Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing

Chu-Pan Wong
Carnegie Mellon University, USA

Jens Meinicke
Carnegie Mellon University, USA
University of Magdeburg, Germany

Christian Kästner
Carnegie Mellon University, USA

ABSTRACT

Generate-and-validate automatic program repair and higher order mutation testing often use search-based techniques to find optimal or good enough solutions in huge search spaces. As search spaces continue to grow, finding solutions that require interactions of multiple changes can become challenging. To tackle the huge search space, we propose to use variational execution. Variational execution has been shown to be effective in exhaustively exploring variations and identifying interactions in a huge but often finite configuration space. The key idea is to encode alternatives in the search space as variations and use variational execution as a black-box technique to generate useful insights so that existing search heuristics can be informed. We show that this idea is promising and identify criteria for problems in which variational execution is a promising tool, which may be useful to identify further applications.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**; **Software testing and debugging**;

KEYWORDS

Variational execution, configurable systems, information flow, automatic program repair, mutation testing

ACM Reference Format:
Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing . In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264837

1 INTRODUCTION

The past decade has seen substantial improvement in automatic program repair and mutation testing. Several approaches in these two areas essentially solve a search problem, in which optimal or near-optimal solutions are sought in a (often huge) search space of candidate solutions, guided by some heuristics and some form of fitness function that distinguishes good solutions from bad solutions. For generate-and-validate automatic program repair, the search problem is to find patches among all possible edits to fix a given buggy program [11, 14]. For mutation testing, a recent search problem was to find interesting combinations of mutants, called strongly subsuming higher order mutants, among possible mutations to increase subtlety, reduce testing effort, and reduce equivalent mutants [8]. Current search-based approaches face scalability issues as the search spaces continue to grow. Search-based automatic program repair and higher order mutation testing have a search space of many possible edits/mutations and an exponential number of combinations, a search space so large that is impossible to exhaustively explore each variant separately.

In this work, we propose a novel integration of *variational execution* [13, 16, 25] into search-based automatic program repair and higher order mutation testing. Variational execution has been designed for configuration testing of highly configurable systems and for dynamic information flow tracking in privacy-sensitive systems, both of which demonstrate that variational execution is effective in exploring variations. More importantly, variational execution enables fine-grained observations of *interactions* among variations, while sharing commonalities to efficiently explore even exponential search spaces in many practical settings. At a high level, our approach encodes patch candidates and mutants as program variations and uses variational execution as a black-box execution engine to accelerate evaluations of fitness and derive useful insights for a more efficient navigation of the search space.

To gauge the potential of this work, we carefully analyze key elements that lead to successful applications of variational execution and show that those elements manifest in search-based automatic program repair and higher order mutation testing. The analysis of potential gives us confidence that this direction is promising. We envision that similar applications could be beneficial for other search-based problems, as long as they exhibit all the key elements of applying variational execution. We hope that this work can provide a new perspective of improving automatic program repair, mutation testing, and other related areas.

2 VARIATIONAL EXECUTION AND EXISTING APPLICATIONS

We first introduce essential concepts of variational execution and two successful applications: configuration testing [16] and information flow tracking [2]. We then derive key elements that enable promising applications of variational execution, which are later used to predict potential of this work.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264837

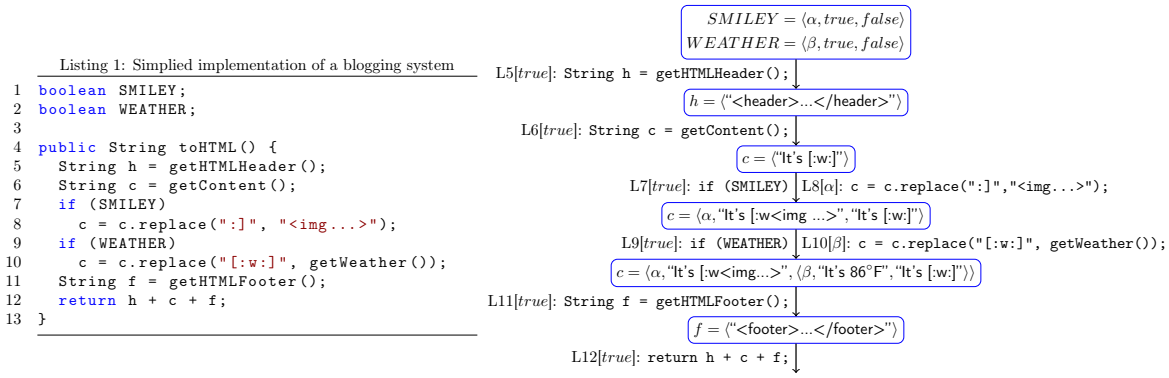

Figure 1: An example showing how variational execution is used for configuration testing, modeled after WordPress [13]. Listing 1 on the left shows the source code. The graph on the right shows the execution trace of variational execution.

Variational execution is a dynamic analysis technique that exploits sharing among similar executions with minor differences [13, 16, 25]. The idea is similar to symbolic execution [10] in that concrete values are replaced with abstract values that represent many possible concrete values. While *symbolic values* in symbolic execution usually represent all possible values of a given type, *conditional values* in variational execution represent a finite and often small set of alternative concrete values, each of which is distinguished by a propositional formula that shows the condition under which this concrete value exists. Conditional values are typically expressed as possibly-nested choices over propositional formulas, such as $int\ x = \langle \alpha, \langle \neg\beta \vee \gamma, 1, 3\rangle, 2\rangle$, which means x has the value 1 if $\alpha \wedge (\neg\beta \vee \gamma)$, 3 if $\alpha \wedge \neg(\neg\beta \vee \gamma)$, and 2 if $\neg\alpha$. Note that concrete values (e.g., 1, 2, 3) and symbolic conditions (e.g., α, β, γ) do not intermix. Operations on a conditional value are guarded by *variability contexts*, which limit the set of concrete values that are affected inside the conditional values. The concept of variability contexts is analogous to path condition in symbolic execution. Computations on conditional values are repeated on all alternative concrete values in the corresponding contexts.

There exist different implementations of variational execution [2, 3, 13, 16, 22, 25]. Current implementations have nontrivial overhead because they need to track conditions and conditional values at runtime, but this overhead can be often justified when exploring very large configuration spaces. We use VarexC [25], a state-of-the-art implementation based on Java bytecode transformation.

2.1 Configuration Testing

Computer programs often come with variations that adjust functionalities on demand, in the form of command-line options, plugins, or extensions. These variations are often called *features* or *options*. Features offer great flexibility, but also incur risk of feature interaction problems [4, 18], where one feature interferes with another when used together. To detect feature interactions, Nguyen et al. [16] applied variational execution to test WordPress with different combinations of 50 plugins, yielding 2^{50} different configurations. Their results show that variational execution can analyze the huge configuration space efficiently and exhaustively and identify a previously unknown feature interaction bug.

Figure 1 illustrates how variational execution is used for configuration testing. The code listing on the left shows a simplified implementation of a blogging system [13]. In the current implementation, there is an issue: if both `SMILEY` and `WEATHER` are enabled, the replacement of smiley image takes precedence and breaks the expansion of weather information, resulting in outputs like "[:w😊". The execution trace on the right shows how variational execution would detect this unexpected feature conflict. Each execution step represents execution of one line along with the variability context. Updated program states are shown in rounded boxes. Notice the sharing of executions before Line 7 and after Line 10.

With variational execution, we can spot the problematic interaction by looking at the content of `c`. In fact, all possible interactions are recorded and detectable by inspecting conditional values. Without variational execution, we would need to run the same program for 4 times in order to ensure absence of feature conflicts like this. Brute-force testing of all configurations does not scale when the number of features is large, but variational execution can efficiently execute the program once and record all possible interactions of features if there is sufficient sharing among executions.

2.2 Information Flow Tracking

Information leaks in security-focused systems have gained substantial attention recently, especially leaks that are caused by subtle implicit information flow. Dynamic information flow struggles with implicit flows, especially from paths that are *not* executed [1, 5]. Austin and Flanagan [2] proposed a form of variational execution to track information flows precisely, called *faceted execution*, which separates the executions of high confidentiality (denoted as H) and low confidentiality (denoted as L) so that sensitive information does not flow from H to L. The key idea is to compress information of both H and L into a *conditional value*. When H and L have the same value, the executions are shared to reduce overhead. When H and L have different values, both values are accessed or updated according to some security-preserving semantics. Multiple principles (i.e.,

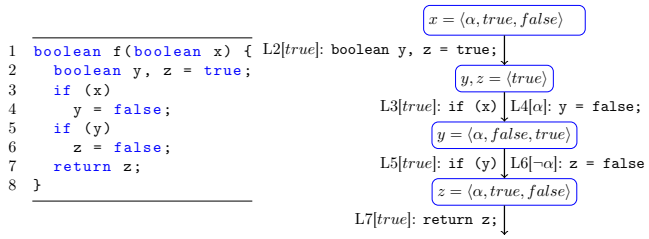

Figure 2: An example illustrating how variational execution can be used to handle implicit information flow.

multiple pairs of H and L) are also supported and their interactions are explored at runtime. This line of work was later extended to support different languages and database systems [3, 21, 22, 26].

Figure 2 shows an example of how to use variational execution to protect sensitive data. The goal is to hide the secret value of x from public observers. As we can see, x is initialized with a conditional value so that private observers see its real value *true* and public observers see a different value. Using variational execution, values of H and L are separated safely. Finally, public and private observers see different values of z, so that the secret value of x is protected.

2.3 Key to Successful Applications

Studies have shown that variational execution is useful for configuration testing and information flow tracking, but we expect more application areas, where this specific flavor of sharing computations with multiple concrete values is effective to explore large spaces. By comparing the above-mentioned applications and their limitations, we derive three key characteristics from the application domains that determine applicability of variational execution.

Finite Variations. The problem domain should have many but finite variations of interest to begin with. In configuration testing and information flow tracking, variations are different features and different privacy principles, respectively.

Interactions. Conditional values are especially useful for exploring interactions among variations at runtime. The overhead of variational execution is easier to justify when exhaustively exploring an exponential search space of all combinations of multiple variations. In configuration testing, developers are interested in the interaction of multiple options; In information flow tracking, interactions among multiple principles need to be tracked soundly to avoid leaking sensitive information in unexpected ways.

Sharing. Variational execution is effective if there is substantial sharing among executions of different variations and their interactions. Variational execution stores and computes all concrete values for all configurations, but it exploits shared values and shared operations to reduce overhead so that it can explore an exponentially large configuration space. If there is no sharing at all among executions, variational execution suffers from the same combinatorial explosion as a brute-force strategy. However, studies have shown that sharing is very common in practice for testing [13, 20]. In information flow tracking, sharing is common in parts that are not affected by principles. Interactions are common, but not among all options at all times [13, 20]. That is, variational execution is effective in large search spaces when interactions among multiple variations are important but not all variations interact on all computations.

3 PROMISING APPLICATIONS

Beyond configuration testing and information flow tracking, we suspect that variational execution is useful for many other applications. In this section, we discuss the potential of applying variational execution to search-based *automatic program repair* and *higher order mutation testing* and explain why we expect that they will benefit from variational execution.

3.1 Automatic Program Repair

Automatic program repair has gained a lot of attention in recent years [14]. One of the most representative search-based approaches is GenProg [24], a technique that uses genetic programming to search for a patch that passes all provided test cases. We suggest applying variational execution to such search-based approaches.

GenProg takes as input a buggy program and a test suite that reveals the bugs. Then it runs a genetic programming loop iteratively to find a patch. Each loop iteration is called a generation, mimicking biological evolutions in genetic programming. In each generation, a new set of patch candidates is generated, each of which often changes *one statement* of the original buggy program. Then, for each patch candidate, GenProg applies it and calculates a fitness value by running the test suite. Based on fitness values, GenProg discards a subset of patch candidates and then moves on to the next generation, unless a patch is found or the resource limit is reached.

Approach. Our idea is to encode multiple patch candidates as boolean options in the program, and then evaluate them and their combinations altogether by executing the test suite *once* with variational execution. The following code snippet illustrates how variational execution can inform generation of multi-edit patches. This example is extracted from a real bug fix from the Defects4J dataset [9]. Two changes are required to pass all the test cases: change1 is necessary to pass test cases t_1, t_2 and t_3, and change2 is required to pass t_3. Traditional search-based approaches might miss change2 because applying it alone does not pass any failing test cases. The root cause to this problem is that fitness function defined as the number of passing test cases is a poor proxy of partial bug fix, and thus combinations of patches are rarely explored in targeted fashion. However, with variational execution, the combination of these two changes (along with many other patch candidates) can be explored and tracked while executing the test suite. Since variational execution provides a bigger picture of patch candidates, it can inform the search of valuable combinations of changes, such as change1 and change2 in our example.

```
1  public Complex divide(Complex divisor) {
2    //..
3    if (change1) return NaN;  // actual fix
4    else return isZero ? NaN : INF; // buggy, see MATH-657
5    //..
6  }
7  public Complex divide(double divisor) {
8    //..
9    if (change2) return NaN;  // actual fix
10   else return isZero ? NaN : INF; // buggy, see MATH-657
11   //..
12 }
```

Characteristics. Patch search is a promising application because it has all key enablers of variational execution. **Variations** are patch candidates that modify a tiny part of the program. Oftentimes a lot of patch candidates are generated. **Interactions** of patch candidates are important to observe because they might provide insights of synthesizing multi-edit patches, which is still an open challenge. Researchers have empirically shown that more than 70 % of bug fixes in practice require more than two repair actions [27]. Using variational execution, we could determine that multiple patch candidates are required to pass a test suite. **Sharing** is very likely because of two reasons. On the one hand, patch candidates are generated independently, and thus often modify unrelated states of the program. On the other hand, the whole test suite is invoked again and again to calculate fitness, causing a lot of redundancy in executing test cases. Since each test case often tests a small part of the program, it is likely that each test case only touches on a small number of patch candidates. With the potentially abundant sharing, variational execution might be able to speed up the evolving cycles of genetic programming, especially with regard to exploring combinations of patches. As a side benefit, we can also inspect how patches affect value differences at runtime and use the insights to guide the search of more promising patch candidates.

Related Work. Different techniques have been proposed to speed up the process of finding patches or improve the quality of generated patches [14]. Our application of variational execution is orthogonal to most recent advances in automatic program repair, as none of the existing work tries to replace the execution engine used for evaluating fitness. There exist other techniques like synthesis-based program repair [12, 15, 19], but we envision that search-based approaches benefit more directly from variational execution.

3.2 Higher Order Mutation Testing

Higher order mutation testing is a technique that seeks valuable combinations of mutants to generate higher order mutants that are likely to denote more subtle faults. Since the set of candidate combinations of mutants is exponentially large, Jia et al. proposed to use search-based optimization techniques like genetic programming to search valuable higher order mutants iteratively [8]. In each iteration, several mutants are randomly combined and evaluated using a given test suite. Iterations go on until valuable mutant combinations like strongly subsuming higher order mutants are found, or resource limit is reached. The goal is to search for combinations of mutants that can only be killed by a subset of test cases that kill the constituent mutants. This way, higher order mutants can replace the constituent mutants without affecting the accuracy in assessing the quality of the test suite, but with fewer executions of tests.

Approach. Our idea is to encode all mutants as options, and use variational execution to explore all mutants at the same time. Since first order mutants usually modify programs at the expression level, changes can be encoded using ternary operators, as shown in the code snippet below where 4 mutants are encoded. We can then derive useful higher order mutants by transforming specifications of higher order mutants into a satisfiability problem and use SAT solvers or BDDs to get solutions. This idea has been explored recently and the results are promising: the approach based on variational execution generates a more complete set of strongly subsuming higher order mutants using an order of magnitude less time when compared to the state-of-the-art on Triangle, the classic example in mutation testing literature [6].

```
1    //..
2    if ((m1 ? (a != b) : (a == b))) {
3      trian = (m2 ? (trian - 1) : (trian + 1));
4    }
5    if ((m3 ? (a != c) : (a == c))) {
6      trian = (m4 ? (trian - 2) : (trian + 2));
7    }
8    //..
```

Characteristics. Higher order mutation testing has all the key enablers of variational execution. **Variations** are used to encode mutants, so we can easily get many variations by generating mutants randomly. **Sharing** is very likely due to the random generation of mutants and local effect of many mutants. With variational execution, we can execute the test suite once and observe the effect of all mutants, avoiding repeated executions of the same test suite. **Interactions** are interesting to inspect because we could detect valuable higher order mutants that are much harder to kill than its constituent mutants. Moreover, we can identify equivalent mutants by inspecting mutant interactions to further reduce testing effort.

Related Work. Jia et al. [8] proposed to generate higher-order mutants using search strategies such as genetic algorithms and greedy algorithms. However, the search process is expensive and ineffective. Wang et al. [23] proposed to reduce redundant evaluations of mutants by identifying equivalent states and compressing executions of those mutants into the same process. Using variational execution, we could achieve a more fine-grained sharing of mutant executions. Heymans et al. [7] proposed a conceptually similar mutation analysis, but it only works on models and exploits only prefix sharing.

4 CONCLUSION

Variational execution has been independently shown to be useful in applications like configuration testing and information flow tracking, but has rarely been explored beyond, with the exception of an outline for detecting semantic merge conflicts [17]. By analyzing successful applications, we derive three key characteristics that determine effectiveness of variational execution: variations, sharing, and interactions. Using these elements, we discuss the potential of applying variational execution to two new domains: automatic program repair and higher order mutation testing. We hope that this work can stimulate research on automatic program repair, higher order mutation testing, and other domains that face similar challenges.

ACKNOWLEDGMENTS

This work has been supported in part by the NSF (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). We thank Eduardo Figueiredo, João Paulo de Freitas Diniz, and Serena Chen for exploring early ideas and prototypes. We also thank Claire Le Goues and Yingfei Xiong for early discussions of this work.

REFERENCES

[1] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 113–124.

[2] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 165–178.

[3] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 15–26.

[4] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (Jan. 2003), 115–141.

[5] Deepak Chandra and Michael Franz. 2007. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. IEEE, 463–475.

[6] Serena Chen. 2018. *Finding Higher Order Mutants Using Variational Execution*. Technical Report 1809.04563. arXiv. Accepted to SPLASH'18 Student Research Competition.

[7] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-Based Mutation Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 655–666.

[8] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (Oct. 2009), 1379–1393.

[9] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 437–440.

[10] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[11] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (Sept. 2013), 421–443.

[12] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 691–701.

[13] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 483–494.

[14] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2018), 1–24.

[15] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[16] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 907–918.

[17] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N. Nguyen. 2015. Detecting Semantic Merge Conflicts with Variability-Aware Execution. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 926–929.

[18] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. 2008. Feature Interaction: The Security Threat from within Software Systems. *Progress in Informatics* 5 (March 2008), 75.

[19] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449.

[20] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 445–454.

[21] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

[22] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the International Conference on Principles of Security and Trust*. Springer, 3–23.

[23] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence Modulo States. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 295–306.

[24] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 364–374.

[25] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, OOPSLA (Nov. 2018). Article 117.

[26] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 631–647.

[27] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 913–923.