per vertex including it transitive dependencies up to distance k. For a sparse enough dependency graph with n vertices and $k << n$, which is a common situation in practical systems where coupling should be minimized, the algorithm runs in nearly $\Theta(n)$; the worst case complexity would be $O(n^3)$ for $k \approx n$ and a fully connected graph (by reduction to computing the graph transitive closure), although it is unlikely for any realistic program to resemble this situation. The function `computeDependencyGraph` returns the vertices and edges of the dependency graph. Each vertex of the graph corresponds to one file of the program under analysis.

Example. The dependency graph of our example program P is shown in Figure 2a. After the execution of the first loop in Alg. 1 with $k = 2$, the dependency relation is augmented with the transitive dependencies shown in red in Figure 2b – (A, C) and (E, B). The resulting initial partitions are thus:

{A, B, C}, {B, C}, {C}, {D, B, C}, {E, A, B, C, D, F}, {F}

Step 2: Split. Some initial partitions may have size larger than the maximum S. This is especially likely for larger values of the neighborhood radius k. This step aims at splitting an oversized partition r_i into smaller sets that fit within the size limit. However, uniformly splitting r_i into the minimum number or necessary disjoint subsets is likely to delete relevant dependency information. Instead, we deliberately produce a non-minimal number of subsets allowing redundancy to preserve dependency information. In particular, for a partition r_i that exceeds the maximum size ($|r_i| > S$), we sort the vertices in descending degree of connectivity (number

Algorithm 1: Initial partitions.

Input : Files $F = \{f_0, f_1, \dots\}$, neighborhood radius k
Output: Initial partitions $R = \{r_0, r_1, \dots\}$

```
1  (V, E) ← computeDependencyGraph (F)
2  // augment dependency relation
3  for v ∈ V do
4  |  neighbors ← verticesWithinDistance (v, k)
5  |  for n ∈ neighbors do
6  |  |  E ← E ∪ (v, n)
7  |  end for
8  end for
9  // build initial partition
10 R ← ∅
11 for v ∈ V do
12 |  r ← {v}
13 |  for (v, u) ∈ E do
14 |  |  r ← r ∪ {u}
15 |  end for
16 |  R ← R ∪ r
17 end for
18 return R
```

of incoming and outgoing edges) and identify two sets of vertices: *high-connectivity*, which includes the p (a percentage) of vertices with the largest degrees of connectivity, and *low-connectivity* ones, which includes the rest of the vertices. The underlying intuition is that files involved with many dependency chains are likely to be relevant for the analysis of most subsets of r_i. Therefore, Alg. 2 first identifies these two sets and then partitions the low-connectivity vertices uniformly into small enough subsets to allow adding to each such subset the high-connectivity vertices. This operation is formalized in the `split` function, which is applied on each partition whose size exceeds S (line 13 handles the corner case of the chosen p not small enough to ensure splitting all oversized partitions.)

Example. Consider $S = 4$. The partition $\{E, A, B, C, D, F\}$ exceeds this size. In Figure 2b, vertex E has a degree of connectivity 5, B and C have degree 4, A and D have degree 3, F has degree 1. Let $p = 1/3$, E and B are selected as the high-connectivity vertices, leading to new partitions $\{E, B, A, C\}$, $\{E, B, D, F\}$ as replacement of $\{E, A, B, C, D, F\}$ (where vertices with the same degree have been sorted alphabetically).

Step 3: Merge. The last step of SplitMerge reduces the redundancy introduced by the previous steps and computes the final partition (Alg. 3). Some partitions computed by the first two steps may be subsumed by others. For example, $\{B, C\} \subseteq \{A, B, C\}$ in the partitions for our program P. In these situations, the information contained in the larger set subsumes the information in any of its subsets. The subsets can therefore be discarded, without loss of information (first loop in Alg. 3).

The second part of this step aims at grouping together partitions for the sake of balancing the analysis load distribution across multiple executors. This can be framed as an instance of the bin packing problem [32], where a set of items – the partitions – have to fit within the minimum number of bins of size S. While finding the optimal solution is NP-hard, many heuristics have been proposed

Algorithm 2: Split.

Input : Augmented dependency graph $G = (GT, ET)$,
partitions R = $\{r_0, r_1, \dots, r_n\}$,
fraction of high-degree nodes $0 \le$ p < 1,
maximum size S
Output: Split partitions $R' = \{r'_0, r'_1, \dots\ r'_m\}$,
$m \ge n, \forall r' \in R' : |r'| \le$ S

```
1  for r ∈ R do
2  |  if |r| > S then
3  |  |  R ← (R \ {r}) ∪ split (r, p, G, S)
4  |  end if
5  end for
6  return R

7  Function split(r, p, G, S):
8  |  (V_T, E_T) ← extractSubgraph(G, r)
9  |  V_{T_s} ← sortByDegreeDesc (V_T)
10 |  /* the p highest degree nodes are replicated
   |     in each subset                          */
11 |  p_r ← ⌊p · |r|⌋
12 |  if p_r > S then
13 |  |  p_r = ⌊p · S⌋
14 |  end if
15 |  hdn ← [v_{T_s 0}, ..., v_{T_s p_r}]      // high-connectivity
16 |  ldn ← [v_{T_s p_r+1}, ...]               // low-connectivity
17 |  nSubsets ← ⌊(|r| − p_r)/(S − p_r)⌋ + 1
18 |  divide ldn uniformly into nSubsets parts
   |    {ldn_0, ldn_1, ..., ldn_{nSubsets−1}}
19 |  return
   |    {hdn ∪ ldn_0, hdn ∪ ldn_1, ..., hdn ∪ ldn_{nSubsets−1}}
```

to efficiently compute near-optimal solutions [26]. Among these, we adopted *next fit* [5], which has a time complexity of $O(n \log n)$ in the number of partitions n (due to sorting). Although, it may result in up to twice the optimal number of partitions, its fast execution time is preferred for the sake of minimizing the maximum analysis latency. Different algorithms can replace `nextFit` to trade off latency for a smaller number of parallel executors.

Example. In our small-size example, the merge phase would result in the final partitioning already after the redundancy reduction phase, since any further merging by `nextFit` would result in an oversized partition. The final partitions are: $\{D, B, C\}$, $\{E, B, A, C\}$, $\{E, B, D, F\}$.

After the three steps of SplitMerge, the resulting partitions satisfy the desired properties: (1) each partition is smaller than the prescribed size S, i.e., $|r_i| \le S$; (2) the union of the partitions contains all files of the input program, i.e., $\cup_i r_i = F$. In the next section, we introduce our empirical evaluation on the impact of splitting strategies in comparison to non-splitting strategies.

4 EXPERIMENTAL EVALUATION

In this section we report on our experiments using SplitMerge with three analysis tools on a portfolio of Java and Python benchmarks.

Algorithm 3: Merge.

Input : Partitions $R = \{r_0, r_1, \ldots, r_n\}$, maximum size S
Output: Merged partitions $R' = \{r'_0, r'_1, \ldots\ r'_m\},\ m \leq n$

```
for r_i ∈ R do
    if ∃ r_j ∈ R s.t. r_i ⊆ r_j and i ≠ j then
        R ← R \ {r_i}
    end if
end for
return nextFit (R, S)

Function nextFit(R, S):
    T_s ← sortBySizeAsc (R)
    R' ← ∅
    r ← ∅
    for t ∈ T_s do
        if |r| + |t| ≤ S then
            r ← r ∪ t
        else
            R' ← R' ∪ r
            r ← {t}
    end for
    R' ← R' ∪ r
    return R'
```

Our evaluation will revolve around the following three research questions:

RQ1. What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?

RQ2. How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?

RQ3. What kinds of static analysis tools would benefit from splitting strategies discussed in the paper?

4.1 Experimental Settings

Static analysis tools. We used two industrial static analysis tools with interprocedural analysis capabilities –Rapid [16] and Infer [36]. Rapid is a tool developed at AWS that performs IFDS/IDE-based [44] type-state analysis to detect incorrect usage of cloud-service APIs. Infer is a static analysis tool developed at Facebook that uses separation logic to detect memory-related issues such as null pointer exceptions, resource leaks, and concurrency race conditions. A third set of experiments will instead use Bandit [4], a static analysis tool to find common security issues in Python. Unlike the other tools in our experiments, Bandit processes each source code file individually.

Benchmark programs. We use three different benchmark suites in our experiment:

- the OWASP Benchmark (v1.2) [42] (`OWASP`), a well-known Java-based web application designed to evaluate the accuracy, coverage, and speed of automated software vulnerability detection tools. It contains 2,740 labeled test cases that demonstrate common web app vulnerabilities, including, e.g., command injection, weak cryptography, path traversal
- the Juliet Test Suite For Java [39] (`Juliet`), created by the NSA's Center for Assured Software (CAS) specifically for testing static analysis tools. It comprises 28,881 test cases that contain vulnerabilities for 112 different CWEs
- open-source packages from Maven Central [33] (`Maven`): Starting from a set of 26,142 open-source Java packages randomly sampled from Maven, we ran Rapid on all packages and, for each package, recorded the number of "seeds" (elements in the codebase that may lead to potential findings). This can be done in linear time. We filtered out packages with no seeds, since their analysis with Rapid is very inexpensive. We also removed any packages that crashed the tool. To make the splitting problem more challenging, out of the 4,611 remaining packages we selected those with at least 1,500 classes. That left us with 138 Maven packages.

Baseline and experiments. We evaluate the SplitMerge splitting strategy in comparison with the naïve SizeLimiting splitting strategy described in Sect. 2, and also against two baseline configurations that do not perform any splitting:

- No Splitting, Unlimited Time (`NoSplit-UT`): no splitting, 16Gb memory, 24h timeout. This strategy approximates the absence of latency constraints. We use the findings reported with `NoSplit UT` as reference to assess accuracy drops due to splitting.
- No Splitting, Unlimited Memory (`NoSplit-UM`): no splitting, 144Gb memory, 10 minutes timeout. This strategy imposes the same timeout we will use for splitting, but allows the analyzers to use virtually unlimited memory (no tool saturated the available memory in our experiments).

SplitMerge and SizeLimiting are allowed 16Gb of memory and 10 minutes timeout. We run all the experiments on Amazon EC2 C5.18xlarge instances (72 vCPUs, 144Gb RAM). We do not limit the number of cores a tool can use. We use Amazon Linux as operating system and `ulimit` to enforce memory limits.

Performance metrics. To evaluate our research questions, we collect the following metrics throughout the experimental campaign:

- **total findings**: the number of unique findings reported by each tool, used as a proxy to detect accuracy losses. When different splitting strategies are applied, we compare the number of findings against the baselines to estimate the impact of splitting. Notably, a tool may also report false positive findings in either the baseline or after splitting. In general, we do not have a reliable means to discriminate between true and false positives and for the sake of this work we pragmatically assume that, ideally, a splitting strategy should result in exactly the same set of findings as `NoSplit-UT`; differences would suggest an impact on the accuracy of the tool
- **best possible latency**: the longest analysis time for any of the partitions of the input program. This is the minimum waiting time for the user, excluding other network and service invocation latency
- **total time**: cumulative analysis time for all partitions. An index of the cumulative cost in computation time. Its value is related to the computational overhead induced by the redundancy allowed when splitting
- **peak heap usage**: the maximum Java heap memory used by an analysis tool written in Java. In our experiment, this metric is only measured for Rapid, which is written in Java

- **peak memory/max resident set size (RSS)**: the maximum amount of memory held by the process running an analysis tool at any time
- **No. of part.**: the number of partitions produced by a strategy for a given benchmark
- **sum of part. sizes**: the sum of storage size for all partitions produced in an experiment.

Configuration. SplitMerge is executed with: maximum partition size $S = 500$, neighborhood radius $k = 2$, and the percentage of high-connectivity vertices $p = 0.1$. These configuration values control the trade-off between latency, total CPU time, maximum memory, and impact on precision. For the experiments reported in this paper, we prescribed a maximum allowed latency of 10 minutes and a maximum of 16Gb of memory per tool process and systematically swept the configuration space to make sure the selected configuration comfortably allows analyzing a partition within our prescribed latency and memory limits. We acknowledge that different latency and resource constraints, benchmarks, and analysis tools may require different tuning of the parameters.

4.2 Experimental Results

4.2.1 ***RQ1.*** *What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?* We answer this question by looking at the total findings detected by the analyzers shown in Table 2 reporting on the `OWASP`, `Juliet`, and `Maven` benchmarks. In the table, **X** means the analyzer did not terminate within the given timeout or crashed, thus no results were reported. On all three benchmarks, SplitMerge allows both Rapid and Infer to detect more findings in comparison to SizeLimiting.

Regarding accuracy, we take the results of `NoSplit-UT` as the baseline for comparison (except for Rapid on `Juliet`, where this strategy did not produce a result). On `OWASP`, SplitMerge allowed Rapid to detect exactly the same number of findings (3,326) as with `NoSplit-UT`, without losing accuracy. We also compared the output of the tool with both strategies: the set of findings is exactly the same. In contrast, we lost 509 (3,326 – 2,817) findings with the naïve splitting strategy SizeLimiting, corresponding to 15% (509/3,326) of the total findings that can be detected by Rapid without splitting. For Infer, splitting the original code using SizeLimiting impacts its recall negatively, as Infer detected much fewer findings compared with non-splitting strategies (230 vs. 401). In contrast, SplitMerge allowed Infer to detect exactly the same findings as with non-splitting strategies.

On `Juliet`, Rapid did not finish the analysis using non-splitting strategies, which gave us no baseline to assess the impact of splitting on its accuracy, besides observing that SplitMerge returned more findings than SizeLimiting. Similarly to `OWASP`, splitting also resulted in loss of findings on Juliet for Infer. It also turns out that Infer crashed (exited with non-zero return code) when analyzing the partitions. As shown in Table 3, the crash rate is 2% with SizeLimiting and 6% with SplitMerge. We conjecture Infer is less tolerant to absences of dependent classes in comparison to Rapid, but further investigation is needed.

For the `Maven` benchmark, we conducted an experiment for each of the 138 Maven packages separately and aggregated the results (which show 138 partitions for the no-split strategies corresponding to the 138 packages analyzed). From the results on `Maven`, we can see that SplitMerge has less negative impact on both Infer and Rapid's findings compared to SizeLimiting, i.e., Rapid only lost 6.7% ((1,193-1,113)/1,193) of the findings using SplitMerge, while it is 17% ((1,193-991)/1,193) using SizeLimiting. On the other hand, Infer detected more findings with SplitMerge than `NoSplit-UT`, as it crashed less frequently (Table 3).

We remark once again that the number of findings is a coarse proxy to evaluate the differential accuracy, while assessing precision and recall of the different tools would require scoring each tool's output against a ground truth to establish which findings are true and false, which is beyond the scope of this study.

RQ1 takeaway: Splitting the original program can sometimes negatively impact a tool's accuracy. However our experiments show that smarter splitting strategies like SplitMerge can controllably reduce accuracy loss. For very large codebases (e.g., `Juliet`) and strict resource constraints, splitting may be the only option if we want to retrieve any findings, since the tools do not scale and thus end up returning no findings at all.

4.2.2 ***RQ2.*** *How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?* After evaluating the accuracy loss of splitting, with this research question, we evaluate analysis time and resource demand. A benefit of splitting a program and analyzing each partition in isolation is the possibility of running an instance of the analysis tool on each partition in parallel, thus reducing the user's waiting time. The best possible latency of analyzing partitions is the maximum analysis time required to analyze any such partitions. On all three benchmark suites, both Rapid and Infer achieved much better latency with splitting strategies on both SAST benchmarks (`OWASP` and `Juliet`) and real-world applications (`Maven`). Using SplitMerge, the analyzers achieved more than 2x speedup (Rapid: 32.3/8.5, Infer: 2.5/1.0) on the `Maven` packages in comparison to `NoSplit-UT`. Since Rapid is written in Java, we also measured the peak heap usage from the JVM and observed that with SplitMerge, Rapid used less than 13.7% of the peak heap consumption of `NoSplit-UT` on `OWASP` and 82% on `Maven`. We also measured the maximum resident set memory size, which indicates a significant reduction of peak memory consumption also for Infer.

RQ2 takeaway: Splitting the codebase allows us to analyze the partitions in parallel. On all three benchmark suites, our experiments show that splitting significantly reduced the latency for both analysis tools. Splitting also significantly reduced the memory required to analyze the benchmark suites compared to non-splitting.

4.2.3 ***RQ3.*** *What kinds of static analysis tools would benefit from splitting strategies discussed in the paper?* Comparing the results of Rapid and Infer, the first observation is that splitting allowed Rapid to analyze `Juliet`, which was intractably large for the non-splitting strategies. We also observed that Rapid is more tolerant than Infer with partitions that miss dependencies. On `OWASP` and `Juliet`, splitting increased the number of crashes (Table 3), resulting in a reduction of the number of findings. On `Maven` we observed the opposite effect, where analyzing the 138 packages individually

Strategy	Total Findings		Best Possible Latency (min)		Total Time (min)		Peak Heap Usage (MB)	Peak Memory Max RSS (MB)		No. of Part.	Sum of Part. Sizes (MB)
	Rapid	Infer	Rapid	Infer	Rapid	Infer	Rapid	Rapid	Infer		
OWASP											
NoSplit-UT	3,326	401	55.5	1.3	55.5	1.3	2,215	5,940.8	212.2	1	24.8
NoSplit-UM	X	401	X	1.3	X	1.3	X	X	213.6	1	24.8
SizeLimiting	2,817 (-15%)	230 (-42%)	1 (55x)	0.01 (130x)	6.8 (-87.7%)	0.7 (-46.1%)	271 (-87.7%)	5,406.8 (-8.9%)	81.8 (-61.4%)	10	24.8
SplitMerge	3,326 (0%)	401 (0%)	1.4 (39x)	0.01 (130x)	8.6 (-84.5%)	0.9 (-30.7%)	304 (-86.2%)	5,997.2 (+0.9%)	79.9 (-62.3%)	14	25.6
Juliet											
NoSplit-UT	X	14,183	X	2.4	X	24.2	X	X	2,844.9	1	249.7
NoSplit-UM	X	X	X	X	X	X	X	X	X	1	249.7
SizeLimiting	7,758 (N/A)	12,456 (-12%)	1.6 (∞x)	0.09 (26x)	101.1 (N/A)	37.1 (+34.7%)	1,855.7 (N/A)	7,098.5 (N/A)	131.5 (-95.3%)	95	249.7
SplitMerge	8,803 (N/A)	13,866 (-2%)	6 (∞x)	0.08 (30x)	185.4 (N/A)	41.1 (+45.5%)	2,623.6 (N/A)	7,933.4 (N/A)	135.8 (-95.2%)	151	298.7
Maven											
NoSplit-UT	1,193	31,246	32.3	2.5	767.4	307.4	14,730.9	17,501	1,776	138	4,205
NoSplit-UM	1,186	32,172	10	1.0	521.8	270.3	14,012.0	48,802	1,790	138	4,213
SizeLimiting	991 (-17%)	18,087 (-42%)	9.1 (3.5x)	1.0 (2.5x)	137 (-82.1%)	190.2 (-38.1%)	11,483.4 (-22%)	16,861 (-3.6%)	964 (-45.7%)	778	4,381
SplitMerge	1,113 (-6.7%)	31,793 (+1.7%)	8.5 (3.8x)	1.0 (2.5x)	543.2 (-30.3%)	715.1 (+132%)	12,181.7 (-17.03%)	17,182 (-1.8%)	1,025 (-42.2%)	2,641	15,756

Table 2: Comparing both SizeLimiting and SplitMerge to baselines on OWASP, Juliet and Maven. The percentages in the rows for SizeLimiting and SplitMerge strategies correspond to the reduction (or gain) in the number of findings, total time and memory usage when compared with NoSplit-UT. Best Possible Latency column shows the speedup achieved with SizeLimiting and SplitMerge strategies. In the cases where NoSplit-UT failed to give a result within 24 hours, we report the speedup as ∞x and the number of findings as N/A.

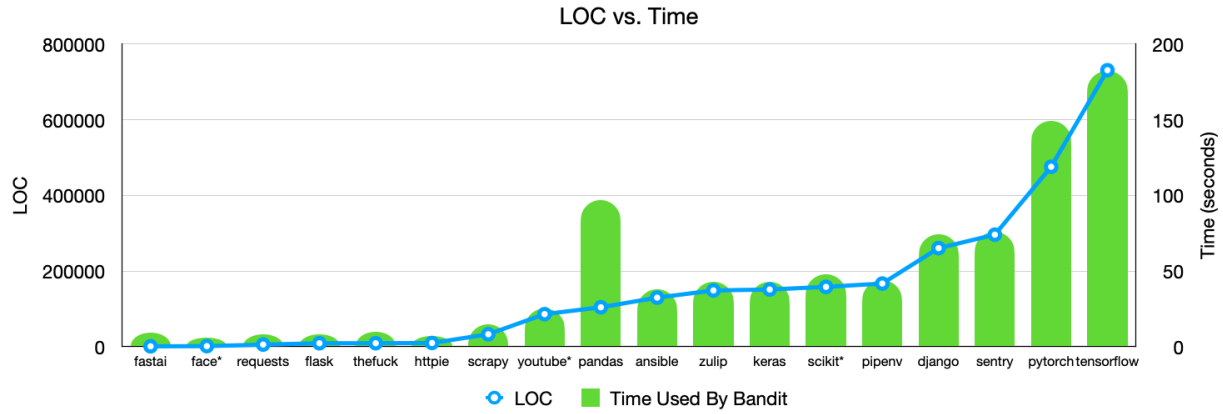


Figure 3: LOC of the 17 open source Python projects with dependency analysis and analysis time used by Bandit.

led to more crashes than grouping all their sources and splitting them with SizeLimiting or SplitMerge. However, for most benchmark applications, the difference in number of findings with and without splitting was limited for both Rapid and Infer, while Rapid was much faster in analyzing all subjects, significantly reducing both the best possible latency (by 73-97% with SplitMerge) and the total computation time (29-84% with SplitMerge). Also for Infer the best possible latency dropped by 60-99%, while the total computation time remained around the same order of magnitude, taking into account that the different number of crashes between NoSplit-UT and SplitMerge make it difficult to discern how much computation time ultimately led to results rather than crashing. Infer also showed a significant reduction in memory demand, up to 95% on Juliet.

A worst-case scenario for SplitMerge. To better define the target application scope of a dependency-aware splitting strategy like SplitMerge, we report on an additional experiment using Bandit [4], a static analysis tool for Python. Bandit processes each source file independently by building an abstract syntax tree and inspecting it for error patterns. While the previous experiments analyzed Java applications, SplitMerge is not restricted to a specific language, provided that the user can specify the elementary code units to partition (e.g., files, modules, or classes) and can identify their dependencies. In the case of Python, dependencies between its classes can be extracted using Snakefood [10] and Importlab [17].

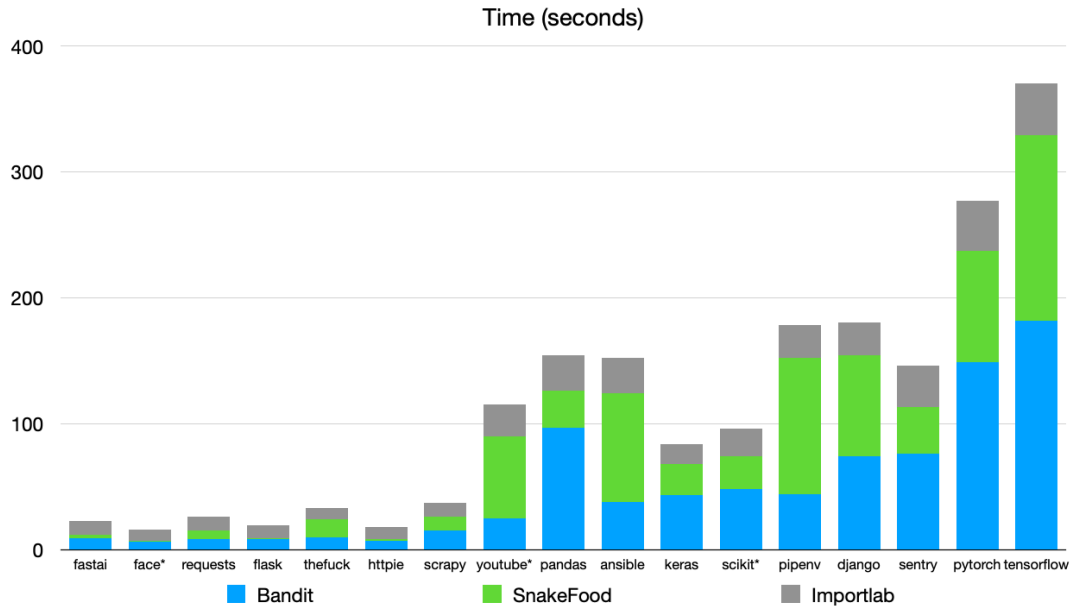


Figure 4: Time used by dependency analysis compared to time used by Bandit.

Strategy	Rapid Timeout	Infer Timeout	Rapid Crash	Infer Crash	Rapid Success	Infer Success
OWASP						
NoSplit-UT	0%	0%	0%	0%	100%	100%
NoSplit-UM	100%	0%	0%	0%	0%	100%
SizeLimiting	0%	0%	0%	20%	100%	80%
SplitMerge	0%	0%	0%	0%	100%	100%
Juliet						
NoSplit-UT	100%	0%	0%	0%	0%	100%
NoSplit-UM	100%	100%	0%	0%	0%	0%
SizeLimiting	0%	0%	0%	2%	100%	98%
SplitMerge	0%	0%	0%	6%	100%	94%
Maven						
NoSplit-UT	0%	4%	0%	24%	100%	72%
NoSplit-UM	0%	4%	0%	22%	100%	74%
SizeLimiting	0%	0.4%	0%	9.6%	100%	90%
SplitMerge	0%	1%	0%	16%	100%	83%

Table 3: Timeout, crash and success rates of analysis runs.

We ran the three tools of the analysis pipeline (Snakefood, Importlab, and Bandit) on 17 popular Python open-source projects[3] and recorded the analysis time. Figure 3 shows the lines of code (LOC, bars) vs analysis time for each program (line). The analysis pipeline including dependency analysis and Bandit shows, as expected, very high scalability, taking only about 3 minutes to analyze `tensorflow`, the largest program in the set. The breakdown of the time required for each step of this analysis pipeline is shown in Figure 4, from which it is evident that dependency analysis takes a significant proportion of the pipeline time, up to 73% for `youtube`, while Bandit only takes a smaller fraction of the pipeline time.

This experiment represents a worst-case scenario for SplitMerge—the analysis does not benefit from preserving dependency information, thus dropping the benefits of SplitMerge's heuristics. To parallelize Bandit's analysis it would be more effective to use SizeLimiting, avoiding the overhead of dependency analysis, which, contrary to the case of costlier interprocedural analyses with Rapid and Infer, does not pay off with Bandit's per-file analysis.

RQ3 takeaway: Our experiments show that splitting is useful to limit the per-machine resource consumption of static analysis tools and different tools require different splitting strategies. Inter-procedural analyses benefit most from splitting even with a naive strategy like SizeLimiting. For example, splitting the input codebase allowed Rapid to analyze code much faster than without splitting. Whereas for Infer, the most prominent benefit of splitting was the reduction in memory usage. Our experiments also show that for inexpensive linter-like or intra-procedural analyses such as Bandit, SizeLimiting may be more beneficial than dependency-guided-splitting strategy like SplitMerge.

5 RELATED WORK

SAST platforms. Several SAST platforms in the literature provide static analysis as a service and present a simple interface to developers that allows them to run a variety of static analysis tools. One of the earliest platforms is Review Bot [6], which integrates FindBugs [18], PMD, and CheckStyle into the code review process. Review Bot runs in the code review process on changes small enough to be reviewed by humans, and the tools are fast linting tools, so it did not have an immediate need for splitting.

The Khasiana1 web portal [37] integrates three static analyzers (FindBugs [18], Safe [21], and Xylem [38]) under a unified service interface. It focuses on discussing the usefulness of the reported findings. The tools are evaluated on hand-picked projects in which scalability is not an issue.

Two more recent platforms are Software Assurance Marketplace (SWAMP) [30] and Google's Tricorder [49] (and its open-source version, ShipShape). These platforms focus on making it easy to plug in additional analyzers. They provide orchestration as well as aggregation and management of recommendations. They do not

[3] https://dev.to/biplov/17-popular-python-opensource-projects-on-github-21ae

focus on how to deal with large inputs. Our splitting approach is agnostic to the specific SAST platform and could be applied to Khasiana1, SWAMP, or Tricorder/ShipShape as well.

An approach that shares a similar motivation to ours is implemented by Cheetah [15] which integrates several static analysis tools into an IDE. Cheetah achieves fast response times by gradually increasing the scope of the analysis: it starts by analyzing only the method currently being edited in the IDE, then increases the scope to class, package, and project level. This allows them to deliver some results early on while gradually increasing the precision if the user is willing to wait for it. In this paper, rather than gradually increasing the scope until reaching a time limit, we reduce the scope by splitting—simplifying the input, to get results within certain resource constraints.

A wide range of static program analysis problems can be viewed as instances of the Context-Free Language (CFL) Reachability problem [45], e.g., inter-procedural dataflow analysis [44], control flow analysis [56], set-constraints [28], specification-inference [8], shape analysis [45], object-flow analysis [58], pointer and alias analysis [31, 59], and program slicing [24] to name a few. Unfortunately, the worst case time complexity for solving CFL-reachability problems is $O(n^3)$, which is known as the "cubic bottleneck" [23]. As a result, highly precise analysis of large-scale software is challenging. In the literature, most work only focuses on very specific optimizations to a particular analysis that are not applicable in general.

Splitting input representation. Singh *et al.* proposed a technique to speed analysis with Polyhedra domain in abstract interpretation by partitioning variables into subsets such that the constraints only exist between variables in the same subset [51]. To mitigate the path explosion problem in symbolic execution, Trabish *et al.* [54] introduced *chopped symbolic execution*, an approach in which users can identify *unimportant* parts in the code, and the symbolic analysis tries to avoid those parts. Barnat *et al.* [7] propose a distributed algorithm for model checking LTL-formulas. The algorithm works by first partitioning and then exploring the state space in parallel. In model checking based on symbolic state representation, the technique in [22] partitions BDDs into smaller BDDs (each representing a subset of states) which are subsequently given to different processes. Kumar *et al.* [29] present a technique for distributed explicit state model checking to improve run time performance.

Parallel/distributed static analysis. The problem of scaling static analysis to potentially very large inputs is also discussed in [20] where the authors present a distributed call-graph construction algorithm designed to run in the cloud. We share the motivation that static analysis needs to be elastic to scale to very large inputs but our approach is designed to be agnostic to specific static analysis tools and techniques. Mendez-Lojo *et al.* [35] proposed a technique to parallelize inclusion-based points-to analysis by formulating it in terms of constraint graph rewrite rules. Su *et al.* [53] introduced a parallel solution to CFL-reachability based pointer analysis by avoiding redundant graph traversals using data sharing and query scheduling. Rodriguez *et al.* [47] presented an actor-model-based parallel algorithm for solving IFDS data-flow problems. Albarghouthi *et al.* [1] proposed a framework to parallelize top down inter-procedural analysis using the MapReduce paradigm. Facebook uses Infer [11] based on bi-abduction which is modular—generates summaries for methods in the program and compositional—composes summaries at call sites. Nevertheless, as we show in our experiments, even a modular analysis like Infer does not scale to large or complex inputs under practical resource constraints. BigSpa [60] provides a data-parallel algorithm for CFL-reachability based static analysis. Graspan [57] and Grapple [61] are single-machine, disk-based tools that model specific static analysis problems, taint analysis and type-state analysis respectively, as transitive closure computation on graphs.

The above approaches parallelize or distribute the analysis workload in a way that is specific to the tool or technique at hand. Most of them modify the existing tool. In our case, we do not introduce any changes to the SAST tools; an important design goal is to be able to add new off-the-shelf tools and update existing tools (as new versions are released) without having to maintain our own modified versions of them.

Automated refactoring. The goal of splitting is to break a large program into smaller units that are sufficiently self-contained to be analyzed in isolation. This is similar to the refactoring problem of breaking up a monolithic system into smaller components. An overview of such refactoring techniques is given in [19]. Recently, we see approaches based on machine learning (e.g., [14]), dynamic analysis (e.g., [27]), and static analysis (e.g., [34]). The splitting problem discussed in this paper is simpler than the problem of automatically refactoring in the sense that our partitions do not need to be functioning programs; they just need to to be sufficiently self-contained for analysis purposes.

Graph partitioning and communities. There is a rich body of work [25] on graph clustering for community detection in networks studied as graphs, e.g. social networks, academic citation networks, and collaboration networks—which might provide a theoretical grounding for future research. However, to the best of our knowledge this line of work currently focuses on preserving maximal connectivity, as opposed to our goal of attaining an effective trade-off between preserving connectivity and load-balancing partitions.

6 CONCLUSION

We discussed how splitting of static analysis inputs can be used to effectively limit the maximum resource consumption of an analysis tool. We motivated that this is an important problem when operating a SAST platform, as adding new analysis tools to the platform must not increase the maximum latency for existing customers.

Our evaluation shows that the splitting strategy has a significant impact on the outcome of the static analysis tool and that not all strategies are suitable for all tools. We showed that, for more complex (super-linear) static analysis tools, more advanced splitting strategies are needed to minimize the effect on the reported number of findings. For inexpensive linter-style tools like Bandit, we see that the overhead of a complex splitting strategy outweighs the benefits and that a simple splitting strategy is sufficient.

We believe that, in the future, the process of picking a splitting strategy and tuning the parameters when adding a tool to a SAST platform can be fully automated. A tool can be run with different configurations on regression data to identify the best combination of strategy and configuration. Also, the configuration parameters for each tool can be re-adjusted on the fly as the available hardware improves or the static analysis tools get updated.

REFERENCES

[1] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. 2012. Parallelizing top-down interprocedural analyses. In *PLDI*. ACM, 217–228.

[2] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[3] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 725–735. https://doi.org/10.1145/2884781.2884816

[4] Python Code Quality Authority. 2008. Bandit. https://bandit.readthedocs.io/en/latest/.

[5] Brenda S Baker and Edward G Coffman, Jr. 1981. A tight asymptotic bound for next-fit-decreasing bin-packing. *SIAM Journal on Algebraic Discrete Methods* 2, 2 (1981), 147–152.

[6] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 931–940.

[7] Jiri Barnat, Lubos Brim, and Jitka Stríbrná. 2001. Distributed LTL Model-Checking in SPIN. In *SPIN (LNCS, Vol. 2057)*. Springer, 200–216.

[8] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In *POPL*. ACM, 553–566.

[9] Andrew Binstock. 2022. Gitleaks: a SAST tool for detecting and preventing hardcoded secrets like passwords, api keys, and tokens in git repositories. https://blogs.oracle.com/javamagazine/post/java-class-file-constant-pool.

[10] Martin Blais. 2007. Snakefood. https://furius.ca/snakefood/doc/snakefood-doc.html.

[11] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NFM (LNCS, Vol. 9058)*. Springer, 3–11.

[12] Justin Collins. 2022. Brakeman: a static vulnerability scanner specifically designed for Ruby on Rails applications. https://brakemanscanner.org/.

[13] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. 2005. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 449–461.

[14] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 72–80. https://ojs.aaai.org/index.php/AAAI/article/view/16079

[15] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-Time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[16] 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.* ACM, New York, NY, USA, 1416–1426. https://doi.org/10.1145/3468264.3473934

[17] Martin DeMello et al. 2017. Importlab. https://github.com/google/importlab.

[18] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 234–245.

[19] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. 2018. From Monolith to Microservices: A Classification of Refactoring Approaches. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment - First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11350)*, Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer (Eds.). Springer, 128–141. https://doi.org/10.1007/978-3-030-06019-0_10

[20] Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. 2017. Toward Full Elasticity in Distributed Static Analysis: The Case of Callgraph Analysis *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 442–453. https://doi.org/10.1145/3106237.3106261

[21] Emmanuel Geay, Eran Yahav, and Stephen Fink. 2006. Continuous code-quality assurance with SAFE. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 145–149.

[22] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. 2005. Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation. In *IFIP (LNCS, Vol. 3725)*. Springer, 129–145.

[23] Nevin Heintze and David A. McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *LICS*. IEEE, 342–351.

[24] Susan Horwitz, Thomas W. Reps, and David W. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *PLDI*. ACM, 35–46.

[25] Di Jin, Zhizhi Yu, Pengfei Jiao, Shirui Pan, Dongxiao He, Jia Wu, Philip Yu, and Weixiong Zhang. 2021. A Survey of Community Detection Approaches: From Statistical Modeling to Deep Learning. *IEEE Transactions on Knowledge and Data Engineering* (2021). https://doi.org/10.1109/TKDE.2021.3104155

[26] David S Johnson. 1973. *Near-optimal bin packing algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[27] Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. *Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*. Association for Computing Machinery, New York, NY, USA, 1214–1224. https://doi.org/10.1145/3468264.3473915

[28] John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI*. ACM, 207–218.

[29] Rahul Kumar and Eric G. Mercer. 2005. Load Balancing Parallel Explicit State Model Checking. *ENTCS* 128 (2005), 19–34.

[30] James A. Kupsch, Barton P. Miller, Vamshi Basupalli, and Josef Burger. 2017. From continuous integration to continuous assurance. In *2017 IEEE 28th Annual Software Technology Conference (STC)*. 1–8. https://doi.org/10.1109/STC.2017.8234450

[31] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *CC (LNCS, Vol. 7791)*. Springer, 61–81.

[32] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.

[33] Maven. 2022. List of Maven Packages. https://gist.github.com/linghuiluo/1b82866051e4c4ebb0fd065549f60100.

[34] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, Ilkay Altintas and Shiping Chen (Eds.). IEEE, 524–531. https://doi.org/10.1109/ICWS.2017.61

[35] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel inclusion-based points-to analysis. In *OOPSLA*. ACM, 428–443.

[36] Meta. 2022. Infer: a static analysis platform for Java, C, and Objective-C. https://fbinfer.com/docs/about-Infer.

[37] Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran. 2010. Making Defect-Finding Tools Work for You. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/1810295.1810310

[38] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate interprocedural null-dereference analysis for Java. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 133–143.

[39] NIST. 2022. Juliet Test Suite for Java. https://samate.nist.gov/SRD/testsuite.php.

[40] Oracle. 2022. JDeps - Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/9/tools/jdeps.htm.

[41] OWASP. 2022. FindSecBugs: the SpotBugs plugin for security audits of Java web applications. https://find-sec-bugs.github.io/.

[42] OWASP. 2022. OWASP. https://owasp.org/www-project-benchmark/.

[43] Praetorian, Inc. 2021. Gokart: a security-oriented static analysis for Golang with a focus on minimizing false positives. https://github.com/praetorian-inc/gokart/.

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

[45] Thomas W. Reps. 1997. Program Analysis via Graph Reachability. In *ISLP*. MIT, 5–19.

[46] Zachary Rice. 2018. Understanding the constant pool inside a Java class file. https://github.com/zricethezav/gitleaks/.

[47] Jonathan Rodriguez and Ondrej Lhoták. 2011. Actor-Based Parallel Dataflow Analysis. In *CC (LNCS, Vol. 6601)*. Springer, 179–197.

[48] Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In *Compiler Construction*, Laurie Hendren (Ed.). Springer, 53–68.

[49] 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.

[50] Amazon Web Services. 2022. Elastic Compute Cloud (EC2) Pricing. https://aws.amazon.com/ec2/pricing/.

[51] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *POPL*. ACM, 46–59.

[52] SonarSource, S.A. 2008. Sonarqube: a Static Application Security Testing (SAST) solution to detect security issues in code review. https://www.sonarqube.org/features/security/.
[53] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *ICPP*. IEEE Computer Society, 451–460.
[54] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *ICSE*. ACM, 350–360.
[55] Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. 2020. Incremental Flow Analysis through Computational Dependency Reification. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 25–36. https://doi.org/10.1109/SCAM51674.2020.00008
[56] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *ESOP (LNCS, Vol. 6012)*. Springer, 570–589.
[57] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code. In *ASPLOS*. ACM, 389–404.
[58] Hao Yuan and Patrick Th. Eugster. 2009. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In *ESOP (LNCS, Vol. 5502)*. Springer, 175–189.
[59] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL*. ACM, 197–208.
[60] Zhiqiang Zuo, Rong Gu, Xi Jiang, Zhaokang Wang, Yihua Huang, Linzhang Wang, and Xuandong Li. 2019. BigSpa: An Efficient Interprocedural Static Analysis Engine in the Cloud. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 771–780. https://doi.org/10.1109/IPDPS.2019.00086
[61] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *EuroSys*. ACM.