Author/s:
Alatawi, E;Miller, T;Sondergaard, H

Title:
Symbolic execution with invariant inlay: Evaluating the potential

Date:
2018

Citation:
Alatawi, E., Miller, T. & Sondergaard, H. (2018). Symbolic execution with invariant inlay: Evaluating the potential. 25th Australasian Software Engineering Conference (ASWEC), pp.26-30. IEEE Conference Publishing Services. https://doi.org/10.1109/ASWEC.2018.00012.

Persistent Link:
https://hdl.handle.net/11343/241766

2018 25th Australasian Software Engineering Conference (ASWEC)

Symbolic Execution with Invariant Inlay: Evaluating the Potential

Eman Alatawi[1, 2], Tim Miller[1], and Harald Søndergaard[1]

[1]School of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia
[2]Computer Science and Engineering, Taibah University, Almadina Almonawara 42353, Saudi Arabia
ealatawi@student.unimelb.edu.au, {*tmiller, harald*}*@unimelb.edu.au*

***Abstract*—Dynamic symbolic execution (DSE) is a non-standard execution mechanism which, loosely, executes a program symbolically and, simultaneously, on concrete input. DSE is attractive because of several uses in software engineering, including the generation of test data suites with large coverage relative to test suite size. However, DSE struggles in the face of execution path explosion, and is often unable to cover certain kinds of difficult-to-reach program points. Invariant inlay is a technique that aims to improve a DSE tool by interspersing code with invariants, generated automatically using off-the-shelf tools for static program analysis using abstract interpretation. To capitalise fully on a static analyzer, invariant inlay applies certain instrumentations and testability transformations to the program source. In this paper we outline the invariant inlay approach, and how we have evaluated the idea, in order to determine its usefulness for programs with complex control flow.**

I. Introduction

Static analysis and systematic software testing are two fundamental techniques that aim to ensure software quality. Recently, there has been an obvious interest in exploiting the strengths of these two techniques to work together to attain their goals effectively.

Dynamic Symbolic Execution (DSE) [1] is a dynamic analysis technique that systematically explores a program, keeping track of how the inputs forced execution to take the path it took. DSE starts with symbolic inputs that represent possible concrete values, and then execution is carried out by manipulating symbolic expressions rather than concrete values. DSE evaluates both branches of any branching condition, so as to find alternative inputs that will steer the execution to follow an alternative path. In this task, a DSE tool is assisted by a suitable constraint solver. Thus, DSE simultaneously and symbolically follows all possible executable paths in the program with a goal of achieving high *coverage* for some chosen definition of coverage.

In practice, the massive number of program paths to be explored and the number of constraints to be solved hinder scalability of DSE [2]. This problem is referred to as *path explosion*.

Path explosion can be caused by nested calls, loops and conditionals, and particularly input dependent loops (those where the number of iterations depends on an input of the program) [3], [2]. The problem is made worse in the presence of indirect relations between symbolic and non-symbolic variables within input dependent loops and loop dependent branches [4]. This can significantly affect DSE results in terms of code coverage. Existing research has proposed various ways of controlling path explosion, including bounding loop iterations [5], search heuristics to maximize code coverage as quickly as possible [6], function summaries [3], [2], state merging [7], redundant path elimination [8], and identification of skippable functions and code fragments [9]. While these solutions have made the approach practical for several types of applications, path explosion still represents a massive challenge in DSE.

In Alatawi et al [4], we proposed combining DSE with abstract interpretation [10] to tackle path explosion caused by input dependent loops. The idea was to precede DSE with well-known analyses from abstract interpretation based on relational abstract domains. This allowed us to capture indirect control dependencies on the inputs of the program and express these as relational invariants. Then, we would insert these invariants as assumptions before the loop to provide DSE with additional information to help in the handling of input dependent loops and loop dependent branches efficiently. In this paper, we extend that work with a form of *testability transformation* [11] based on the inferred invariants to help DSE to reach a specific target in the program. In the presence of input dependent loops, this enables DSE to reach interesting points in the program that would otherwise not be reached, as symbolic execution easily gets caught inside input dependent loops, making no useful progress.

Testability transformation [11] (or targeted program transformation [12]) is a source-to-source program transformation that improves the testability of a given program by applying some transformations to improve and simplify the process of generating test inputs. The transformed program is used to generate tests that aim to reach a defined target (i.e., assert statement) before it is discarded. After that, the original program is used to be tested against the generated test inputs. Targeted transformations can have a remarkable impact on symbolic execution scalability [12]. We call our approach *invariant inlay*.

To rephrase our goal, we want to incorporate two important program analysis techniques, one static and one dynamic, combining abstract interpretation and dynamic symbolic execution to improve systematic test input generation. More precisely, we explore techniques that help DSE increase coverage, and in

```
x = input;
y = 0;
while (x > 0) {          // Input-dependent loop
    x = x - 1;
    y = y + 2;
}
if (y > limit/2) {       // Loop dependent branch
    error();             // Target
}
```

Fig. 1. Illustrative loop example

particular, make progress in loop intensive programs. We have built a simple proof of concept implementation that supplies a DSE tool with program runtime invariants generated by abstract interpretation.

In this paper we explain the approach and evaluate our particular combination of testability transformation, static program analysis (to provide relational invariants), and dynamic program analysis (DSE).

II. Motivating example

In Figure 1, the termination condition of the loop depends directly on the symbolic input x. In a first round, a DSE tool might explore the branch that does not exercise the body of the while loop. Then it will pick a positive x value. Assuming that an input value $x = 5$ is chosen, the DSE tool will generate as a first path condition: $(x > 0), (x - 1 > 0), (x - 2 > 0), (x - 3 > 0), (x - 4 > 0), (x - 5 > 0)$. Only x, and not y, appears in this path condition because the initial value of y cannot affect the execution path of the program). Then, a new test will be generated by negating each constraint in the path condition leading to generation of 5 tests in this case. In fact, an infinite number of tests can be generated to explore this simple loop because x is unbounded leading to cases in which the DSE algorithms get "stuck" in the loop.

The information available to the DSE tool just before the 'if' statement does not suggest which input x, if any, will reach `error()`. This is because y depends on the number of loop iterations, and y is never made to depend on other variables via assignments. Hence the branch (`y > limit/2`) is *indirectly* dependent on x. This can make it difficult for the DSE tool to find a value of x that makes (`y > limit/2`) true (or false).

Existing DSE tools ensure that execution terminates in a reasonable time by setting a bound on the number of iterations of input-dependent loops. However, setting the bound arbitrarily causes some program paths to remain unexplored. For example, assuming that $limit = 10000$, the *then* branch of the condition (`y > limit/2`) in our example will not be covered within 60 seconds using a well-known state-of-the-art DSE tool, namely KLEE [13]. To cover `error()`, the loop body must be executed more than 2500 times (given the particular value of *limit*), so the loop execution terminates with an appropriate y value. In our experiment, KLEE is able to reach `error()` after 5 minutes and 22 seconds, generating a new test value $x = 2501$.

In the example, whenever the value of x is decremented by 1, the value of y is incremented by 2, so there is a linear

```
x = input;
y = 0;
x0 = x;                  // Instrumentation
cnt = 0;                 // Instrumentation
/* [y=cnt*2, x0=cnt+x] */
while (x > 0) {
    /* [y=cnt*2, x0=cnt+x, x>=1] */
    x = x - 1;
    y = y + 2;
    cnt = cnt + 1;       // Instrumentation
}
/* [y=cnt*2, x0=cnt+x, x<=0] */
if (y > limit/2) {}
    /* [y=cnt*2, x0=cnt+x, x<=0, cnt>limit/4] */
    error();             // Target
}
```

Fig. 2. Simple loop example with invariants

relation between x and y. However, such realtions will not be captured by DSE which only tracks dependencies on program symbolic input x during execution, not relations to or among other non-symbolic variables (such as local variables) of the program under test. More specifically, if x_0 is the initial value of x, and cnt represents the loop counter, then the relation between x and y is captured by the loop invariant $x_0 = cnt + x \wedge y = cnt * 2$. Such relations can be statically and automatically inferred by *abstract interpretation* [10]. For the example we can use convex polyhedra [14] to approximate the set of reachable runtime states. Figure 2 shows the invariants inferred by polyhedral analysis.

Providing such information about relations between program variables and loop iterations to the DSE at the beginning of the symbolic execution can guide DSE to generate a test input that steers execution to reach the desired target. Knowing the loop invariant $x_0 = cnt + x \wedge y = cnt * 2$, together with $x \leq 0$ and the calculated constraint at the targeted line $cnt > limit/4$, the DSE tool can have a constraint solver find a solution, say, $x = 0, x_0 = 2501, cnt = 2501$, and hence produce a successful input value 2501.

III. The Invariant Inlay Approach

Given a program that contains a loop whose number of iterations is *directly or indirectly* dependent on program unbound symbolic inputs, and a target line that is guarded by conditionals that depends indirectly on the program symbolic inputs and the number of times that the loop has been executed, our aim is to generate a test that steers the program execution to reach the target.

The approach consists of generating an over-approximated version of program P using forward and/or backward relational abstract interpretation to capture indirect relations in the program that might be missed by the nature of DSE (such as when there is a control dependency on a non-symbolic variable that will be ignored by DSE). In our use of abstract interpretation, each program point is associated with a relational invariant on numerical program variables. Every run-time program state will satisfy the invariants irrespective of what values input may take, but in general, the invariants will over-approximate the set of possible run-time states. We then

use the inferred invariants in source to source program transformations that result in a loop-free (but over-approximated) version of the input program. This program is then handed to the DSE engine to generate test inputs.

Invariant inlay consists of four steps:

Step 1: Program instrumentation. We introduce two types of variables: Symbolic input initial value holder v_0 for each symbolic variable v_s, and a loop counter c_i for each loop L_i to represent the total number of loop iterations. Then, for each input v, we assign to v_0 the initial symbolic value of v, and increment the loop counter c_i within the loop to explicitly represent the total number of loop iterations.

These new variables are used to prompt the abstract interpreter to explicitly discover symbolic relational invariants using any relational abstract domain. This can help in relating program inputs to loop iterations, and infer any relational invariant between program inputs, loops and other non-symbolic variables that either directly or indirectly are dependent on the program symbolic inputs or a preceding loop.

Using v_0 helps preserve the value of the input v before it might be changed inside P, so at any program point, v_0 represents the initial symbolic input that will lead the execution to reach that point regardless of the current value of the v at that point.

Step 2: Invariant generation. In the second step, we invoke forward abstract interpretation to analyze the the instrument program P', and infer relational invariants using Cousot and Halbwachs's *polyhedral domain* [14]. Polyhedral abstract interpretation generates invariants at each program point which may reveal non-trivial relations among sets of variables, and these relations can be used to strengthen generated path constraints. Every state actually met at a program point during DSE must satisfy the corresponding invariant by definition.

Step 3: Testability transformations. Our approach uses source to source branch-coverage preserving transformations [11]. A testability transformation is not necessarily semantics-preserving. Rather it preserves the sets of test inputs which are adequate according to the branch-coverage criterion. Examples of testability transformations include using merging or splitting loops, induction variable substitution, changing float variables to integer variables, and replacing large constant tables with mathematical formulas [11]. It has been pointed out [12], [15] that effectiveness of DSE can vary considerably, in terms of exploring the program space, between semantically equivalent programs. A program transformation that improves a program's runtime behaviour may simultaneously hamper its symbolic execution. Consequently, testability transformations can improve the scalability of DSE effectively, owing to their impact on both path exploration and constraint solving [12].

Invariant inlay provides the invariants associated with the target as assumptions at the beginning of the program, and then generates a loop summary using inferred loop invariants by abstract interpretation to describe the overall effect of the loop on variable values including symbolic and non-symbolic variables, and loop counters. This is done by assigning to each program variable its value given by an expression over the input variables, their initial symbolic values, and other non-symbolic variables or loop counters if there is a relation between them as indicated by the invariant.

```
:
ASSUME(x0==x+cnt && cnt==y/2 && x<=0 && cnt>2500)
if (x > 0) {
  ASSUME(x0==x+cnt && cnt==y/2 && cnt>=0 && x>=1)
}
else if (x<=0) {
  ASSUME(x0==x+cnt && y=cnt*2)
}
if (y > limit/2) {
  ASSUME(x0==x+cnt && cnt==y/2 && x<=0 && cnt>2500)
  error();           // Target
}
```

Fig. 3. Simple loop example with transformations

The loop summary is represented as an **if** statement that contains information about the entry and exit state of each loop in conjunction with the loop invariant. The loop effect on the program variables is included as assumptions inside the **if** statement. The summary replaces the original loop, providing an over-approximated loop-free version of the original program. This version is used by DSE to generate test data. Figure 3 lists the new transformed version of the example given in Figure 1.

Step 4: Test input generation. The transformed version of the program is then used by a DSE tool to generate tests that aim to reach the defined test target (assert statement). Then this version is discarded, and the original program is used to be tested against the generated test inputs.

IV. Empirical evaluation

The goal of our evaluation is to understand the effectiveness of using abstract interpretation to alleviate the path explosion problem in directed DSE in terms of efficiency (that is, time to reach the target), and target coverage.

Dependent and independent variables. We consider one independent variable which is the test suites origin: We generated two sets of test suites for each program: one using DSE on original programs; and another test suite using transformed programs based on invariant inlay.

We consider the following dependent variables that can be influenced by the type of the generated test suite (i.e., the independent variable): time to reach the defined test target in seconds, invariant generation time, number of paths explored, indicating how invariant inlay affects the path exploration, and whether it has positive or negative impact on path explosion. We also consider the total number of tests generated by DSE until the required target is reached, and finally, the target coverage to indicate whether the required target is reached or not.

Subjects. We have applied our method on a collection of 200 programs, most of which are chosen from the software verification competition (SV-COMP) benchmarks [16] that are widely used in program analysis research. We have added some interesting examples from the literature. We report 15

interesting cases from all of these, of which 14 subjects come from SV-COMP benchmarks, and one essentially taken from Figure 2 in paper [17] (subject 2). We mainly considered subjects from the "loops" category of (SV-COMP) benchmarks. Owing to restrictions imposed by the abstract interpretation tool, we chose only subjects that have integer inputs. Each subject in the SV-COMP benchmarks has a clearly defined outcome outlined in the file name. Some benchmarks have assertions to check some properties, so we have used them as interesting lines for KLEE to reach. However, some benchmark programs have no assertions. Thus, we have used the defined outcomes specified in the benchmark file name to insert an assertion that checks for the property (such as no-overflow or true-termination), to see whether KLEE is able to cover that assertion and generate an appropriate test input leading to that assert statement.

Program instrumentation is done by inserting two types of auxiliary variables, namely initial values holders I_o, and a loop counter c_l for each loop l. The program is then compiled to LLVM bitcode to be used by the analysis tools.

Inferring relational invariants. We use the abstract interpretation tool *PAGAI* [18] to analyze the program, first to check the reachability of the defined target, and then to infer relational invariants over program inputs, auxiliary variables, loop counters, and other local variables. We have configured PAGAI to use polyhedral abstract interpretation, as implemented in the Apron library [19], to infer the relational invariants.

Transformations, and tests generation. In the transformed version of the program, we use the *klee_assume* feature to insert discovered invariants to reach the target at the very beginning of the program to augment the path condition with these new discovered relations. We also transform the loop into an over-approximated conditional statement as per the process explained in section III. Finally, we invoke KLEE on the instrumented subjects to generate test inputs. Given the size of the subjects, we used a maximum testing time of 60 seconds per subject. We configured KLEE to report the total number of generated tests that cover only new paths, and instructed KLEE also to exit once it reaches the target defined as an assert statement.

V. Results and discussion

Table I summarizes the preliminary results of processing a sample of 15 subjects that represent various cases we encountered during our evaluation. *Subject* is the Subject id, *LOC* is the lines of code in the subject, *Target reached?* says whether at least one test covers the target (represented as an assertion), *Time (s)* is the time taken by KLEE to generate a test that reaches the target, *Exec. paths* is the number of execution paths explored, *# Tests* is the number of tests generated by KLEE before reaching the target, and *InvGen Time (s)* is the invariant generation time used by PAGAI.

Invariant inlay helped KLEE reach test targets in subjects (1, 2, 3, 4 and 5) that could not be covered otherwise (KLEE timed out, referred in the table as t/o). These programs contains input directly (or indirectly) dependent loops, loop dependent branches (inside or after the loop body), or nested loops.

In such cases, KLEE might suffer from path explosion, and consume the maximum test time before reaching the target. Thus, invariant inlay was successful in guiding KLEE to reach the target that could not be reached otherwise in the allocated test time. Subject 6 represents the case that even when KLEE is able to reach the target, invariant inlay could speed up that process as a result of the added relational information.

In the subjects 7, 8, 9, 10, 11 and 12, where KLEE was able to reach the target, using invariant inlay *increased* the total testing time (considering also the time of invariant generation). For example, PAGAI spent 195.7 and 175.2 seconds in generating the invariants for subjects 11 and 12, respectively. By inspecting the code, we found that the reason behind the long invariant generation time is the complex branch conditions that are used in these subjects. For example, subject 13 consists of one loop fetching input and sending it to a calculation function that has 131 conditional statements with compound conditions, and the loop is not an input dependent one. Thus, invariant inlay can negatively impact the efficiency of DSE, depending on the abstract interpreter and the abstract domain used. In cases where there is no occurrence of input dependent loops or loop dependent branches (such as when the loop has a fixed number of iterations), this adds analysis time unnecessarily. Many of these cases could be detected with a static, syntactic analysis, thus avoiding invariant inlay altogether.

Invariant inlay did not help KLEE reach the target in the subjects 13, 14 and 15, as PAGAI was unable to generate useful invariants. For example, PAGAI could not generate useful invariants for subject 13 because of code complexity caused by recursive implementation of multiplication by repeated addition. In this case, the generated invariants turned out to be too weak to help KLEE, and KLEE consequently succumbed to the path explosion caused by recursive function calls.

VI. Related work

Saxena et al. [3] propose an approach called Loop-Extended Symbolic Execution that captures how loop-dependent variables are related to the lengths and counts of elements in the program input based on an input grammar by running a separate static analysis. Godefroid and Luchaup [2] use loop summaries to deal with certain types of unbounded loops that include induction variables, whose values are modified by a constant value or constant times for each loop iteration. Loops are summarized by loop pre-conditions and post-conditions that are derived from dynamically inferred partial loop invariants relating the program inputs to the induction variables. In contrast, invariant inlay leverages the strength of abstract interpretation to infer loop invariants, and captures the relations between program symbolic inputs and all program variables purely statically before applying DSE. Symbolic execution is then guided by the inferred relations, and simplified by applying testability transformations.

TABLE I
RESULTS

Subject	LOC	Standard DSE				DSE with Invariant Inlay					
		Target reached?	Time (s)	Exec. paths	# Tests	InvGen Time (s)	Target reached?	Time (s)	Total time (s)	Exec. paths	# Tests
1	23	✘	t/o	1067	2	0.537	✔	0.081	0.618	5	5
2	39	✘	t/o	2	2	0.904	✔	0.053	0.957	2	2
3	38	✘	t/o	361	3	0.324	✔	0.084	0.408	4	4
4	31	✘	t/o	998	2	0.267	✔	0.062	0.329	2	2
5	33	✘	t/o	103	3	6.964	✔	18.794	25.758	43	4
6	24	✔	1.153	19	3	0.77	✔	0.122	0.892	3	3
7	40	✔	0.031	2	2	0.615	✔	0.041	0.656	1	1
8	84	✔	0.086	4	4	1.045	✔	0.042	1.087	4	4
9	56	✔	0.403	43	9	3.03	✔	0.764	3.794	43	9
10	56	✔	0.213	14	6	0.535	✔	0.220	0.755	48	3
11	598	✔	6.705	1500	80	195.7	✔	10.73	206.43	2651	109
12	621	✔	1.507	403	43	175.2	✔	1.402	176.602	417	44
13	50	✘	t/o	2157	5	0.5	✘	t/o	-	2249	5
14	568	✘	t/o	34355	56	19.256	✘	t/o	-	43503	53
15	36	✘	t/o	411	5	1.373	✘	t/o	-	6	5

VII. CONCLUSION

Invariant inlay is a technique that aims to improve DSE by interspersing or replacing code with invariant assertions. Invariants can be generated automatically by off-the-shelf tools for static program analysis. We find that using invariant inlay can *increase* the coverage achieved by DSE when DSE alone cannot cover the test target due to path explosion caused by input dependent loops, and also due to existence of indirect relations between the program inputs and other variables. In such cases, DSE can be guided successfully and efficiently to reach the target by traversing a smaller number of program paths. However, using invariant inlay in cases where DSE might not suffer from path explosion can add unnecessary analysis time. In addition, invariant inlay's capability is limited by the strength of the generated relational invariants using abstract interpreters. In cases where abstract interpreter is unable to generate useful invariants, or cannot handle the program under test, invariant inlay might not support DSE as intended. Our current work is to completely automate the approach, and explore further testability transformations that fit with DSE either in the general or directed form.

ACKNOWLEDGMENTS

The first author gratefully acknowledges support from Taibah University, Saudi Arabia, through a PhD scholarship. The work was also supported by the Australian Research Council through Linkage Grant LP140100437.

REFERENCES

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05)*. ACM, 2005, pp. 213–223.

[2] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proc. 2011 Int. Symp. Software Testing and Analysis (ISSTA'11)*. ACM, 2011, pp. 23–33.

[3] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA'09)*. ACM, 2009, pp. 225–236.

[4] E. Alatawi, H. Søndergaard, and T. Miller, "Leveraging abstract interpretation for efficient dynamic symbolic execution," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Software Engineering*, G. Rosu, M. Di Penta, and T. N. Nguyen, Eds. IEEE Comp. Soc., 2017, pp. 619–624.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[6] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Software Engineering*. IEEE Comp. Soc., 2008, pp. 443–446.

[7] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm SIGPLAN Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[8] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 19–32, 2013.

[9] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proc. 40th Int. Conf. Software Engineering*. ACM, 2018, pp. 350–360.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM Symp. Principles of Programming Languages (POPL'77)*. ACM, 1977, pp. 238–252.

[11] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper, "Testability transformation—program transformation to improve testability," in *Formal Methods and Testing*. Springer, 2008, pp. 320–344.

[12] C. Cadar, "Targeted program transformations for symbolic execution," in *Proc. 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, 2015, pp. 906–909.

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.

[14] P. Cousot and N. Halbwachs, "Automatic discovery of linear constraints among variables of a program," in *Proc. Fifth ACM Symp. Principles of Programming Languages (POPL'78)*. ACM, 1978, pp. 84–97.

[15] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, "Studying the influence of standard compiler optimizations on symbolic execution," in *Proc. 26th IEEE Int. Symp. Software Reliability Engineering*. IEEE Comp. Soc., 2015, pp. 205–215.

[16] D. Beyer, "Competition on software verification (sv-comp)," benchmarks; available at https://sv-comp.sosy-lab.org.

[17] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 529–540, 2014.

[18] J. Henry, D. Monniaux, and M. Moy, "PAGAI: A path sensitve static analyzer," *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 15–25, 2012.

[19] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *Computer Aided Verification (CAV'09)*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 661–667.