Commit-Time Incremental Analysis

Padmanabhan Krishnan
Rebecca O'Donoghue
Nicholas Allen
Yi Lu*
Oracle Labs
Brisbane, Queensland, Australia
(paddy.krishnan,rebecca.o.donoghue,nicholas.allen)@oracle.com

Abstract

Most changes to large systems that have been deployed are quite small compared to the size of the entire system. While standard summary-based analyses reduce the code that is reanalysed, they, nevertheless, analyse code that is not changed. For example, a backward summary-based analysis, will examine all the callers of the changed code even if the callers themselves have not changed. In this paper we present a novel approach of having summaries of the callers (called forward summaries) that enables one to analyse only the changed code. An evaluation of this approach on two representative examples, demonstrates that the overheads associated with the generation of the forward summaries is recovered by performing just one or two incremental analyses. Thus this technique can be used at commit-time where only the changed code is available.

CCS Concepts • **Security and privacy → Software security engineering**; • **Software and its engineering → Automated static analysis**; *Software maintenance tools.*

Keywords incremental analysis, caller and callee summarisation

ACM Reference Format:
Padmanabhan Krishnan, Rebecca O'Donoghue, Nicholas Allen, and Yi Lu. 2019. Commit-Time Incremental Analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315568.3329968

*Currently at Queensland University of Technology. Email:yt.lu@qut.edu.au

SOAP '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329968

1 Introduction

If static analysis tools are to be used regularly in the software development cycle, they must be fast and precise from a developer's perspective [14]. There are numerous techniques that can be used to make static analysis both scalable and precise [3]. Current static analysis tools can be integrated as part of the nightly build process. However, they are not efficient enough to be run at commit time. This is because most tools require reanalysis of a large subset of the code that has not changed, but either depend on or potentially influence the changed code. Additionally, software developers will generally be interested in viewing only the side effects of small changes that they have made to an application. That is, the tool must report results only on the changed component.

Tools such as Tricorder [14] perform only lint-level checks when presented with files that correspond to a change. However, to do deeper level analysis, it needs to trigger a build of all targets affected by the change. Approaches such as Reviser [2] and JIT Static Analysis [5] address the issue of incremental analysis.

To perform an analysis on only modified components of an application, the information required about any unchanged code must be persisted in a summary [6]. This summary can then be used in place of any analysis of the unchanged code. The performance gained through recomputation of only changed code must not be outweighed by the additional cost of computing, storing and using the summaries amortised over a set of analyses.

In the context of security-related analysis, one needs to detect the flow of potentially malicious input (from taint sources) to methods that perform security-sensitive operations (taint sinks).

Consider Listing 1, which illustrates the flow of a tainted value. Here the taint source is the result of the call to the method `getParameter` on the request object `req`. The assignment of the return value now taints the value in `fName`. The taint sink is the `executeQuery` that executes the SQL command. If the value in `qstring` is tainted, executing the query in `qstring` can lead to a SQL injection. In this example, since the method `createQuery` does not perform any sanitisation on its input parameter, there is a potential security vulnerability.

Assume the method service is modified but the calls to the methods createQuery and getResult are maintained. Also assume that we are using the standard demand-driven [3] analysis. Here the callee (or backward) summary-based analysis will instantiate the summaries of the methods that are called by the method under analysis. In our case, the methods createQuery and getResult will be analysed. The analysis will conclude that the value in sName can flow to a security-sensitive sink.

To detect a security vulnerability, one needs to determine whether the value in sName is tainted. This requires an analysis of all the direct and indirect callers of service. In our example, the analysis needs to examine the methods setup and doPut to conclude that the original vulnerability is not removed.

Listing 1. Example: Taint Flow

```
public class MyServlet extends HttpServlet {

  protected void
     doPut(HttpServletRequest req,
           HttpServletResponse res)
   {
     String fName = req.getParameter("name");
     setup(fName);
   }

  private void setup(String value)
  {
     Provider provider = new Provider(...);
     provider.service(value);

  }
}

public class Provider {

  Engine engine;

  public void service(String sName)
  {
     String sql =
        SQLHandler.createQuery(sName);
     result = engine.getResult(sql);
  }
}

class SQLHandler {

  public static String
         createQuery(String name)
  {
     String q;
     q = "select * from data where name=" + name;
     return q;
  }

class Engine {

  public ResultSet getResult(String qstring)
  {
     return = conn.executeQuery(qstring);
  }
}
```

Unfortunately, for commit-time analysis, the callers of service may not be available as part of the commit. Also, the number of methods that may need to be analysed as part of the call-chain is unclear (in the worst case, all methods). This means, under certain circumstances, the performance of the standard backward analysis may be unacceptable.

To avoid reanalysing the callers, the effect of the callers must be summarised. To distinguish such summaries from the callee-summaries that we is used in the demand-driven analysis, we call these "forward" summaries and show how they can be used to analyse only the changed code. Thus, we get a truly commit-based analysis where only the code that has been changed is analysed. We also show that the overheads of generating and using the "forward" summaries is much less than the savings obtained by analysing only the changed code.

Our technique is explained in Section 2. Our implementation and experimental results along with the limitations of our approach are described in Section 3 and Section 4, respectively.

2 Our Approach

As observed in the previous section, extending the standard backward dataflow analysis techniques to the incremental case will compute a new summary for any changed code and then propagate all flows backwards, recomputing summaries up the call stack up to the entry methods. That is, all the methods that invoke (directly or indirectly) the changed method will be reanalysed. This technique will often result in recomputing summaries of unchanged code to ensure that no flows are lost. Our aim is to recompute summaries for changed code only, and to use persisted summaries for all unchanged code. Towards this, forward summarisation (i.e., relevant flows along call paths from entry points to relevant program locations) are used in conjunction with backward summarisation. This is illustrated using the call graph in Figure 1.

A standard backward analysis will first analyse the method m_4 and m_5 and generate summaries for them. Method m_5's summary will be used to analyse methods m_2 and m_3, while the summary of methods m_2, m_3 and m_4 will be used to analyse m_1. We call such summaries *backward summaries*. Let us now assume that method m_3 is changed. In the standard incremental analysis, the backward summary of m_5 will be used to reanalyse m_3 and the changed summary will be used to reanalyse m_1. This process is applied to all the callers of m_1. In our approach, we will not reanalyse m_1 or its callers.

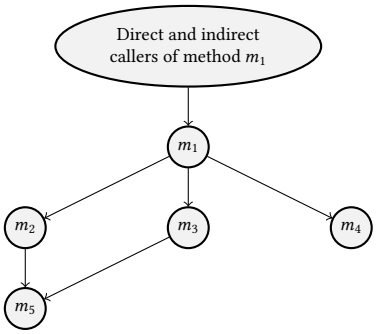

Figure 1. Example Call Graph

Rather, we will use the *forward* summary of m_1 along with the *backward* summary of m_5 to reanalyse m_1. No other code needs reanalysis.

In our experience, summarising *all* the flows along call paths from the entry methods to other methods is not scalable because the summary is almost as large as the code itself. Because of this, we can summarise only a small subset of the flows. Therefore, we focus only on summarising relevant behaviour based on potential client analyses. For our use case of security analyses, we summarise only the flow of tainted information. But our technique can be generalised to support any predicate, provided the size of the summaries is a fraction of the size of the program.

To support our incremental analysis, we need to extend the security analysis on the entire code to compute and persist the forward and backward summaries. The generation of such summaries is usually done as part of the nightly build and analysis process. It means that the overheads of generating the forward summaries is not borne by the actual commit-time analysis.

The analysis that is run nightly also reports all the vulnerabilities in the entire codebase. The incremental analysis needs to consult this report to indicate which vulnerabilities have been removed or introduced by the changed code.

3 Implementation and Experimental Results

3.1 Summaries as a Graph

To evaluate the performance of the incremental analysis technique, the key components were implemented using the PGX graph analytics package [8]. For our analysis, the program is represented as a directed graph. The nodes of the graph represent individual program values/variables at particular execution points within the application under analysis while the edges describe flows between them. While our implementation considers only values/variables, it could be extended to support field sensitivity by generalising the nodes of the graph to represent particular access paths.

In this representation, each individual method has a subset of nodes that are the entry points connected to callers (e.g. arguments), and a subset of nodes that are the exit points connected to callees (e.g. actual parameters at callsites).

The nodes of the graph possess properties describing their location within the application and their taint status. This status indicates whether a node should be considered a taint source or sink during an analysis of that method. A security vulnerability is reported if there is a flow path through the graph from a taint source to a taint sink.

The precomputed forward and backward summaries related to an individual method designate some method entry nodes as local taint sources (forward summaries). This means that there is a backward taint flow to a universal taint source, and some method exit nodes are local taint sinks (backward summaries). Therefore, there exists a forward taint flow to a universal taint sink.

To perform an incremental analysis using the precomputed summaries, the local subgraph for each changed method is analysed. A security vulnerability is reported if there is a flow path through the local subgraph, from a taint source (local or universal) to a taint sink (local or universal).

For example, Figure 2 shows the graph representation of the program in Listing 1, with the return value of getParameter marked as a universal taint source and the parameter passed to executeQuery marked as a universal taint sink. The highlighted edges indicate the taint flow from source to sink detected by the full analysis.

Suppose that the definition of service was changed, as shown in Listing 2:

Listing 2. Example: Change

```
public class Provider {
  public void service(String sName) {
    String pre = "T" + sName;
    String sql = SQLHandler.createQuery(pre);
    result = engine.getResult(sql);
  }
}
```

To perform an incremental analysis of this method, the changed code must be recompiled, and a new local graph representation derived. Then, the new local graph is analysed in the context of the relevant summaries produced by the full analysis, as shown in Figure 3. In this case, the forward summary for service itself is needed, as are the backward summaries of the callees createQuery and getResult. The forward summary for service indicates that the first argument is a local source (it is tainted via some caller). The backward summary of createQuery indicates that taint flows from the first argument to the return value. The backward summary for getResult indicates that the first argument is a local sink (it eventually flows to a universal taint sink). The incremental analysis of the new local graph in the context of these previously computed summaries finds that there is a taint

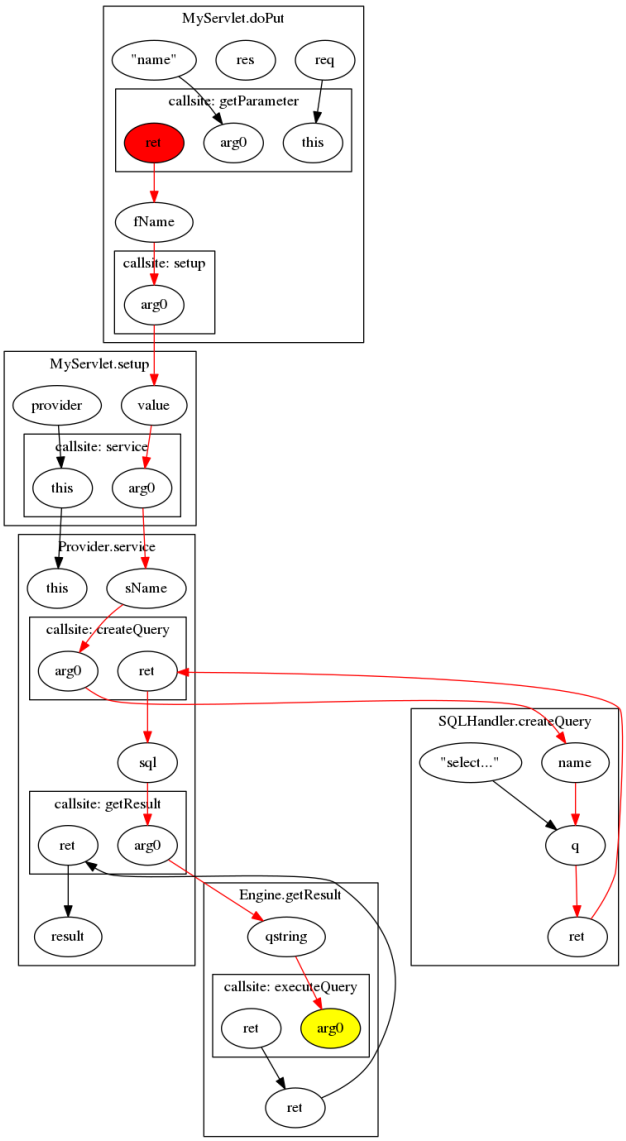

Figure 2. Example: Graph Representation

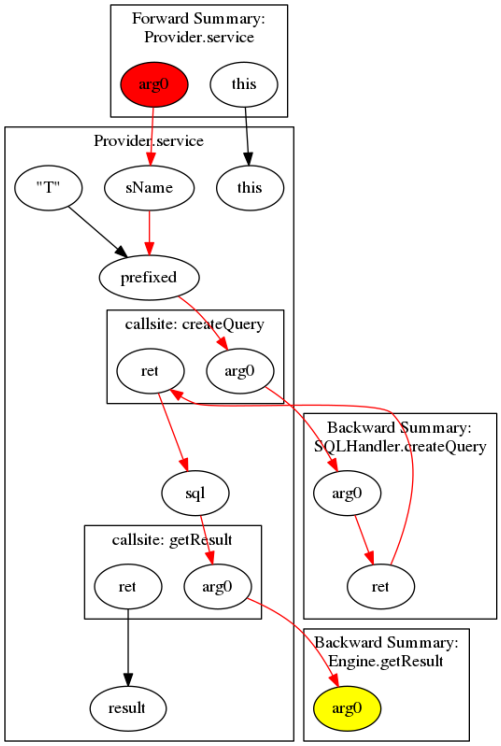

Figure 3. Example: Incremental Analysis

flow from the local source to the local sink (highlighted), and so a security vulnerability is reported.

3.2 Evaluation on Internal Codebases

The internal codebases selected for experimentation are representative cases of likely real-world uses of the technique. The two codebases discussed in this document referred to as "Codebase A" and "Codebase B" have 2.87 MLOC and 1.75 MLOC containing 132,485 and 56,117 methods, respectively.

Table 1 shows the time taken by the full analysis as well as the extra time to generate the different summaries, we produce these summaries using the Green Marl DSL [7] in PGX.

To evaluate the benefit of using our summary approach, we measure the time it takes to perform the incremental analysis using both the forward and backward summaries, and compare it against using only the backward summaries. We also report on the time it takes to read the various summaries that have been generated. Towards this, we randomly pick 100 methods to trigger the recomputation of the taint flows. Our focus is to identify new taint flows through the modifications using a reachability query. We use PGX's PGQL [10] queries to find a path from taint source to sink. Using only the backward summaries, we must analyse the method subgraph and all caller subgraphs to locate a path from a universal source to a universal or local sink. When using both forward and backward summaries, we analyse only the changed method's subgraph to find any paths between universal or local sources to universal or local sinks.

In our experiments, each change triggered, on average, the reanalysis of 15 methods for the standard backward incremental analysis. But Codebase A had 8 and Codebase B had 5 cases where the number of methods that required reanalysis was higher than 500. In Codebase A, there was was a pathological case where more than 14,000 methods had to be recomputed. We consider such cases as outliers because, normally, if a core component is changed, it is often better to do a full analysis. The change impacts a significant fraction of the code which makes commit-time analysis ineffective.

Table 1. Full Analysis And Summary Generation Time (ms)

Application	Full Analysis: No Summary Generation	Generating Only Backward Summaries	Generating both Forward and Backward Summaries
Codebase A	65,371	45	85
Codebase B	9,575	63	65

Table 2. Incremental Analysis Time (ms)

Application	Backward Incremental Analysis	Reading backward summaries	Our Incremental Analysis	Reading both Backward and Forward Summaries
Codebase A	15.5	169.5	1.6	11.3
Codebase B	14.0	166.5	1.26	11.1

The results, *ignoring the outliers*, are shown in Table 2. From these results, we can see the benefit of generating the forward and backward summaries when compared with the standard backward analysis. The extra cost of forward and backward summary generation is not very high, and can be recovered by performing only one forward and backward incremental analysis. Our experiments also confirm the benefits of the standard backward analysis using the summaries over the full analysis.

In summary, our experiments demonstrate that, for changing code, the forward and backward summary-based analysis is faster than performing a backward-only summary-based analysis. This is because the summaries of the callers of changed code are not needed when forward summaries are used.

4 Limitations

We have assumed that each incremental analysis of a changed method uses only the persisted forward and backward summaries to determine the effects of that method's callers and callees, respectively. Thus, when changes are committed and analysed, we do not invalidate the existing summaries and recompute the new summaries. If the information contained within a summary is rendered invalid by the change, the results will be inaccurate. For example, if a changed method appears on the call path to another changed method, and the change in the former removes a taint flow previously reaching the latter, the incremental analysis of the latter will consider only the pre-existing flows contained in the forward summary, resulting in a false positive. This also includes the well-known limitations of not having an incremental call graph construction. That is, we do not modify the call graph after a commit. To overcome this, we would need to have a form of modular points-to or heap analysis [12]. But these techniques are quite expensive because they have to summarise all the effects on the heap. While our current prototype implementation does not flag such issues, our implementation integrated with Parfait [4] will.

Thus, for our technique to be useful, the various changes have to be relatively small and independent of each other. As the full summaries will be recomputed as part of the nightly build, any potential errors in the incremental analysis will be short-lived. Also for large codebases, different programmers are likely to be making independent changes to different parts of the codebases and often only on their own branch. Because of this, the likelihood of errors is reduced. In cases of more pervasive or high-impact changes, it is better to rerun the full analysis, like Reviser [2] does, or use the trace-based invalidation technique [11].

We have evaluated our technique on only two internal codebases. While they are quite large, they may not be representative of different codebases. Furthermore, we have not used real commit data. We have relied on randomly marking methods as changed to evaluate the benefits of our incremental analysis.

Currently, our filtering of security vulnerabilities removed by the changed code is limited. A particular vulnerability could have multiple taint-sources and/or paths to the security-sensitive sink. For pragmatic reasons, we do not store all such possibilities as part of the report. The modified code could fix one of the causes, thereby signalling that the vulnerability has been removed. This should not be interpreted as all causes have been fixed. Only a full analysis can be be used to conclude that all the causes of the vulnerability have been fixed.

5 Related Work

Reviser [2] is closest to our work. They do show that in certain situations the incremental analysis takes roughly the same time as a fresh full analysis, as seen in our experiments on Codebase A. This high cost is incurred when the values from the changed code propagates to a number of callers. By storing the taint-flow summary for the callers (in the forward summary) and the destinations of the potential tainted value (in the backward summary), we do not need to reanalyse the callers or the callees. We can instantiate the summaries and indicate whether a security violation can be triggered by the changed code. While they look at all nodes that are reachable from changed nodes, we focus only on the changed code. Further experimentation is required to understand the relative strengths of the two approaches.

Another approach is a layered approach [5], where the analysis is first used locally and then used more globally, therefore the local analysis cannot detect defects caused by "global" flows. Because of this, they always need to perform the global analysis to detect all defects.

Sootkeeper [9] describes an infrastructure that enables one to store and, thus, reuse intermediate results. This infrastructure enables modular analysis, but this is, by itself, not sufficient. This is because if there is a change to the code, one still needs to determine what can be reused and what needs to be recomputed. In our implementation, we use similar ideas to persist summaries and reuse summaries for all the unchanged code.

[1] use slicing to get impacted code. This is orthogonal to our approach because we do not update the incremental summaries. [13] use path abstraction, which leads to very precise but not scalable solutions. If required, we can be flow sensitive, but path sensitivity is expensive and not scalable.

6 Conclusion

In summary, the main novelty of our work is that we do not analyse callers; instead we precompute "forward" summaries. These forward summaries capture only the state of relevant predicates (such as taint) at the point of entry of a particular method, covering all call paths to that method. While the standard backward summary approach allows incremental analysis with complexity dependent upon the changed methods and their transitive (unchanged) callers, our forward summary approach eliminates the need to re-analyse unchanged callers, allowing incremental analysis with complexity dependent upon only the methods directly modified by the change.

References

[1] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE*, pages 746–755. ACM, 2011.

[2] S. Arzt and E. Bodden. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *ICSE*, pages 288–298, 2014.

[3] E. Bodden. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In *SOAP*, pages 85–93. ACM, 2018.

[4] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo. Transitioning Parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, May/June 2012.

[5] L. N Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Just-in-time static analysis. In *ISSTA*, pages 307–317. ACM, 2017.

[6] D. Gopan and T. Reps. Low-level library analysis and summarization. In *Computer Aided Verification (CAV)*, number 4590 in LNCS. Springer, 2007.

[7] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[8] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A fast distributed graph processing engine. In *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 58:1–58:12. ACM, 2015.

[9] F. Kübler, P. Müller, and B. Hermann. Sootkeeper: Runtime reusability for modular static analysis. In *SOAP*, pages 19–24, 2017.

[10] Oracle Labs. PGX: PGQL Specification. Available: https://docs.oracle.com/cd/E56133_01/2.7.0/reference/pgql-specification.html, 2018.

[11] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC*, 2013.

[12] R. Madhavan, G. Ramalingam, and K. Vaswani. Modular heap analysis for higher-order programs. In *SAS*, volume LNCS 7460, pages 370–387. Springer, 2012.

[13] R. Mudduluru and M. K. Ramanathan. Efficient incremental static analysis using path abstraction. In *FASE*, volume LNCS 8411, pages 125–139. Springer, 2014.

[14] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, pages 598–608. IEEE, 2015.