PAClab: a Program Analysis Collaboratory

Rebecca Brunner
Robert Dyer
brunner@bgsu.edu
rdyer@bgsu.edu
Bowling Green State University
Ohio, USA

Maria Paquin
Elena Sherman
mariapaquin@u.boisestate.edu
elenasherman@boisestate.edu
Boise State University
Idaho, USA

ABSTRACT

We present a web-based Program Analysis Collaboratory (PAClab) tool that helps researchers to obtain realistic program benchmarks using user-defined selection criteria. Based on selection criteria, PAClab identifies relevant projects and its programs from open-source repositories, obtains those programs, and if necessary performs sound program transformations to adapt them to the targeted verification tool. PAClab makes the resulting program benchmarks available for download. PAClab is designed as a scalable, modular, and parametrizable tool that takes advantage of a computer cluster to handle multiple user requests.

CCS CONCEPTS

• **Computing methodologies** → **Distributed programming languages**; • **Theory of computation** → **Program analysis**.

KEYWORDS

program analysis, program benchmarks, collaboratory

ACM Reference Format:
Rebecca Brunner, Robert Dyer, Maria Paquin, and Elena Sherman. 2020. PAClab: a Program Analysis Collaboratory. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417936

1 INTRODUCTION

Program benchmarks play an important role in program verification research. Properly designed benchmarks can accelerate development of new verification tools, strengthen empirical evaluations of theoretical advancements implemented in existing verification tools, and even compare performances of different verification tools as done in the SV-COMP competition. While synthetic program benchmarks are instrumental to test verification tools [3, 9], researchers tend to prefer realistic benchmarks comprised of real-world programs. Moreover, depending on the specific verification task, those programs should include specific features. For example, a verifier that identifies memory safety of arrays should use benchmarks containing array-manipulating programs. Generally, the definition of an ideal benchmark depends on the verification task, maturity of the verification tools, and the context of evaluations, e.g., testing of new features of a verifier or tuning it for a competition.

With such multi-faceted requirements posed on ideal benchmarks, it is challenging to construct a benchmark from open-source programs, even with the help of mining repository tools. Tools such as Boa [1] or RepoReaper [5] can query large open-source repositories such as GitHub to find projects that contain programs with desirable features, which potentially could become benchmark programs. However, program verification researchers would have to download and build these candidate projects and then determine whether a particular part of a program could be included in a benchmark, and if so then scope the program for a targeted verifier.

Automating real-world application compilation and running static analyses on them is challenging. Ongoing research efforts to address these challenges have resulted in projects such as 50K-C [4] and Hermes [7] that attempt to develop such automated techniques and frameworks. However, while allowing program analysis researchers to submit analyses for evaluation and retrieve the evaluation results, due to their complexity such frameworks limit the researchers' direct access to benchmark source code. For some researchers that might pose a challenge since it is common, especially when developing novel techniques, to reason about the correctness of the analysis results by examining benchmark source code.

Motivated by these challenges, we developed an alternative methodology that is based on the rationale that the code of the entire project might not be relevant or scalable for analyses evaluation. That is, it might just be a few Java class files out of the entire project that have interesting behavior to analyze, explore, and benchmark. Hence, instead of compiling an entire project we attempt to extract relevant programs from it and apply sound transformations to remove dependencies they might have on other project components, while preserving relevant program behaviors. In this case, program analysis researchers can obtain more manageable, easily compilable source code to perform benchmarking. We perceive our approach as complimentary to 50K-C [4] and similar projects, since it provides intermediate steps in the effort to scale evaluations by filling the gap in incremental analysis evaluations between toy programs and complete real-world applications.

We implemented our approach in a web-based Program Analysis Collaboratory (PAClab) tool, which is an instance of our vision for Software Engineering Collaborates [8]. Utilizing a user's criteria for benchmark selections, PAClab **(1)** identifies relevant projects and programs from open-source repositories, **(2)** downloads them from the repositories, **(3)** through a series of source code transformations

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417936

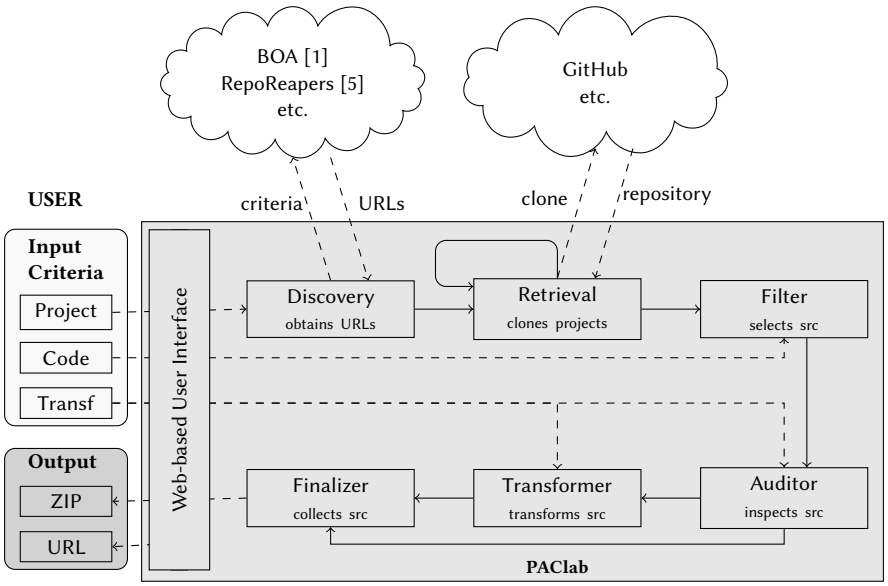

Figure 1: PAClab's architecture and workflow for a user session. Solid arrows represent the direction of PAClab's workflow and dashed arrows represent PAClab's interaction with users and external resources.

adapts them to a targeted verification environment, and **(4)** makes them available for download as a compressed archive file.

While developing PAClab we focused on preserving as much of the original structure of the programs as possible. Thus, PAClab preserves the directory structure of a program, source code identifiers, and program comments. In addition, PAClab allows researchers to disseminate their benchmarks by sharing the benchmark result URL, which besides the benchmarks archive file, also contains the selection criteria used to generate that benchmark. Such preservation of benchmarks' provenance could facilitate reproducibility of evaluations and could make it possible for researchers to report failures found during verification back to the open-source developers.

To summarize PAClab has the following unique features

- It selects programs based on the project-level metadata as well as on source code-level structure.
- It obtains current versions of open-source code and provides provenance of the resulting benchmarks.
- It automatically makes extracted program files compilable outside their host projects.
- It provides users with a web-based UI for cohesive interactions with PAClab's backend components.

In the next section we discuss the tool's architecture and general workflow for using PAClab while highlighting the implementation aspects of its components. We also describe the tool's UI, that allows users to seamlessly interact with PAClab and share their benchmark results. Then we discuss the potential impacts of PAClab on software engineering research and finally we conclude.

2 ARCHITECTURE AND WORKFLOW OF PACLAB

Figure 1 presents an overview of PAClab's architecture and workflow for a single user session. PAClab is composed of a front-end component users interact with and several back-end components. The front-end is a web application that allows users to submit their selection criteria, monitor the session's progress, and obtain the results. PAClab's back-end includes six main components: *Discovery*, *Retrieval*, *Filter*, *Auditor*, *Transformer*, and *Finalizer*. Their sequential actions result in a downloadable compressed archive of program benchmarks and a URL of the session, which are presented to users through the web-based UI.

Workflow Using the UI users submit selection criteria for project metadata, the target code's desirable features, and transformation targets. When users submit criteria, PAClab creates a new project selection session and invokes several backend components and passes appropriate selection criteria to each of them. First, the *Discovery* component uses the project selection criteria to query data sources such as Boa [1], GHTorrent [2], RepoReapers [5], etc., to discover candidate projects matching the project selection criteria and obtain URLs for those projects. Next, using the obtained URLs the *Retrieval* component manages cloning and updating of remote Git projects. The self-loop on *Retrieval* indicates PAClab first checks its local cache before cloning remote repositories.

Once PAClab has source code of all the candidate projects, it invokes the *Filter* component and passes the location of the candidate projects and the user-specified code-specific selection criteria for a method, e.g., the number of expressions of a particular type. *Filter* extracts source code files that satisfy those requirements. Then the *Auditor* component checks whether those files require additional transformations to be suitable for the targeted tool. For example, this component checks whether the extracted program is compilable. If *Auditor* determines that no additional transformations are required, then it writes the source file into the session's benchmark location. Otherwise, PAClab invokes the *Transformer* component and requests specific transformations. Upon completing transformation tasks, *Transformer* verifies resulting source code files are

indeed suitable for the targeted verification task and records them into the session's benchmark location.

The last step in PAClab's workflow archives and compresses the session's benchmark data and via the UI provides it to the user for download. In addition, the URL of a project selection session and resulting benchmark programs can be shared with other researchers, either by directly sharing the session URL or via a box where users provide emails of people to share with.

Implementation The front-end is implemented as an open-source Django Python project[1] running on Ubuntu 16.04 on a quad-core 3Ghz Xeon with 16GB ram. To use PAClab, users visit the website[2] and create accounts. Users are then able to create a project selection. On this page PAClab provides a predefined set of project filters, such as the minimum number of commits or files the project should contain and users can specify values for each filter. These criteria are used to select the initial set of candidate projects.

The back-ends run on 15 identical servers as the front-end. The PAClab back-end servers run several daemons watching for new project selection sessions initiated by users. When PAClab detects a user's request, then it connects to a project data source to request projects matching the user's criteria. The *Retrieval* component is implemented as a Python script that processes Git URLs not already cached on the cluster and clones them locally. PAClab can run this component on multiple back-end servers to ensure scalability of the tool. Note that the backend currently uses Boa [1] to find potential projects matching the criteria. Boa's dataset is from 2015, so the obtained project list does not include projects newer than 2015. It then clones the GitHub projects that still exist and verifies each project's current metadata still passes the selection criteria. Thus, the cloned source code data remains current (not from 2015).

The source code-level back-end components operate on the source-code's abstract syntax tree (AST), which is Eclipse JDT's API for both static code analysis and code manipulation. The *Filter* component[3] makes use of several AST visitors to determine wether each source code files in obtained projects match code-specific selection criteria. Currently, PAClab filters by an expression type, which is currently set to integer types. Since extracted programs are not necessary compilable outside of their projects, PAClab cannot rely on Eclipse JDT to resolve an expression type, and hence, determine whether a particular selection source code criteria are satisfied.

In order to determine expression type *Filter* implements the `InferType` visitor that infers a type of expression e based on its usage context when elements of e cannot be resolved. For example, if e is the right-hand side of a binary expression that has a variable with an integer type on its left-hand side, then the algorithm infers that e's type is also an integer type. Using the results of `InferType`, *Filter* traverses the program's AST to keep methods that satisfy code selection criteria and removes the AST nodes of methods that do not satisfy those criteria. The filtered source code becomes benchmark candidates.

The *Auditor* component inspects each benchmark candidate to determine whether it should be transformed or used in its current form. By default, *Auditor* compiles each benchmark and if compilation fails then the candidate benchmark is marked for transformation, otherwise copies it to the final benchmark location.

Transformer, the program transformation component[3], is currently configured to target an intra-procedural program analysis A over integer expressions for a given class file P that is compilable in a type system Γ_1 with the set of types T_1. P might have several Java classes defined in it, i.e., it defines T_P types. In addition, the evaluation environment for the analysis A has its local type system Γ_2 with the set of types T_2 in which we would like P to be compilable. Thus, $T_1 \cap T_2 \supseteq T_P$, that is, the local type system can resolve the types defined in P.

The transformation algorithm makes use of several AST visitors including `InferType`, which it executes first. The second visitor iterates over P's statements and if it neither can resolve nor infer integer type of one of its expressions then *Transformer* removes such statement because it causes P not to compile and yet does not affect A's reasoning over targeted types. When the visitor cannot resolve an expression but `InferType` infers its type, then *Transformer* substitutes this expression for a pre-defined integer expression. Since an intra-procedural analysis safely over-approximates method invocation or access to fields, we define all those expressions with inter-procedural elements to be evaluated to any abstract value, i.e., $\top$. For the default setting the expression `r.getNextInt()` is used in the substitution and the algorithm also creates `Random r = new Random()` and adds it into the current method's AST. For Symbolic PathFinder (SPF) [6] the algorithm instantiates new symbolic expression `Debug.makeSymbolicInteger("x0")`. Note that the algorithm determines the "smallest" unresolved sub-expressions of e by traversing e's AST in the reverse pre-order.

After this step P has no unresolved expressions, i.e., the local type system can resolve all P's expressions. Next, the algorithm removes unused parameters, fields and variables, updates method's return type (if its type is not in Γ_2). Consequently, all unresolved super-types of classes defined in P are also removed so are unresolved imports. The last step is to add necessary imports such as `java.util.Random` for the default setting and `gov.nasa.jpf.symbc.Debug` for the SPF setting.

Before writing the modified P into the final benchmark location *Transformer* checks whether P is compilable in Γ_2. The compiler errors of uncompilable code are written in a special log that guide the future development of *Transformer*. To our knowledge the ability to *automatically make Java classifies compilable outside of their host project* is a unique capability of PAClab.

PAClab's *Finalizer* component is a Python script on the webserver that locates all transformed candidate projects and generates a single ZIP file. This archive is then available for download via a link on the UI.

User Interface The UI provides a user with the selection criteria page where she can add parameters to the predefined filter criteria for selecting input projects (Fig. 2). The user can use "+" and "-" buttons to add and remove selection criteria filters, respectively.

Upon submission the UI displays the progress of the project discovery and cloning processes together with the project selection criteria. When project selection phase is completed the UI displays information as in Fig. 3, which includes relevant data about this process, including how many projects were discovered (by Boa),

[1] https://github.com/bgsu-pal/paclab-www/
[2] https://paclab.dev/
[3] https://github.com/bgsu-pal/paclab-transformer

Figure 2: Project selection.

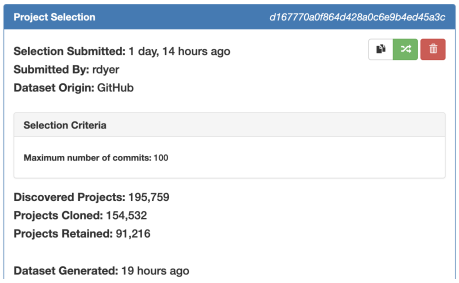

Figure 3: Completed selection.

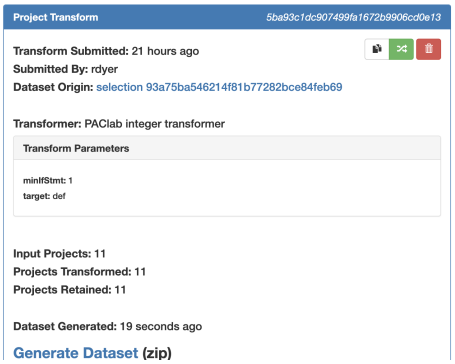

Figure 4: Completed transformation.

how many were cloned, and how many of those cloned projects passed the criteria. While not shown here, if she wishes, the user can download selected projects as a zip file.

Next the UI enables the green button on the top right corner that takes the user to the page to enter code and transformation selection criteria. The current implementation supports integer types (`I`) and three structural code selection criteria: the minimum number of expressions of `I` type, the minimum number of conditionals with predicates over `I` type expressions and the number of arguments with `I` type in a method. Also, there are two targeted transformations: `DEF` (default) and `SPF` (for Symbolic PathFinder). Fig. 4 displays the final UI page containing the code selection and targeted transformation parameters as well as the results of this final phase. The user can download the compilable benchmark dataset as a zip file.

Users can share generated project selections and project transforms via their unique URLs. Also, they can utilize a share box feature to email the links to the specified email addresses.

3 POTENTIAL IMPACT OF PACLAB

Because parametrization is inherent to PAClab's design, the tool can assemble various benchmarks for different verification environments and tasks, hence benefiting verification researchers in several ways. For example, PAClab can supply a scoped set of program benchmarks suitable for testing new verification tools or novel verification techniques of existing tools, thus enabling an agile test-driven development of verification software or a lightweight way to explore feasibilities of new verification ideas. As a tool or a technique becomes more stable, researchers can gradually change PAClab's selection criteria to include additional programs.

We also expect PAClab to significantly improve transparency and reproducibility of empirical evaluations of verification tools. By stating and justifying PAClab's selection criteria used to obtain programs, researchers clearly describe the kind of programs used in their evaluations. However, using the same selection criteria PAClab might produce a different set of benchmark programs due to the dynamic nature of open-source repositories. Hence, to ensure reproducibility, PAClab allows researchers to share the result URL of a session. Then if other researchers want to obtain the same benchmarks they just need to follow the link to download the benchmarks and obtain auxiliary session data.

PAClab also has the potential to strengthen empirical evaluations of verifiers. First, it can produce benchmarks containing large quantities of programs, which would make empirical evaluations more generalizable. Second, the benchmark session metadata can inform researchers about the prevalence of particular types of programs among open-source programs. If PAClab finds only a small fraction of suitable programs, then even if such a benchmark yields a strong empirical study its overall significance might be questioned. Perhaps, it would give an indication that either the verification technique should expand its breadth or it is not particularly useful.

Most importantly, in every scenario PAClab supports provenance of program benchmarks and allows researchers to focus on developing and analyzing verification techniques instead of dedicating their effort to searching for proper program artifacts, which is a subjective and time-consuming task. PAClab also relieves its maintainers from the burden of continued curation of program benchmarks to ensure the latest versions of programs are included.

Presently, PAClab is being introduced to the SPF community, in particular to strengthen empirical evaluations in Google Summer of Code (GSoC) projects. SPF is a popular mature symbolic execution tool which since its first publication in 2007, has around 300 citations. The tool has a large user base: the repository has 40 forks, started by 48 users and is being actively developed.

4 CONCLUSION

Here we presented PAClab, the Program Analysis Collaboratory. PAClab allows researchers to more easily generate benchmark programs for evaluating their program analysis techniques. The framework automates many difficult and time-consuming tasks such as locating and downloading open-source projects matching the user's criteria and adapting those projects through transformations for a given verification environment. These generated benchmarks are easily shared with other researchers, helping facilitate reproductions of prior evaluations.

While our current implementation focuses on Java projects and has limited selection criteria, we are enhancing PAClab's capabilities daily as well as extend its user base.

ACKNOWLEDGMENTS

This work supported by the US National Science Foundation (NSF) under grants CNS-18-23357 and CNS-18-23294.

REFERENCES

[1] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA) *(ICSE '13)*. IEEE Press, 422–431. https://doi.org/10.1109/ICSE.2013.6606588

[2] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. https://doi.org/10.1109/MSR.2013.6624034

[3] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 590–600. https://doi.org/10.1109/ASE.2017.8115669

[4] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A Dataset of Compilable, and Compiled, Java Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR'18)*. ACM, 1–5. https://doi.org/10.1145/3196398.3196450

[5] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (01 Dec 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[6] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

[7] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. 2017. Hermes: Assessment and Creation of Effective Test Corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/3088515.3088523

[8] Elena Sherman and Robert Dyer. 2018. Software Engineering Collaboratories (SECLabs) and Collaboratories As a Service (CaaS). In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 760–764. https://doi.org/10.1145/3236024.3264839

[9] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532