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
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, naively 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 many 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("foo") is used in a relative pathname without input validation. An attacker could provide an input like 

```java
// 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 Thing().doSomething(p);
        File fileTarget = new File("/tmp", bar);
        response.getWriter().println("...");
    }
}
```

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 data that is shared between partitions.

---

\(^1\)https://github.com/OWASP-Benchmark/BenchmarkJava/tree/53874cc8751e43b63de35b1b91add47c2f9121d8/
3 THE SPLITMERGE STRATEGY

We aim to distribute the analysis of a program \( P \), consisting of \( n \) files \( \{F_1, \ldots, F_n\} \), 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 JDep [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 \( \tilde{F} = \{F_1, F_2, \ldots\} \) and a neighborhood radius \( k > 0 \), Alg. 1 constructs for each file a 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 \( O(n^3) \); 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

\[ S = \{A, B, C, D, E, F\} \]

\[ S = \{A, B, C\}, \{B, C\}, \{C\}, \{D, B, C\}, \{E, A, B, C, D, F\}, \{F\} \]
Algorithm 1: Initial partitions.

**Input**: Files \( F = \{f_0, f_1, \ldots \} \), neighborhood radius \( k \)

**Output**: Initial partitions \( R = \{r_0, r_1, \ldots \} \)

1. \( (V, E) \leftarrow \text{computeDependencyGraph}(F) \)
2. // augment dependency relation
3. for \( v \in V \) do
4.   neighbors \( \leftarrow \text{verticesWithinDistance}(v, k) \)
5.   for \( n \in \text{neighbors} \) do
6.     \( E \leftarrow E \cup (v, n) \)
7. end for
8. end for
9. // build initial partition
10. \( R \leftarrow \emptyset \)
11. for \( v \in V \) do
12.   \( r \leftarrow \{v\} \)
13.   for \( (v, u) \in E \) do
14.     \( r \leftarrow r \cup \{u\} \)
15. end for
16. \( R \leftarrow R \cup r \)
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 \texttt{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 \texttt{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
to efficiently compute near-optimal solutions [26]. Among these, we adopted \texttt{nextfit} [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 \texttt{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 \texttt{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 \texttt{SplitMerge}, the resulting partitions satisfy the desired properties: (1) each partition is smaller than the prescribed size \( S \), i.e., \( |r_i| \leq S \); (2) the union of the partitions contains all files of the input program, i.e., \( \cup r_i = F \). 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 \texttt{SplitMerge} with three analysis tools on a portfolio of Java and Python benchmarks.
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 **INFERENCE** [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. **INFERENCE** 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

---

**Algorithm 3: Merge.**

**Input:** Partitions $R = \{r_0, r_1, \ldots, r_n\}$, maximum size $S$

**Output:** Merged partitions $R’ = \{r’_0, r’_1, \ldots, r’_m\}$, $m \leq n$

```plaintext
1. for $r_i \in R$
2.    if $\exists r_j \in R$ s.t. $r_j \subseteq r_i$ and $i \neq j$ then
3.       $R \leftarrow R \setminus \{r_i\}$
4.    end if
5. end for
6. return nextFit ($R$, $S$)
7
8. Function nextFit ($R$, $S$):
9.    $T_R \leftarrow$ sortBySizeAsc ($R$)
10.   $R’ \leftarrow 0$
11.   $r \leftarrow 0$
12. for $t \in T_R$
13.    if $|r| + |t| \leq S$ then
14.       $r \leftarrow r \cup t$
15. else
16.       $R’ \leftarrow R’ \cup r$
17.       $r \leftarrow \{t\}$
18. end for
19. $R’ \leftarrow R’ \cup r$
20. return $R’$
```
• 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% ((1,193-991)/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 controlably 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...
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 $\infty$.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Total Findings</th>
<th>Best Possible Latency (min)</th>
<th>Total Time (min)</th>
<th>Peak Heap Usage (MB)</th>
<th>Peak Memory Max RSS (MB)</th>
<th>No. of Part.</th>
<th>Sum of Part. Sizes (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rapid</td>
<td>Infer</td>
<td>Rapid</td>
<td>Infer</td>
<td>Rapid</td>
<td>Infer</td>
<td>Rapid</td>
</tr>
<tr>
<td>NoSplit-UT</td>
<td>3,326</td>
<td>401</td>
<td>55.5</td>
<td>1.3</td>
<td>55.5</td>
<td>1.3</td>
<td>2,215</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>2,817</td>
<td>230</td>
<td>(55%)</td>
<td>(130x)</td>
<td>0.01</td>
<td>(52%)</td>
<td>(130x)</td>
</tr>
<tr>
<td>SplitMerge</td>
<td>3,326</td>
<td>401</td>
<td>1.4</td>
<td>(39x)</td>
<td>0.01</td>
<td>(52%)</td>
<td>(130x)</td>
</tr>
<tr>
<td>NoSplit-UM</td>
<td>X</td>
<td>14,183</td>
<td>2.4</td>
<td>X</td>
<td>2.4</td>
<td>X</td>
<td>14,242</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>7,758</td>
<td>12,456</td>
<td>1.6</td>
<td>(26x)</td>
<td>0.09</td>
<td>(26x)</td>
<td>101.1</td>
</tr>
<tr>
<td>SplitMerge</td>
<td>8,803</td>
<td>13,866</td>
<td>6</td>
<td>(30x)</td>
<td>0.08</td>
<td>(30x)</td>
<td>185.4</td>
</tr>
<tr>
<td>NoSplit-UT</td>
<td>1,193</td>
<td>31,246</td>
<td>32.5</td>
<td>2.5</td>
<td>767.4</td>
<td>307.4</td>
<td>14,730.9</td>
</tr>
<tr>
<td>NoSplit-UM</td>
<td>1,186</td>
<td>32,172</td>
<td>10</td>
<td>1.0</td>
<td>521.8</td>
<td>270.3</td>
<td>14,012.0</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>991</td>
<td>18,887</td>
<td>9.1</td>
<td>(25x)</td>
<td>1.0</td>
<td>(25x)</td>
<td>137</td>
</tr>
<tr>
<td>SplitMerge</td>
<td>1,113</td>
<td>31,793</td>
<td>8.5</td>
<td>(25x)</td>
<td>1.0</td>
<td>(25x)</td>
<td>543.2</td>
</tr>
</tbody>
</table>

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, without splitting was limited for both Rapid and Infer, while Rapid was much faster in analyzing all subjects, significantly reducing.

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].
We ran the three tools of the analysis pipeline (Snakefood, Importlab, and Bandit) on 17 popular Python open-source projects 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.

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

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Rapid Timeout</th>
<th>Rapid Crash</th>
<th>Rapid Success</th>
<th>Infer Timeout</th>
<th>Infer Crash</th>
<th>Infer Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWASP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoSplit-UT</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>NoSplit-UM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>0%</td>
<td>0%</td>
<td>20%</td>
<td>100%</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>SplitMerge</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Juliet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoSplit-UT</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>NoSplit-UM</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>0%</td>
<td>0%</td>
<td>2%</td>
<td>100%</td>
<td>98%</td>
<td></td>
</tr>
<tr>
<td>SplitMerge</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>6%</td>
<td>100%</td>
<td>94%</td>
</tr>
<tr>
<td>Maven</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NoSplit-UT</td>
<td>0%</td>
<td>4%</td>
<td>0%</td>
<td>24%</td>
<td>100%</td>
<td>72%</td>
</tr>
<tr>
<td>NoSplit-UM</td>
<td>0%</td>
<td>4%</td>
<td>0%</td>
<td>22%</td>
<td>100%</td>
<td>74%</td>
</tr>
<tr>
<td>SizeLimiting</td>
<td>0%</td>
<td>0.4%</td>
<td>9.6%</td>
<td>100%</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>SplitMerge</td>
<td>0%</td>
<td>1%</td>
<td>0%</td>
<td>16%</td>
<td>100%</td>
<td>83%</td>
</tr>
</tbody>
</table>

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

---

3https://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 [51, 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 programming programs; they just need 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.
Input Splitting for Cloud-Based Static Application Security Testing Platforms

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

REFERENCES


[8]<boost>Atualizado para 2022</boost>


