Automated Testing of Database Driven Applications

PhD Thesis Dissertation

Maryam Abdul Ghafoor

2015-03-0036

Advisor: Dr. Junaid Haroon Siddiqui

Department of Computer Science

School of Science and Engineering

Lahore University of Management Sciences

December, 2019

Dedicated to the faults in programs that make the possible; impossible.

Hence making THIS WORK possible!

Acknowledgements

Someone said, and I quote, ”Diamond is just a piece of coal pressed hard under the ground for a long time.” Yes! It’s all about the journey of coal turning into a diamond. What a journey it was. My journey of Ph.D. was as straight as a sine wave that started with fuel of hope, motivation, and aspirations. There were peaks of achievements gained through constant hard work and also troughs of dead-ends in research apparently leading nowhere but then followed by the momentum of guidance of my advisor that takes a new twist and turns to continue this fantastic journey. What a test of patience it was!

First and foremost, I would like to thank the One, who gave me strength when I was weak, courage when I was unsure of my circumstances and showed me the way even in the darkest night of my life. Thank you, Allah.

I would like to express sincere gratitude and special thanks to my advisor, Dr. Junaid Haroon Siddiqui. I consider my self to be extremely privileged to have been part of his research group. You have been a tremendous mentor. You inspired me to start this journey. Your immense knowledge, valuable guidance, and constant effort assisted me in my research. Your dedication, enthusiasm, and steadfast attitude taught me many lessons and helped me grow into a better person. Your words of encouragement proved to be a driving force for me and paved my way to success. Thank you, sir!

I am grateful to my committee members for their guidance and for making my defense a memorable experience. Your discussion and feedback have been invaluable. Many thanks to Dr. Basit Shafiq and Dr. Fareed Zaffar. You were not only part of my committee, but I also learned a lot from you during courses that I have taken from you. I would also like to thank Dr. Zubair Malik and Dr. Zartash Afzal Uzmi for kindly agreeing to be part of my defense committee. Thank you, Dr. Zubair Malik, for your support and useful insight in the field and managing to attend my defense till late hours. Thank you, Dr. Zartash Afzal, for being part of my final defense committee for

taking the time to read my thesis and inspiring me in many other ways.

I am much thankful to all my teachers from whom I have taken courses and done cross-domain research. I extend my gratitude to Dr. Hamad Alizai and Dr. Imdad Ullah Khan for giving me the opportunity to apply program analysis techniques in IoT and Big Data domain. Applying program analysis techniques in these areas opened a new horizon of research for me. I would also like to thank Dr. Suleman Shahid, for mentoring me in my HCI project. It also helped me in my research abroad. During my Ph.D., I also worked with Dr. Patty Kostkova in University College London, UK. I am thankful to her for great support and an enjoyable experience.

Many thanks go to my lab fellows for all the fruitful discussions and consolation when I needed them the most. Special thanks to my extraordinary friend, Atira Anum, for having those short breaks, walks, snacks, and all long discussions, which opened my mind and always ended in coming up with new ideas. Thanks, Affan Rauf and Sohaib Ayub, for all help that you extended. PAGLUMS is a second home to me, and I will never forget the time that I spent here with you guys. Dozens of people, including my seniors, helped me in various ways in the computer science department. Thank you all. Especially Dr. Ayesha Afzal, for all your moral support that you always extended and Aneesa Abbasi for making project a memorable experience. I genuinely appreciate it.

I am grateful to my parents; this would not have been possible without their constant prayers, endless support, and warm love. What I am it runs in the family, Ammi Jan. I can never thank you enough. From praying hands to taking care of my children in my absence. Thank you, Abbu Jan, for supporting me in every decision that I made.

I am thankful to my siblings, FAMMAAS, for all their wise words, advice that they provided me throughout this incredible journey. Special thanks to baji (Amna) for all of the incredible strength that you have forced me to see in myself. For making me realize that I am strong and capable of doing anything I set my heart to. Api (Fatima) for making me have faith in Almighty that He never disappoints praying heart, and I will get whatever He has for me. Apa (Mumtaz), for believing in me and guiding me towards emotional and financial independence by showing me the ocean of opportunities and possibilities whenever I felt stray. Bhai(M Abdullah), for making

me laugh on the toughest day of my life. I would not have achieved this without your wonderful support and confidence in me. Many thanks, Ash(Ayesha), for being able to listen whenever I needed someone to talk to – your young, jolly presence used to freshen up my days. Lastly, Emaan(Saima) for setting an example for me. Since our childhood, we shared "All for one and one for all." You folks proved it right.

....and Yes, here comes a time, at last, I had not to listen anymore from my relatives, friends (near or far) "When is your Ph.D. going to end?" I appreciate their consistency in questioning (built-in feature of Asians) that pushed me harder towards my goal, sometimes setting me back (that I always took as other end of sine wave of this journey).

Infinite thanks to my children– the most basic source of my life energy. They made me stronger, better, and more fulfilled than I could have ever imagined. I love you to the infinity and beyond. Taimur and Hadee, my champions — for their never-ending support to make sure I never lose track of hopes and dreams. Sakeenah - your laughter and love made me keep smiling even when I was dead stressed. Ibrahim and Haleemah, my two little angels who came in this world during this journey. Ibrahim (micro as named by your father), you are a miracle, little games that we played together used to make my day. Haleemah – my baby, my sunshine, for your company on the keyboard and making sure that I complete tasks by staying up late till early morning. Thanks for walking this journey with me. We share an amazing bond, and you all are my rock. Proud to be the mother of such a wonderful bunch!

Lastly, Salman, I would not have achieved this without your love, caring gesture, and voice of concern. Thanks for being my support till this day.

Journal

1. Maryam Abdul Ghafoor, Muhammad Suleman Mahmood, Junaid Haroon Siddiqui, ”Extending Symbolic Execution for automated testing of stored procedures”. Software Quality Journal, 2019.

Conference

1. Maryam Abdul Ghafoor, Junaid Haroon Siddiqui,Automated testing of NoSQL Database applications. (Submitted in 2020 IEEE International Conference on Software Testing, Verification and Validation)

2. Maryam Abdul Ghafoor, Muhammad Suleman Mahmood, Junaid Haroon Siddiqui, ”Effective Partial Order Reduction in Model Checking Database Applications”. IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016.

3. Muhammad Suleman Mahmood, Maryam Abdul Ghafoor, Junaid Haroon Siddiqui, Symbolic Execution of stored procedures in database Management System. 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016.

4. Maryam Abdul Ghafoor, Junaid Haroon Siddiqui, Cross Platform Bug Correlation Using Stack Traces. 2016 International Conference on Frontiers of Information Technology (FIT)

Abstract

The testing of database-driven applications is crucial for the correct working of the system. Despite the progress in the field of automated software testing, its usage in database-driven applications remains a difficult task. A test case written for these applications includes database state in addition to the standard test inputs. Moreover, database-driven applications are conjunction of imperative and declarative languages. The interplay between these languages has imposed new challenges on automated testing of database-driven applications. In today's data-driven world, databases and data inside them power business decisions. Recent advancements in technology have opened many paradigms for enterprise applications to store, manage, and retrieve information from the databases.

In this thesis, we propose that different program analysis techniques can be employed to test the functionality of database-driven applications. Dynamic execution is a state-of-art program analysis technique that works by executing the application under test and provides high code coverage. We present a testing technique that adapts dynamic symbolic execution to model databases and database systems for SQL based applications. We also employ model checking well-known program analysis technique in SQL based concurrent applications to detect schedules that lead to an inconsistent database state. Our hybrid approach of using dynamic symbolic execution along with model checking can test NoSQL based database-driven applications.

To assess the effectiveness of using dynamic symbolic execution on database-driven applications, we evaluated our technique on 500 procedures from online projects. Our results show that our methods successfully model databases for both SQL and NoSQL applications. Furthermore, it generates around 350 test cases that trigger constraint violations or hit user-defined exceptions. In the case of concurrent applications, our approach gives 1.4x reduction in state space of program often providing a 2.8x speedup for verification.

To conclude, our proposed approach for testing database application provides full path coverage along with thorough testing, and for multi-threaded applications, it efficiently explores state space of the program.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

An increase in the popularity of the Internet has made use of data ubiquitous. Data is everywhere and is growing exponentially. Vast amounts of data is stored in databases for analysis. Databases and the data inside them are pervasive and used by numerous applications to empower business decisions. A large number of enterprise applications are designed around databases. These data-driven applications store their data in a Database Management System (DBMS). This ensures data integrity as well as efficient accessibility of data. Many organizations build database-driven applications around databases to make business decisions by performing logical operations on data stored in them. The correctness of these applications is crucial for the smooth working of the system. This requires thorough testing. Various software testing techniques are used to ensure the correctness of database-driven applications. Testing can be done manually but it is expensive and inefficient. On the contrary, not only is automated software testing cost-effective, it also has some inherent advantages such as increased efficiency, coverage, and reduced human resources. It exhaustively tests all program paths with a set of possible inputs.

Recent advancements in technology have opened many paradigms to store, manage, and retrieve information from databases. Most off-the-shelf applications that manage enterprise data use Relational Database Management System (RDBMS) to store critical data because of its transactional semantics, support for business operations, data-ware housing, analysis and reporting capabilities [3]. In such applications, data is scaled vertically, whereas nature and type of data remain

the same. This structured data is stored in the schema and the relationship between data is defined through constraints. Database-driven applications in a distributed environment are highly concurrent applications that use a centralized database where a single database server serves multiple client applications. Millions of clients access these database systems simultaneously, putting the consistency of database state at stake.

Tremendous increase in applications requiring semi-structured data posed new challenges for traditional RDBMS. Such applications rely on flexible data models. Modern developments in databases led to scalable databases that work well with unstructured data, such as NoSQL based databases, which slowly gained popularity due to their ability to collect large amounts of loosely structured data. These databases are schema free and allow horizontal scaling of data. Moreover, they have a flexible data model that can be resized at any point in the life cycle of software applications.

1.1 Problem description: Testing database-driven applications

Database-driven applications heavily interact with database systems. Such applications usually are a combination of imperative and declarative languages. The majority of the testing techniques are designed for imperative languages and are not directly applicable to database-driven applications. However, some automated testing techniques ensure the correctness of database applications, a critical requirement for their correct working. In database applications, program inputs, queries, database states, program variables, and branch conditions are all closely related and often depend on each other. For example, program input variables are frequently used to formulate queries.

Similarly, branch conditions involve results retrieved from database operations. This dependency leads to many possible combinations of database state and program input. A test case for such applications includes database state as well as usual program input. As database-driven applications treat DBMS as an external system, their testing is itself a challenge. Symbolic execution is one of the several approaches that exist for automated test case generation. It works by gathering constraints along a path and then solves them to get the test case. It is considered to provide increased code coverage for white-box structural testing of the application under test (AUT).

Many SQL based applications store business logic for common tasks in stored procedures. Stored procedures are modular programs [4] that can have database accesses in addition to business logic. Stored procedures reside on the server and they offer faster execution and reduced network traffic, especially when called frequently. Effective performance of stored procedures requires exhaustive testing.

Symbolic execution has been used with database-driven applications [5–9]. These techniques are not directly applicable to the testing of stored procedures for two reasons. Firstly, because of the challenges posed by the multi-lingual nature of these applications (SQL and some imperative language) and secondly, these applications consider the database as an external system. There are two research schemes in practice for the symbolic execution of such code. The first scheme transforms the AUT into classical program code by replacing SQL queries with variables to represent database content and then modifying statements of AUT to act on these variables. This allows direct application of conventional symbolic execution on program code. The other scheme applies symbolic execution on the application code directly. For this, [10] uses the results of the queries to generate test input (database relations) for program code by constraining relations within path conditions. These path conditions are then solved to generate test input data. Both schemes rely on either program code or program specification [2] for extracting language and database integrity constraint required to form path condition.

Database-driven applications are also used in a distributed environment where they depend on a centralized database. These applications are highly concurrent and allow multiple clients to access data at the same time. Order of execution of clients' requests may result in a different database state. Verification of highly concurrent applications requires modeling all possible orders to uncover schedules that lead in an inconsistent database state. Model checking is a program analysis technique used for bounded exhaustive testing of concurrent applications [11–13], but it suffers from state space explosion, especially while dealing with large or concurrent applications. This gives rise to the need for testing technique that allows model checking for testing of concurrent applications while reducing the exploration of state space of a program.

An increase in data has led to the use of NoSQL databases. Much work has already been

done in the domain of database-driven applications accessing data using standard query language's data manipulation commands, whereas NoSQL applications still require the researcher's attention. Though tools for unit testing of NoSQL applications exist, they rely on the information provided by the user in the form of specification or annotations.

1.2 Our solution

We use program analysis techniques to model database applications and database systems to enable efficient verification of database management systems and the applications using them, resulting in higher reliability and improved software quality. In this work, we propose and adapt explicit-state bounded model checking and symbolic execution to find bugs in business logic in database applications.

We adapt symbolic execution to automate test case generation of stored procedures. We instrument internal query nodes of the database to gather language and internal database constraints directly from within the database, which makes precise and efficient constraint collection possible. This enables our technique to execute the original code without transition so that it directly deals with program variables and database content symbolically while preserving their interaction with each other. We also use model checking for testing concurrent database-driven applications to find schedules that may result in an inconsistent database. We combine symbolic execution with model checking to test NoSQL based applications. This enables us to treat language constraints as well as database internal constraints as a choice point.

We have been able to analyze complex business logic stored in the form of stored procedures using symbolic execution. Symbolic execution is used in many paradigms but was not previously used in the verification of stored procedures. Our implementation of a symbolic executor generates test cases for stored procedure along with the database state. Furthermore, we model accesses to the database as a multi-threaded application and form a technique that reduces the number of schedules that a test case must exercise. Our technique EPOR (Effective Partial Order Reduction in model checking database applications) employs partial order reduction to explore the state space of the program in less time, as shown in the experimental evaluation.

1.3 Contribution

In our work we made following contributions

1.3.1 Symbolic execution of SQL based applications

- **Symbolic execution of stored procedures:** We demonstrate an end-to-end technique for symbolic execution of stored procedures in PostgreSQL.
- **Instrumentation at query node level:** Our instrumentation at the level of query nodes allows more precision and efficiency as compared to any other technique at the abstraction of a high-level query language like SQL.
- **Constraint collection from query nodes:** We identified and collected constraints from three sources using query node instrumentation.
- **Support for joins:** We support database joins allowing better coverage.
- **Handling of aggregates and complex assignments:** We handle aggregate functions and complex assignments that are often used in taking decisions in business logic.
- **Estimation of table rows:** We estimate number of rows to be modeled which enables us to test significant number of scenarios.

1.3.2 Model checking of SQL based applications

- **More Precise Partial Order Reduction.** Our Effective Partial Order Reduction technique is more precise as it finds dependencies among queries by executing them and observing them inside the database.
- **Instrumentation of PostgreSQL** We instrumented PostgreSQL query nodes to extract unique row identifiers to perform precise dependency analysis.

1.3.3 Combining symbolic execution and model checking for NoSQL based applications

- **Modeling collections of MongoDB:** We model collections of database to contain symbolic documents.
- **Modeling NoSQL based database accesses:** We modeled NoSQL based database accesses to interact with symbolic database. We provide full path coverage of the program code by making database access a choice point.
- **Symbolic execution of NoSQL based applications:** We adapted symbolic execution for testing of NoSQL based database-driven applications.

1.3.4 Implementation and experimentation

We make following contribution in implementing and evaluation of aforementioned approaches.

- **Implementation** We provide implementation of our techniques. We implemented symbolic execution of stored procedures on PostgreSQL [1] (an open-source database with support for stored procedures) using the Z3 SMT solver [14]. We also implement our technique using JPF [2] – an open source model checker– for partial order reduction of Java based database applications using PostgreSQL at back-end. We provide implementation of symbolic execution of NoSQL databases using JPF-Symbc [3] for test case generation of NoSQL based database applications using MongoDB [4] at the back end.
- **Evaluation:** We evaluated our technique on 500 procedures that are part of some open source, publically available on-line projects and an Accounting and CRM ERP, PostBooks [5]. Our symbolic executor has generated around 350 cases that trigger constraint violations or

[1] `http://www.postgresql.org`
[2] `javapathfinder.sourceforge.net/home.html`
[3] `https://github.com/SymbolicPathFinder/jpf-symbc`
[4] `https://www.mongodb.com`
[5] `http://www.xtuple.com/postbooks`

hit user defined exceptions showing the effectiveness of our technique. We evaluated our technique on PetClinic4[6], which is an official sample distributed with Spring framework. We also evaluated our technique in comparison of DPF and observed significant reduction in state space size in some cases.

1.3.5 Organization

This document is organised as follows:

Chapter 2 details the background of automated software testing for database applications. It also provides literature review of different program analysis techniques that can be employed for automated testing of such applications.

Chapter 3 describes how symbolic execution can be adapted for database applications that rely on relational database model. It also gives quantitative proof of benefits for employing symbolic execution for testing such applications in comparison to other techniques.

Chapter 4 introduces effective partial order reduction for model checking database applications. We provide experiments to show that our technique is scalable and results in significant reduction in state space of program.

Chapter 5 describes how symbolic execution in conjunction with model checking can be used to test NoSQL based database applications.

Chapter 6 discusses how program analysis techniques can be employed to identify similar errors in program by analyzing error reports within bug repositories.

Chapter 7 introduces the idea of employing program analysis techniques in other domains such as embedded devices as well as to track the trackers on the Internet.

[6] `http://static.springsource.org/docs/petclinic.html`

Chapter 2

Background and Literature Review

"*All code is guilty until proven innocent.*"

Today, it is an important requirement to have bug free software systems. After writing a program, developers test software for correctness against program specifications prior to release to end user. Software testing is the process of validating and/or verifying system properties before releasing it to end users. It plays an important role to uncover errors present in the program through program execution.

2.1 Automated Software Testing

Test input lies at the heart of the software testing and its quality determines the effectiveness and correctness of software. These test inputs can be generated manually or in an automated fashion. In manual software testing, tester manually writes test cases to verify system properties given in program specifications. The number of written test cases for the program depends on the various factors such as length of the program, variety of operations it offers and also on the complexity of software. With an increase in the value of these factors, it becomes difficult for the tester to write test cases manually that provide 100% program coverage, hence eventually missing some scenarios. Due to this, not only some errors remain uncaught but also writing of manual test cases makes software testing extremely expensive. On the contrary, test automation not only reduces the cost

of testing software but also has some inherent advantages over manual software testing. It requires less human resources and can uncover errors that are hard to find. Automated software testing increases effectiveness, efficiency and coverage of software testing by exercising all program paths but often at the cost of time. However, this trade-off is suitable as the reliability of software is top most priority in every field. Automated software testing revolves around automated test input generation. Much work has already been done in area of random test input generation [15] [16]. Although these techniques provide limited coverage but still they have an advantage over manual testing as they do not require input from the user, deep understanding of code and/or manipulation of code.

2.1.1 Classification of Automated Software Testing

Automated software testing consists of a broad array of techniques that can be mainly classified as black-box and white-box testing. In another dimension, we can also classify automated testing as dynamic and static testing.

Black-box testing

Black-box testing focuses only on testing the functionality of the code. The tester does not need to know the internal structure, flow or implementation of the code. He will only be concerned with the input, the expected output, and the actual output.

White-box testing

White-box testing requires tester to have knowledge of program design and implementation. In white-box testing [17] source code is accessible; its logic can be inferred.

Static testing

Static testing involves analysis of the source code in order to verify code requirements without any execution. Static analysis ensures exploration of all paths of the program without exposing runtime behavior of the program. Errors occurring at run-time might not be caught by static analysis of

code. Static analysis of code can not capture run time behavior of the program that is why it is quite common to have false positives if we rely on static analysis only.

Dynamic testing

Dynamic testing involves execution of program code to verify software requirements. Dynamic testing of code exercises only one path of the program during a single execution but is true representation of the program's behavior at run time. Dynamic testing might not be able to catch bug if path with a bug is not executed but it eliminates possible false positives. Dynamic execution of the code can provide high code coverage but can suffer through state space explosion.

Despite the pros and cons of later three; these are widely used for testing applications. Automated software testing techniques are employed in testing of various application softwares such as web applications, databases, networking application, embedded devices as well as system programs, algorithms, specifications etc.

2.2 database-driven Applications

An increase in the popularity of Internet has made use of data ubiquitous. Information from this ever-growing data is stored in databases and can be accessed through database-driven applications. Data is everywhere and is growing exponentially. To predict, make life better, we are storing vast amounts of data. Clients from around the globe access these databases through applications. Recent advancement in technology has opened many paradigms to store, manage and retrieve information from the data sources (storage systems). Here, we discuss two main types of applications that make use of this data. Firstly, SQL based data-driven applications that deal with structured data and use standard query language to manipulate data. Secondly, NoSQL based data-driven applications that deal with loosely structured and unstructured data. These applications are NoSQL based applications and can manage huge amount of data.

2.2.1 SQL based database-driven applications

A large number of enterprise applications store their data using relational models in database management system (DBMS) to ensure data integrity and efficient accessibility of data. In such applications, database models are defined prior to the applications accessing them. Properties of such data models are defined for once and all. Data stored within these data models is then manipulated using standard query language commands. SQL based applications rely on relational models to define relationship among data elements and link them with each other.

2.2.2 NoSQL based database-driven applications

Applications that deal with vast amount of data, such as Big data are usually NoSQL based database applications. With expansion in the data and with increases in ever-growing requirements of business processes, these application's data model keeps on changing. This gives rise to the redefinition of how applications store, use and access data sources. Databases, such as MongoDB etc are used at the back end to manage data for such applications. Due to its schema-less nature, it is scalable and flexible as well, making it more suitable for applications that may expand in their functionality with time.

2.3 Program Analysis Techniques for Testing Database Applications

Testing of database applications can be broadly categorized as usability testing, security testing, performance testing and functionality testing. The focus of our research is on the testing functionality of the database applications. Functionality testing of a database application is crucial for the correct working of these systems. Much work has been done in the domain of database-driven applications accessing data using standard query language's data manipulation commands, whereas on other hand NoSQL applications still require the researcher's attention. Various techniques are used to ensure the correctness of database applications. These include brute force,

sampling databases, testing with mock databases, static analysis of code and through dynamic execution of code. Brute force is not an option as it is unimaginably expensive and even if we sample an existing database, it may compromise on code coverage. Some techniques generate mock databases but sometimes such techniques are unable to mimic the behavior of real databases. Database-driven applications heavily interact with a database system. As database applications treat DBMS as an external system, their testing is itself a challenge. In database applications program inputs, queries, database states, program variables and branch conditions are all closely related and often depend on each other. Sometimes program input variables are used to formulate SQL queries and often branch conditions involve results retrieved from SQL operations. This dependency leads to many possible combinations of test database state and program test input.

Since the last decade, automated testing of database applications has gained a lot of research attention. Test input generation for a database application is different from test case generation of simple applications in a way that for database applications, we need to generate database state in conjunction with usual program inputs. There are several techniques employed for the automated verification of database applications. Below we give an overview of these techniques.

2.3.1 Query based test case generation

Query Generation is the process of collecting constraints from either program code or from specifications to form queries that are used to generate database state required to cover the functionality of the database-driven application. Binnig et al. [18] introduced reverse relational algebra for generating a test case given an SQL SELECT statement and the desired output. In general, test oracles like this are not available. Furthermore, a given output exercises one particular behavior of an SQL statement, whereas our work is concerned with exhausting many possible behaviors under some bounds. Veanes et al. [19] modeled SQL queries as constraints and used SMT solvers to generate database table data. They also needed some specifications of the required output like the result should be empty or it should have a specified number of rows. While using SMT solvers to improve the analysis, they are also concerned with finding one particular database state for one particular query. Tuya et al. [20] introduced a new coverage criterion for testing of SQL statements con-

sidering the semantics of multiple SQL constructs. They later [21] worked on a constraint-based approach to generate test cases for SQL queries that satisfy their proposed criteria. The criteria were written in Alloy [22] i.e. required additional specifications and was focused on a single SQL statement. Another approach involves query generation to generate database states using symbolic execution [23]. It addresses the testing of database-driven applications from another perspective by leveraging the use of dynamic symbolic execution to generate database queries for existing databases using knowledge of prior executions. It gathers constraints from application code and uses them to generate database queries. Another paper [24] also discusses the query-based generation of test inputs for the database application. In contrast to the above approaches, our approach is focused on complete stored procedure analysis and does not need any additional specifications.

2.3.2 Schema constraints based test case generation

AGENDA [25] was one of the first tools for automated test case generation of database-driven application. It generates test cases for transactions in applications by considering the database schema constraints. To model the conditions imposed by the transaction logic, it relies on user supplied constraints and is focused on specific kinds of tests. Marcozzi et al. [26] proposed an algorithm for testing the control flow graph of Java code interacting with the database. It generates Alloy [22] relational model constraints for a given database schema, a finite set of paths from the control flow graph, and variables along those paths (both method variables and those used in SQL queries). It generates a symbolic variable for each value taken by the method variables or database tables during path exploration. The Alloy model generated ensures the execution of the path that involves these symbolic variables. Alloy Analyzer solves these constrains to generate test cases. In later work [27], they described how an SMT solver can be potentially used to model the constraints and generate test cases. This would make analyzing larger applications possible. However, this is an idea paper and they have not implemented or evaluated it. Potentially such a technique would face the same hurdles as other approaches implemented outside the database management system. The testing of integrity constraints on schema has also gained the researcher's attention. G.M. Kapfhammer [28] presented a technique for testing of a relational schema. It can handle multiple

databases as it works on SQL based constraints that are related to database schema and not stored in real memory. In our work, because of implementing it at the query engine level, we are able to implement and evaluate complex queries like multi-table joins which previous techniques are unable to handle.

2.3.3 Symbolic execution for test case generation

Symbolic execution is a program analysis technique for exhaustive program exploration while addressing the non-deterministic behavior of inputs. It treats variables in the program symbolically instead of using concrete values. Symbolic execution relies on path condition that is a conjunction of constraints gathered from branch conditions along a certain path in the program. These path conditions are then solved using SAT or SMT solvers for test input generation. Compared to dynamic testing, symbolic execution explores program behavior along as many execution paths as possible. It also enables to reason about the paths that are not reachable through random input. However, symbolic conditions for the path containing loops are difficult to enumerate for all iterations. When we compare it to static testing such as data flow analysis, it is able to provide precise information as it tracks actual symbolic values and relations of program variables along the paths of program. Many applications rely on persistent data storage, which makes testing of databases ubiquitous. Manual test cases are quite expensive for such complex applications. One of the popular techniques for automated test case generation is dynamic symbolic execution (DSE), which efficiently generates program inputs by gathering constraints and then negates those constraints to explore other side of the branch. PEX [29] is a Microsoft research tool, which is a dynamic symbolic execution engine for generating test inputs. In last decade, much work has been done on the generation of test inputs using DSE [30], [10], [31]. As far as we know, Emmi et al. [10] were the first ones to apply the idea of symbolic execution to database-driven applications. They used concolic execution and used two constraint solvers to obtain the test cases. First constraint solver was used to solve arithmetic constraints while other was specialized to solve string constraints. They were able to support partial string matches that are expressed in SQL with LIKE keyword. Although they supported a wide variety of constraints that appear in the WHERE clause,

their supported SQL grammar was limited to queries using a single table where as our work can handle multiple tables with complex joins. Emmi et al. designed their symbolic executor to maximize branch coverage in the code. Other researchers have also used symbolic techniques to test databases [5–9].

2.3.4 Coverage criteria based test case generation

A large number of techniques are based on coverage criteria. *Coverage criteria* refers to statement or code coverage, block coverage, branch coverage and path coverage, that is to which extent code is tested. It quantifies the testing of code, which is directly proportional to the quality of testing. In the case of database applications, coverage criteria is used to generate test inputs as well as database state. Some techniques require user directed data generation for databases. However, these techniques provide lower coverage [32]. Much work has been done to generate database state as well as test input to achieve high branch coverage of source code [24] [25] [10] [33]. Li et al. [23] and Pan et al. [5] extended their approach for different coverage criterion such as boundary value coverage, CACC and logical coverage. In another work [7], they use dynamic symbolic execution to generate program test input for higher block coverage. They address the problem by observing close relationships between branch conditions, input variables, SQL constraints and data obtained from the database. Based on these observations, it then formulates and executes database queries to get the appropriate program input. Our work is different from the above symbolic approaches as it is hosted inside the database and is able to apply symbolic analysis at a finer granularity and is, therefore, able to generate test cases for complete stored procedures.

2.3.5 Declarative Specification based Testing

A common approach in many of the above techniques is using declarative specifications in Alloy [22] and using the Alloy Analyzer to solve them. The solutions are often converted back automatically to INSERT queries that can populate a database. While Alloy is a powerful language, converting imperative constraints like those which are mixed with queries in a stored procedure are difficult to model, resulting in a substantially reduced SQL subset being modeled. In contrast,

our technique of instrumenting the query nodes in the database query execution engine results in both declarative queries and imperative constraints converted into a series of imperative sequential tasks on which standard symbolic execution techniques can be applied. While we do not support the entire SQL grammar, our limitations are not fundamental in nature and the technique can be easily extended to other SQL statements. Khalek and Khurshid [6] presented a framework that uses Alloy [22] to model a subset of SQL queries by automatically generating SQL queries, database state and expected results of queries when executed on a database management system. They have modeled SQL queries and database schema using Alloy which used SAT solver to populate tables. The focus of their work is testing the correctness of database management system itself and not of the applications using them.

2.3.6 Mutation Testing

Another popular white-box technique to test database applications is mutation testing. *Mutation testing* involves mutation of the code, i.e., we change or mutate some statements of the program code. However, these changes are small and do not affect the functionality of the code. Mutated and original program both are then executed with same set of test inputs, if the output of both programs is same, then it means that change in the statement does not affect the execution, and hence we kill the mutant. The objective of mutation testing is to check the quality of test cases by killing all mutants. Mutation testing is more reliable in terms of assessing the adequacy of tests than coverage criteria because techniques based on coverage criteria expose large number of test cases; some of which might be unnecessary. There are automated tools for code mutation [34] and SQL query mutation [35]. MutaGen [36], kills mutants in database-driven applications. Firstly, using SynDB framework, it translates SQL constraints to simple program constraints. Then it generates program code mutants using code-mutation tool [34] as well as SQL query mutants using SQL query mutation tool [35] at the points where database is accessed. It derives query mutant constraint and inserts them in transformed program code. Finally, it leverages DSE on original and transformed code to generate program inputs that satisfy weak query mutations. Execution of these inputs kills mutants which are translations of SQL query constraints.

2.3.7 Mock database based testing

Test case generation of database applications also requires a database state. Whenever the database is not accessible or in situations where a change in database state can have unpredictable outcomes, mock databases are used. These mock databases should be able to handle all scenarios given in the program code. Generation of mock databases that are compatible with the application's working is quite a challenging task. These mock databases are then used for test input generation. MODA (Mock Object based test generation for Database Applications) [33] is a tool that generates test inputs for database by producing and using mock database instead of real and actual database to mimic interactions between application and database. It is implemented using Microsoft research tool, PEX [29], which is a Dynamic symbolic execution engine for generating test inputs.

2.3.8 Use of tools and frameworks

Commercial IDEs like Visual Studio[1] have support for unit testing of stored procedures. However, this support is limited to automatically filling databases, executing the procedure, and comparing output. The database values and procedure inputs are not automatically generated. Work has been done to prevent SQL injection attacks in stored procedures [37]. They combine static analysis to instrument SQL statements in stored procedures and a dynamic part to compare the statements to what was observed statically. However, this technique is specific to SQL injection and cannot be extended to generic test case generation. There are some frameworks for testing database-driven applications. One of such frameworks is Agenda that populates database using input ranges specified by users [38] [39].

2.3.9 Software model checking

Software Model checking is an effective technique for verification of concurrent applications [11–13, 40]. It deals with exploration of all possible interleaving of processes leading to a state space explosion for a larger set of processes. In general, concurrent events are modelled by exploring

[1] `https://www.visualstudio.com`

all possible interleaving of events relative to each process which results in a large set of paths with many states on each path. Each path represents one unique interleaving of processes. Model checking is applied to different domains. Model checking has also been used for web applications [41–43] to model check interleaving of web requests but not of individual database operations. Thus, these techniques cannot detect issues when two requests are running concurrently and their individual database operations are not performed in a transaction. Web applications that avoid one transaction per request model to achieve scalability have a possibility of races between database operations. JPF (Java PathFinder) [44] is an explicit state model checker for race and deadlock detection in Java programs. It is also used for test input generation [45]. Model checking of programs leads to state space explosion. In order to circumvent state space explosion partial order reduction is often employed.

Partial Order Reduction

Partial order reduction in model checking is concerned with removing transitions that work on different shared variables. Flanagan and Godefroid presented partial order reduction that depends on dynamically detected dependencies [46] and showed that it performs much better than static detection. Model checkers employ partial order reduction to avoid unnecessary exploration of schedules. Partial order reduction addresses the problem of state explosion for concurrent systems by reducing the size of state space to be searched by model checking software. Partial order reduction [46–48] on shared variables and objects is a well-researched field. It exploits the commutativity of concurrently executed transitions, which results in the same state when executed in different orders. It considers only a subset of paths representing a restricted set of behaviors of the application, guaranteeing that ignored behaviors do not add any new information. Paleari et al. [49] found dependent operations through a log from a single execution, ignoring program semantics, and did not perform model checking of other possible orders. This made it label many operations as dependent when they really were not. Zheng and Zhang [50] used static analysis to find race conditions when accessing external resources. Static analysis inherently has more false positives. Compared to both these techniques, our technique has less false positives as it is based on model

checking but on the other hand it requires actual execution.

Model Checking of database applications

Relational database management systems are accessed by multiple clients to perform operations on data simultaneously. Output of these operations depends on the order in which these requests are served. Testing of such applications is challenging task as it involves test case generation for different schedules. As number of requesting clients increases, number of possibilities in which requests come to the server increases exponentially. Automated testing of such behavior is challenging task. Gligoric and Majumdar [51] applied model checking to database applications. They designed and implemented DPF (Database PathFinder), an explicit-state model checker for database-backed web application also built on top of the JPF model checker. They presented several implementation choices such as stateful vs stateless search, state storage for state restoration, backtrack strategies and dynamic partial order reduction. They presented multiple strategies for partial order reduction at different granularity levels by analyzing SQL statements and their dependencies on each other. They showed the fine-grained partial order reduction reduces the number of states most. The two most fine-grained approaches consider dependencies based on rows and attributes. We apply and optimize dynamic partial order reduction while model checking database operations. Our approach differs in two ways. First, we find the set of affected rows from the query engine in the database. Determining affected rows from inside the database is more precise than the pessimistic approach of determining what might be affected by the SQL statement. This also makes the technique more robust. Second, for operations that do not form disjoint sets, they pessimistically declare them dependent, however, we proceed to execute and check the alternate order in the database to see if there really is an affect given the data stored in the database. For example, they considered select and insert operation always dependent as insertions and selection operations do not form disjoint sets. Our approach is more effective in a way that we find dependencies over database state by re-executing queries for dependency analysis resulting in less number of false positive and more precise partial order reduction.

DPF also introduces a further mode where operations that do not overlap in attributes are con-

sidered independent. This is an orthogonal improvement that can be easily combined with our approach. In cases where one of the operations is a SELECT, we are still able to identify that the operations are independent because of our re-execution, albeit giving the cost of re-execution which was not necessary if attribute detection is added. Lastly, DPF does not handle complex queries like joins or sub-queries. We have not implemented complex queries. However, our technique enables much easier implementation of joins because we do not need to process SQL statements and deduce affected rows from them. Rather, we can use our existing instrumentation in the database to find rows affected in both tables and find disjoint sets. We aim to achieve that in future work.

Database PathFinder (DPF) [51] introduces model checking of database applications and describes partial-order reduction techniques in the context of database applications. However, it treats the database as an external entity and performs partial order reduction at the SQL language level. This forces it to take some pessimistic decisions about dependencies of queries on each other. Furthermore, working at the SQL level makes it difficult to handle complex queries. We address these problems in Effective Partial Order Reduction (EPOR) with a more precise analysis that further reduces the number of states.

2.4 Program Analysis Techniques for Big Data

Data is growing faster as are the bugs. Due to high competition in the software industry and keeping on-going demands of customers in view, software companies release their product at a very early stage of development that makes it impossible to test applications properly. Due to this problem, such applications are highly prone to errors (bugs) that can even lead to program crash. Crashing of a program causes annoyance for its users. There are bug report management systems that can manage millions of bug reports. Program analysis techniques can be employed to find, localize and repair faults in the program using these bug reports.

2.4.1 Automated techniques for fault localization, bucketing and repair

Several automated program analysis techniques have been employed for fault localization and repair in the program. Few address the problem of finding correlated bug reports. Many open-source programs [52] have built-in systems for bug reporting. Similarly, Apple [53] and Windows [54] also report crashes automatically to the development site. Millions of bug reports are collected each day. Developers examine these bug reports to find the faulty piece of code to fix the bug. Crashes, due to these bugs appearing to happen in different locations might be correlated, i.e., due to the same faulty function used in different places. Identification of correlated bugs is yet another challenging task that involves analysis of crash signature in a particular stack trace of the bug report. Correlation between bug reports can be used to improve the fixing of bugs.

Automatic bug bucketing

There are tools for bug reporting, such as Windows Error Reporting [54], Apple Crash Reporter [53], Mozilla Crash Reporter [52] etc. These tools focus on collecting bug reports and then grouping similar reports together. Bucketing of bugs targets to group similar bugs in a bucket which helps developer to prioritize fixing of the bugs, depending on the size of the bucket. However all of these techniques work on bugs related to one product at a time. In their paper, Yingnong Dang Rongxin and Hongyu proposed a method ReBucket [55], to cluster crash reports based on the similarity of stack traces. It calculates similarity among stack traces and then, based on similarity, it assigns the crash reports to appropriate buckets. They evaluated their technique using crash data of five Microsoft products Microsoft Publisher, Microsoft OneNote, Microsoft PowerPoint, Microsoft Project and Microsoft Access.

Bug localization

Much work has also been done in the field of bug localization based on crash reports. Bug localization techniques are used to recover potentially buggy files from bug reports. In a closely related work on improving bug localization using correlations in crash reports [56], authors exploit the significance of crash signatures, stack frames of traces and order of frequent subsets to create

Crash Correlation Groups (CCG). It offers a diversity of crashing scenarios that help developers identify the cause of bugs more efficiently. Our work is different from theirs in a way that they applied their technique on a single application at a time, whereas we find correlation among stack traces of different applications. In another work for locating crashing faults, the authors proposed a novel technique CrashLocator [57] for automatically locating crashing faults using stack traces. CrashLocator computes the suspiciousness scores of all functions by generating approximate crash traces through iterative stack expansions, using a function call graph generated from source code.

Correlated bugs

Another approach [58] for correlated bugs revolves around call graph of program. This is different from our work as according to their approach, two bugs are correlated if occurrence of one bug triggers other, whereas focus of our work is to find two similar programs using same buggy function.

2.5 Program Analysis Techniques for IoT

"*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.*" as stated by Mark Weisers in his paper [59]. Internet of Things (IoT) is all about bringing such thinking paradigm into reality.

IoT has its application in almost everything. Embedded devices turned a concept of smart cities into reality. From smart buildings with automated energy management [60] to agriculture [61], medical sciences [62], [63] and even for improved localization [64]; these devices play their role. Program analysis techniques can be employed not only to test programs written for these devices but also to make them efficient. Both static [65] and dynamic program analysis techniques can be used to combat security threats due to hidden bugs. Embedded devices are low powered energy-constrained devices. Program analysis can be used to save energy by checkpointing these devices before they can power out.

2.6 Program Analysis Techniques for Network and Computer Security

"If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology."

With advancements in technology, threats to network security have increased. Program analysis techniques are used to expose security vulnerabilities.

2.6.1 Malware detection

Malware detection is yet another major area of research. Much work has been done on the dynamic analysis of web pages and programs to classify malware. Nazca [66] monitors network traffic to differentiate between downloads of legitimate and malicious programs. It observes network traffic for many clients in a large scale network and applies several techniques to find suspicious downloads. It inspects downloads that exhibit malicious behavior to map suspicious connections on malicious neighborhood graph to detect malware distribution. Techniques to detect malware that make use of heap spray are static, dynamic and hybrid. Nozzle [67] provides defense against attacks that involve heap spray. Another work by the same group presents a semi-dynamic approach to detect malware [68] and hybrid approach [69]. It trains the classifier using websites marked as malicious by Nozzle and then on run time, detects whether a site is malicious or not.

2.6.2 Intrusion detection

Intrusion detection is notoriously hard to find and in recent years has taken the attention of researchers. PAYL [70] is automated payload based anomaly detector to detect intrusion in an unsupervised manner. Firstly, it computes profile byte frequency distribution and standard deviation of the application payload reaching each host and port. Next, it calculates the similarity of new data with a pre-computed profile using Mahalanobis distance and reports intrusion whenever distance of new input is larger then predefined threshold. HERCULE [71] is the intrusion analysis

system to detect attached communities. It is modeled as a community discovery problem that can be represented as a weighted graph of log entries across multiple lightweight logs.

2.6.3 Privacy leaks and authorships

Detection of privacy leaks [72], [73] in an android application is also a challenging task. Droid-Infer [73] addresses this issue in an effective way. Social networks are also prone to privacy leaks through profile information, tweets, photos etc. Face/off [74] detects an unauthorized user on the bases of photos and restricts it to access private information. Another work [75] addresses privacy leaks due to multi-party privacy conflicts. In another work [76] prevents disclosure of the sensitive data using static text analysis on both code and user interface. Third parties can access user information through cookies to perform interest mining to present users with the data of their interest. REPRIV [77] provides in browser privacy by restricting information sharing with third parties without user's consent. Another promising area of research is code reuse attacks, their detection and prevention techniques. The paper [78] presents an approach to prevent code reuse attacks using profile guided information related to hot and cold areas of code. PARS [79] is a modular password analysis and research system for password security implemented using various cracking algorithms, password strength metrics and password meters from academic, commercial and Alexa top 150 websites.

Chapter 3

Extending Symbolic Execution for Automated Testing of Stored Procedures

Stored procedures in database management systems are often used to implement complex business logic. Correctness of these procedures is critical for flawless working of the system. However, testing them remains difficult due to many possible database states and constraints on data. This leads to mostly manual testing. Newer tools offer automated execution for unit testing of stored procedures but the test cases are still written manually.

We propose an approach of using dynamic symbolic execution for generating automated test cases and corresponding database states for stored procedures. We model the constraints on data imposed by the schema and the SQL statements, treating values in database tables as symbolic. We use SMT solver to find values that will drive the stored procedure on a particular execution path.

We instrument the internal execution plans generated by PostgreSQL to extract constraints. We use Z3 to generate test cases consisting of table data and procedure inputs. Our evaluation using stored procedures from a large business application and various GitHub repositories quantify the evidence of effectiveness of our technique by generating test cases that lead to schema constraint violations and user defined exceptions.

3.1 Introduction

A lot of information systems rely on Relational Database Management System (RDBMS) to store critical data because of its transactional semantics, support for business operations, data-ware housing, analysis and reporting capabilities [3]. NoSQL based databases are slowly gaining popularity for collecting large amounts of loosely structured data but most critical infrastructure still relies on classic RDBMS.

Many applications using databases interact with multiple users simultaneously for which they use RDBMS to store, organize and manage data flow [80]. Such applications often require access to multiple database tables and sometimes involve decisions in the form of conditional statements. If database and application servers are on separate machines then the network overhead can be significant due to multiple requests to the database server. Moreover order of database accesses may lead to difference in database states [81]. To circumvent this problem application programmers often place business logic for common tasks in stored procedures within the database server. Stored procedures are modular programs that can have database accesses in addition to business logic and even contain transactions. As stored procedures reside on the server, they offer faster execution and reduced network traffic especially when called frequently. Moreover, stored procedures have been part of business applications for a long time; making them a good candidate for testing.

Correctness of stored procedures is crucial for correct working of the application. Effective performance of these applications requires thorough testing which is a laborious and expensive task due to its complexity. This gives rise to the need for automated software testing; of particular reference is automated test data generation for database-driven applications where the key idea is to automatically generate representative database state as well as usual program inputs. Symbolic execution is one of the several approaches that exist for automated test case generation. It is considered to provide increased code coverage for white-box structural testing of the application under test (AUT).

Symbolic execution is a powerful program analysis technique based on systematic exploration of (bounded) program paths, which was developed over four decades ago [82,83]. A core principle of symbolic execution is to build *path conditions*, a path condition is conjunction of the branching

conditions on the specific path. Solution to a (feasible) path condition is an input that executes the corresponding path. Automation of symbolic execution requires constraint solvers or decision procedures [14, 84] that can handle the classes of constraints in the ensuing path conditions.

A lot of progress has been made during the last decade in constraint solving technology, in particular SAT [85] and SMT [14,84] solving. These technological advances have fuelled the research interest in symbolic execution, which today not only handles constructs of modern programming languages and enables traditional analyses, such as test input generation [30, 31, 86, 87], but also has non-conventional applications, for example in checking program equivalence [88], in repairing data structures for error recovery [89], and in estimating power consumption [90].

Symbolic execution has been used with database-driven applications [5,10,22,23,26,27] using different techniques that are not *directly* applicable to testing of stored procedures. Firstly, because of the challenges posed by the multi-lingual nature of these applications (SQL and some imperative language) and secondly, these applications consider database as an external system. There are two research schemes in practice for symbolic execution of such code. The first scheme [1, 91] transforms the AUT into classical program code by replacing SQL queries with variables to represent database content and then modifying statements of AUT to act on these variables. This allows direct application of conventional symbolic execution on program code. The other scheme applies symbolic execution on application code directly. In this regard [10] uses result of query for generating test input (database relations) for program code by constraining relations with in path conditions. These path conditions are then solved to generate test input data whereas [2] directly translates database and application constraints in relational constraints.

In this work, we adapt symbolic execution to automate test case generation of stored procedures written in PL/pgSQL for PostgreSQL[1](an open-source database with support for stored procedures). For this purpose, we instrument internal query nodes of the database. This enables us to gather language and internal database constraints directly from within the database. Our work is different from the transformation scheme in a way that our executor executes the original code without transition and, contrary to other scheme [10], our executor directly deals with

[1] `http://www.postgresql.org`

program variables and database content symbolically while preserving their interaction with each other. Unlike a closely related approach [2], our symbolic executor does not require SQL DDL statements to extract database integrity constraints.

In contrast to dynamic symbolic execution of simple programs where program conditions are the only source of constraints, we identified three sources of constraints gathered during execution of the program through instrumentation of PostgreSQL.

(i) ***Language construct constraints*** are imposed on program variables, e.g. IF statement.

(ii) ***Database integrity constraints*** are imposed on the database tables to ensure that the data always satisfies certain properties. These constraints may be violated in some cases during the execution of the application, causing exceptions in the procedures. A typical example would be a primary key constraint violation that occurs when a user requests insertion of a record which already exists in the system.

(iii) ***SQL construct constraints*** are indirectly imposed by SQL statements. We extract these constraints from the query plans generated during execution of the SQL statements.

The important difference between language and SQL construct constraints is that the language constraints result in exactly two choices (True and False) where as, in the latter's case, choices depend on the number of table-rows being processed, e.g. if the table being scanned has two rows, then SELECT statement has four choices as a result. Similarly, eight choices with three rows in a table.

In this paper, we demonstrate an end-to-end technique for symbolic execution of stored procedures in PostgreSQL using the Z3 SMT solver [14]. Our major contribution is instrumentation at the level of query nodes that allows us to gather PL/pgSQL constraints as well as database tables integrity constraints due to same type system. We support database joins, aggregate functions and complex assignments that are often used in making decisions in business logic. Furthermore, we estimate the number of rows to be modeled initially which enables us to test significant number of scenarios. Our model for SQL Data Manipulation Language (DML) statements can change the number of symbolic rows in a table as the procedure is being executed. Our algorithm is adaptable to other database systems but its implementation is specific for the query node types and processing

in PostgreSQL. We evaluated our technique on 500 procedures that are part of some open source, publically available on-line projects and an Accounting and CRM ERP, PostBooks [2]. Our symbolic executor has generated around 350 cases that trigger constraint violations or hit user defined exceptions showing the effectiveness of our technique. The earlier version of this paper is published in *Proceedings of 31st International Conference on Automated Software Engineering* [92]. In this paper, we extend the scope of testing of stored procedures by handling joins, complex assignments and aggregate functions. Moreover, we provide a estimate of the number of rows to be modeled for a relation accessed by the stored procedure. We also evaluate heuristic on a larger number of procedures of multiple open-source projects.

3.2 Background

3.2.1 Symbolic execution

Introduced in [83], symbolic execution is a white-box structural testing technique for executing a program with symbolic values [93]. Symbolic execution defines semantics of operations that are already defined using concrete values. It also maintains a *path condition* for the current program path being executed by following its control flow. A *path condition* specifies necessary constraints on input variables that must be satisfied to execute the corresponding path. Static symbolic execution is applied on control flow graph for coverage criteria [94]— defined by static inspection of code, i.e. without involving any concrete input. In contrast, dynamic symbolic execution [17, 18, 91, 95] uses concrete input values to execute instrumented program for path exploration. Symbolic execution is applied in parallel to get the path constraint of the concrete execution. Then, by negating this constraint, we get another path constraint whose solution leads execution to another concrete path. With new inputs, this process is repeated for sufficient code coverage. As an example, consider the following program that returns the absolute value of its input. To symbolically execute this program, we consider its behavior on integer input, `x`.

[2] `http://www.xtuple.com/postbooks`

```
L1   static int abs(int x) {
L2     int result;
L3     if (x < 0)
L4       result = 0 - x ;
L5     else
L6       result = x;
L7     return result; }
```

```
path 1:[X < 0] L2 -> L3 -> L4 -> L7

path 2:[X > 0] L2 -> L3 -> L6 ->
L7
```

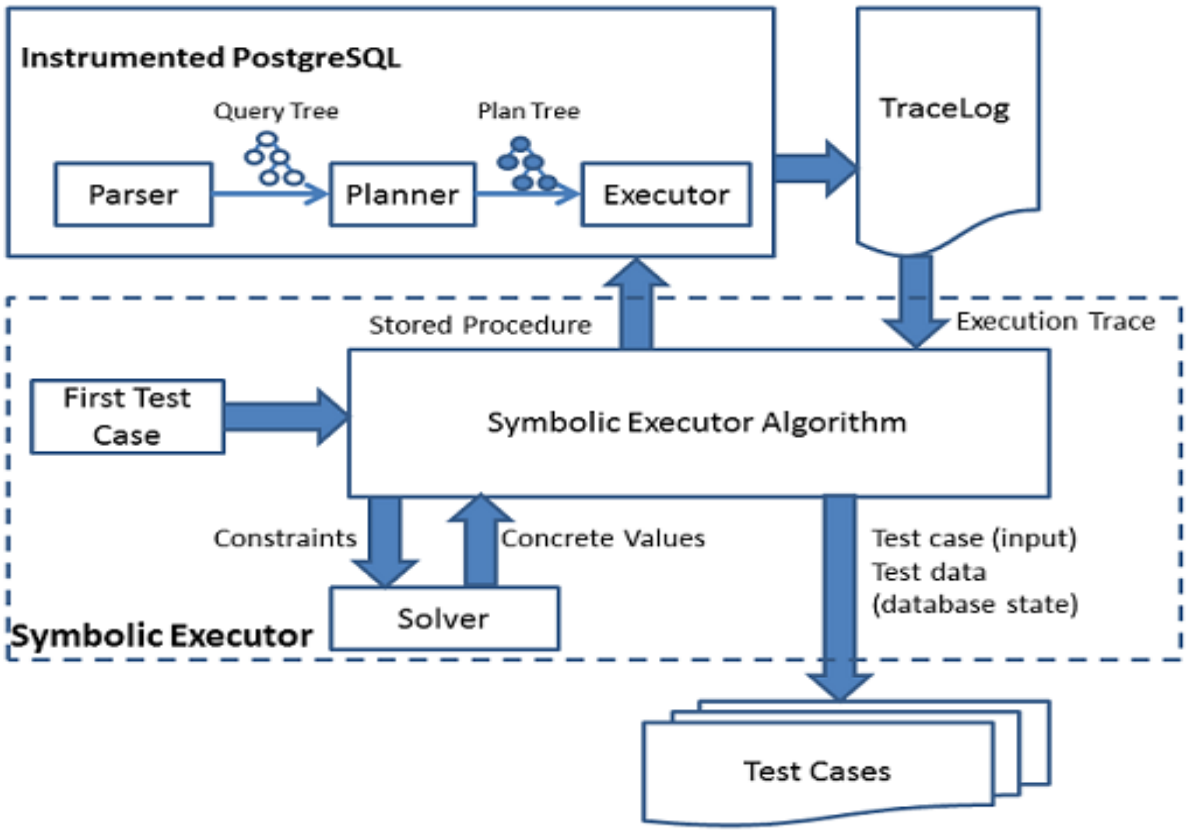
Upon encountering the conditional statement (choice point), a constraint is generated for the symbolic input x. We perform algebraic operations on symbolic values. At reaching end of the program, a path condition (shown in square brackets) is formed by collecting constraints along the path and then solved to generate concrete values for the symbolic values. Constraints gathered from choice points are then negated to drive the execution of the program to another path. Even though execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

3.2.2 database-driven applications

Relational databases are comprised of set of relational schema and integrity constraints where relations (tables) can have any number of columns of variable data types. Data driven applications quite often include several calls to DML statements spread throughout the code to interact with the relational database. For improved performance of such applications, business logic comprising of conditional and iterative operations as well as database accesses of program code is placed in form of stored procedures within database server.

Stored procedures are user defined functions containing usual program code as well as embedded SQL statements for selection and modification of database tables [4]. In stored procedures, SELECT statement is used to get values from database tables based on criteria defined by WHERE clause and stores them in program variables. Data modification statements UPDATE, INSERT and DELETE, which are function of program variables, modify database content provided they do not violate integrity constraints imposed on the tables.

Figure 3.1: Architecture of our technique



3.3 Technique

3.3.1 High level overview

Our approach aims to automate test generation of stored procedures within PostgreSQL, using symbolic execution. Symbolic execution of stored procedure is a challenging task as it involves constraints imposed on the relations appearing in DML statements in addition to the classic program constraints in form of conditional statements. Stored procedures used in PostgreSQL allow multiple client applications written in any language to have consistent database routines. Here we have chosen stored procedures written in PL/pgSQL. On execution of stored procedure, PostgreSQL prepares a plan— a tree like structure— consisting of nodes(called query nodes). Each node in PostgreSQL corresponds to a specific operation, e.g, SequentialScanNode represents se-

quential scan on table. Every node relies on its child node to provide information related to subsequent operations. We instrument internal query nodes of PostgreSQL to record execution of stored procedure in a file named 'TraceLog'. Then we process TraceLog to perform symbolic execution by gathering constraints on databases as well as from language constructs.

Execution of a PL/pgSQL procedure can be divided into execution of PL/pgSQL language constructs, and SQL statements. Since both have the same type system, they rely on the same expression processing subsystem to process conditions and expressions as shown in Figure 3.1. Our technique utilizes the fact that database tables have the same type system, making it simpler to map a relation between program variables and data stored in the database, a feature not available to logic written in other languages. This enables us to support a larger subset of the SQL grammar as given in Table 3.1.

```
1  CREATE OR REPLACE FUNCTION updateSalary(x integer)
2  RETURNS integer AS
3  DECLARE
4   salary integer; experience integer;
5  BEGIN
6    SELECT A.empSalary, A.empExp INTO salary, experience from emp A WHERE
         empNo = x;
7    IF NOT FOUND THEN
8       RETURN -1;
9    ELSEIF experience >= 4 THEN
10      salary = salary + 1000;
11   ELSE
12      salary = salary + 500;
13   ENDIF;
14   UPDATE emp SET empSalary = salary WHERE empNo = x;
15   RETURN 1;
16 END;
```

Figure 3.2: Example code of stored procedure

3.3.2 Illustrative example

We will use example code of stored procedure 'updateSalary' in Figure 3.2 to explain our technique. This procedure updates the salary of a particular employee based on his work experience. It first queries database to extract information of the employee then, based on his experience, modifies his salary.

Generation and interpretation of log file 'TraceLog'

PostgreSQL prepares execution plan as tree where TraceLog entries appear in pre-order traversal notation. For example, x + y appears as (+ (23 x) (23 y)), where 23 represents integer data type. Furthermore, start and end of each operation(SQL and language constructs) is marked to define scope of the operation. Table 3.2 gives details about notations used in TraceLog. For reader's convenience, procedural statements and description of their respective tracelines, i.e. lines of TraceLog are discussed side by side. In the following subsections, we discuss TraceLog entries(TE) against program code(PC) used as example.

Handling of function calls

As soon as a procedure executes, information about the function call is logged in TraceLog. The following TE shows that function returns some value as a result.

Execution of 'updateSalary' by our symbolic executor will generate 'TraceLog' as shown in Figure 3.3 whose contents are described in this section.

PC: CREATE OR REPLACE FUNCTION updateSalary(x integer)
RETURNS integer AS
TE: T_Result TargetList TARGET_ENTRY FunctionCall Func_id
ARGUMENT_START 23 1 ARGUMENT_END AS updateSalary

Number and type of arguments of the function appear within argument start and end markers. It also lists function identifier which is used to handle different types of function calls. Function calls are categorized on the basis of function language. For *PL/pgSQL functions*, we generate call condition by creating symbolic values for function arguments and imposing constraint on them

whereas for return value we only create symbolic variable. We solve call condition as soon as the function starts. *Non PL/pgSQL functions*, e.g., SQL built-in functions like nextval, date, time etc. don't show execution details in the trace.

```
1  T_Result TargetList TARGET_ENTRY FunctionCall 7129501 ARGUMENT_START 23 1
       ARGUMENT_END AS updateSalary
2  START_FUNCTION 7129501
3  PLPGSQL_STMT_EXECSQL
4  T_SeqScan 7129489 TargetList TARGET_ENTRY Col 23 empSalary AS salary
       TARGET_ENTRY Col 23 empExp AS experience Conditions 65 Col 23 empNo
       Param 23 $1
5  Into 23 salary 23 experience 23 increment
6  PLPGSQL_STMT_EXECSQL_END
7  PLPGSQL_STMT_IF Not Param 0 found
8  PLPGSQL_STMT_IF_END
9  PLPGSQL_STMT_IF 150 Param 0 experience 23 4
10 ...
```

Figure 3.3: 'TraceLog' for Example Code

Handling of SQL statement

Every time PL/pgSQL encounters an SQL statement, PostgreSQL prepares execution plan represented as a tree. We instrumented SequentialScan, NestedLoop, Agg (aggregate), Result and ModifyTable (For DML statements) nodes of plan tree to extract information. SequentialScan is used to perform a scan on the table. NestedLoop joins scans of two relations. Agg computes aggregate function. Result node generates a single row where values in the row being generated can be variables, constants or any expression containing variables and constants. We treat results of the expressions as symbolic elements in a new symbolic row. ModifyTable supports inserts, updates and deletes and relies on its child node to provide the data it needs. In all three cases, we check that the table modifications do not violate the table constraints by creating constraint conditions on the modeled table and adding the result as a choice.

PC: SELECT A.empSalary, A.empExp INTO salary, experience

FORM emp A WHERE empNo = x;

TE: PLPGSQL_STMT_EXECSQL T_SeqScan table_id TargetList

TARGET_ENTRY Col 23 empSalary AS salary

TARGET_ENTRY Col 23 empExp AS experience

Conditions 65 Col 23 empNo Param 23 $1

PLPGSQL_STMT_EXECSQL_END

Here, SELECT statement is used to get information about employee whose empNo matches input parameter of the function. SELECT corresponds to sequential scans and appears as 'T_SeqScan $table_{id}$' in TraceLog. It also contains TargetList (list of attributes of the table), where each entry in list tells about the data type as well as name of the attribute. Later, this information enables our symbolic executor to create the table model. Scan condition extracted from database query processing system appears in pre-order notation (as discussed above) after keyword 'Conditions'. Here, 65 is integer representation of '=' in PostgreSQL. Our algorithm for symbolic execution uses this information to create symbolic variables and to impose constraints on already defined symbolic objects.

Handling of conditional statements

IF statement within code can be a simple expression or it can have a complicated condition with an SQL statement embedded in it. For the simple case, the constraint is obtained from the expression processor where as in latter's case, it comes from the SQL processing. Consequently, generating cases for the SQL is the way to explore the possible directions the code can take from the IF condition. Its quite common for SELECT statements to return no data against scan condition. As a good programming practice, 'IF' conditions are used to address such cases. For following lines of code, our instrumentation for language constructs will mark start of IF condition followed by the condition itself. Here we are dealing with the case when no row is returned by the sequential scan. However, absence of this condition would likely be a bug causing an exception, which will be uncovered by our tool.

PC: IF NOT FOUND THEN
TE: PLPGSQL_STMT_IF Not Param 0 found

ELSEIF is treated as an IF statement. Condition in TE appears in pre-order notation. 150 is operator id, where as experience and 4 are the operands. Here our symbolic executor will create constraint by imposing condition on symbolic object for experience. Later, we will solve it to get new set of inputs. Execution of stored procedure with new inputs will force execution to within ELSEIF statement while regenerating TraceLog.

PC: ELSEIF experience >= 4 THEN
TE: PLPGSQL_STMT_IF 150 Param 0 experience 23 4

Handling of assignments

We also keep track of assignment statements in TraceLog to update state of our symbolic variables with expression result. i.e. *target variable* == *expression result*. Assignment also occurs with keyword INTOF which allows the SQL statements in PL/pgSQL procedures to directly assign their result values to variables. Here multiple variables can be simultaneously assigned new values. However, in both cases target variables hold a single value. There is a special case, when the result of query is directly stored in a variable using assignment operator. In such a case, target variable can hold different values (zero or more) depending on the number of rows in a table, type and complexity of operation in WHERE clause. We treat this point in execution as branch point and explore all paths by assigning possible values to the target variable one by one.

We will process TraceLog from this point onwards in same manner until we reach end of the path. Then by flipping conditions, we will explore other paths.

Features of data manipulation language statements

Select query

$< select-query >::= SELECT < id-list > [< func > (< att-name >)]$

$FROM < table-id > [, < table-id >][WHERE < condition >]$

$< id-list >::= * |< att-name > (, < att-name >)^*$

$< func >::= MIN \mid MAX \mid SUM \mid AVG$

$< table-id >::=< table-name > [[AS] < col-alias-name >]$

$|< table-id > [< join-type >]JOIN < table-id > ON < condition >$

$< join-type >::= INNER |< outer-join-type > [OUTER]$

$< outer-join-type >::= RIGHT \mid LEFT \mid FULL$

$< condition >::= (< condition >_1 AND|OR < condition >_2)$

$| (< boolean-op >< condition >) |< att-name > <|<=|=|! =|>=|> < expr >$

$< boolean-op >::= NOT$

$< expr >::=< expr >< op >< expr >|< att-name >| var|const$

$< op >::= +|-$

Insert query

$< insert-query >::=$INSERT INTO $< table-id >$

$VALUES < set-expr > < att-name >=< expr >[WHERE < condition >]$

$< set-expr >::= < att-name >=< expr > (, < att-name >=< expr >)*$

Update query

$< update-query >::= UPDATE < table-id >$

$SET < set-expr > < att-name >=< expr >[WHERE < condition >]$

$< set-expr >::= < att-name >=< expr > (, < att-name >=< expr >)*$

Delete query

$< delete-query >::=$DELETE FROM $< table-id > [WHERE < condition >]$

Joins and Aggregate functions within queries

used in select-query as stated above

Features of language constructs supported by our technique
Function calls
$< func-call >::=< func-name >< arg-list >$ RETURNS AS $< var-type >$
$< arg-list >::=< arg > [, < arg >]^*$
$< arg >::=< var >< var-type >$
$< var-type >::= char \mid integer \mid text \mid numeric \mid boolean \mid date \mid void$
IF condition
$< if-stmt >::= IF < cond >< stmt > \mid [ELSE[< if-stmt >] < stmt >$
$< stmt >::=< exp >$
FOR loop
$< for-loop >::= FOR < select-query > LOOP$
Assignment statement
$< assign-stmt >::=< var >=< select-query >$
Expressions
boolean operators
$< boolean-exp >::= [NOT] < exp >$
$< op >::= AND \mid OR$
$< exp ::=< var > \mid < exp >< op >< exp >$
binary operators
$< binary-exp >::=< exp >$
$< exp ::=< var > \mid < exp >< op >< exp >$
$< op >::= + \mid - \mid * \mid$
Coalesce expressions
$< coalesc-expr >::=$ COALESCE $(< var > [, < var >])^*)$
Data types
$< var-type >::= char \mid integer \mid text \mid numeric \mid boolean \mid date$

Table 3.1: SQL grammar and Language Constructs supported by our symbolic executor

TraceLog Entries	Description
SQL constructs	
START_FUNCTION	Marks the start of PL/pgSQL or user defined function
END_FUNCTION	Marks the end of pl/pgSQL or user defined function
PLPGSQL_STMT_EXECSQL	Shows that SQL statement is about to be executed
PLPGSQL_STMT_EXECSQL_END	Marks the end of execution of SQL statement
T_SeqScan <table_id>	Shows that sequential scan has been performed on table with table_id
T_Result	RepFresents result of database operation
TargetList <TARGET_ENTRY>	Represents list of entries where each entry is separated by TARGET_ENTRY. This list corresponds to attributes in DML(SELECT, UPDATE, DELETE) statements
TARGET_ENTRY	Displays information related to attribute of table in form of "Col col-type name—place no of col AS alias-name" where col-type is integer representing data type of col
FunctionCall id	Shows that function is called. id corresponds to function identifier and used to locate the name of function
Conditions id	Shows the start of conditions imposed on the variables. Id is integer representing conditional operator such as $\geq$, $\leq$ etc
Col id col-name	col-name is name or alias of table attribute and id is its data type
Col	Attribute of a table represented as column
Param id param-name	input parameter where id is data type and param-name is name of parameter. As general rule, we named variables as $variable_number. $variable_number is a sequence no.

PLPGSQL_STMT_ $< lang - construct >$	marks the start of language construct, where $< lang - construct >$ can be any of IF, WHILE, FOR statements
PLPGSQL_STMT_ $< lang - construct >$ _END	marks the end of language construct, where $< lang - construct >$ can be any of IF, WHILE, FOR statements
T_NestLoop join-type	Represents join operation on tables. join-type can be any of JOIN_INNER, JOIN_LEFTOUTER, JOIN_RIGHTOUTER, JOIN_FULL, JOIN_CROSS
ARGUMENT_START	marks the start of function arguments.

Table 3.2: Syntax and description of TraceLog entries

3.3.3 Symbolic executor algorithm

Unlike other techniques that either require some specification [19] or Data Definition Language(DDL) statements [2] for table data structure and database constraints, our symbolic executor relies on TraceLog. We process TraceLog line by line and populate our data structure which represents control flow graph of the program, essentially a tree like structure where nodes store information about program statement. Our symbolic executor algorithm generates test cases by traversing nodes of control flow graph in a depth first manner. Each *node* is a snapshot of current traceline corresponding to the execution of particular line of procedure. It also stores ChoiceSet—a set of choices where each choice is a path condition that leads out of the node towards different execution paths— along with ResultSet. ResultSet holds expected result corresponding to a choice in ChoiceSet. Note that DML statements in a code cause multiple path conditions out of the node.

Our algorithm uses symbolic execution at its core. We query database to get list of all procedures and apply our algorithm on each of them. For each stored procedure, our algorithm generates first test case by solving for its input parameters. Then it executes procedure with resultant concrete values. It creates symbolic values for database table elements using TraceLog. It also treats those procedure variables symbolically that receive values from these elements. It also imposes integrity constraints on these elements. Whenever database access modifies the state of a variable,

new constraints are defined on it. Every time language construct is encountered, new constraints are added to already defined symbolic variables. Upon reaching the point where we are unsure about the path taken by the program execution, we gather constraints along the path to form path condition and solve it. Then we execute stored procedure again after setting up database state with new set of inputs to lead execution to concrete path while generating a new TraceLog. We process TraceLog while skipping the processed tracelines assuming that the execution of those statements will be exactly the same.

Our assumption holds for PL/pgSQL language nodes but it is not completely true for SQL. SQL, being a declarative language, leaves it for the database to decide how to execute the statement through a planner. Thus, it is possible for the SQL statement to join two tables using a different algorithm in a different execution. Both plans would give the same final output but it will throw off our symbolic executor which was expecting the exact same trace till the last processed traceline. We addressed this issue by disabling planner optimizations without restricting language grammar in any case.

We keep track of the explored paths and on reaching its end, we explore another path by negating previous path conditions. We repeat this process until all paths have been explored.

3.3.4 Working example of symbolic executor

Let's consider the illustrative example of updating employee's salary from section 3.3.2. Before initiation of symbolic execution of 'updateSalary', our program connects to the database to extract signature of the procedure, i.e. number of input parameters and their corresponding data types. Using this information, we generate first test case. It will only contain program input values. Symbolic execution with concrete input is also called 'Concolic Execution'. Execution of stored procedure is shown in Figure 3.4.

(a) **Stored Procedure Execution** We run the procedure using the first test case to generate TraceLog (see Figure 3.3). We then analyze each line of TraceLog.

(b) **Table Rows Estimation** Next, we parse TraceLog to get information about database accesses to estimate number of rows in the initial model of the table (see section 3.4.1). By default this

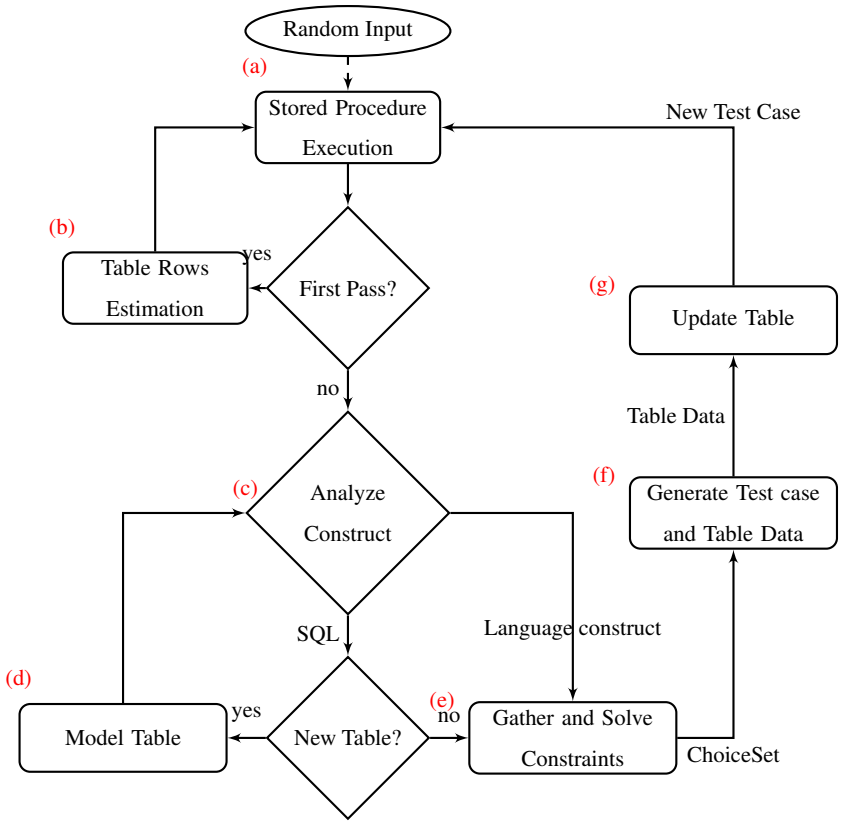


Figure 3.4: Flow of Symbolic Execution of Stored Procedure

number is two, which, in our observation, covers basic queries. SequentialScan performed on `Emp` will have its table-identifier in log file (see Line 4 of Figure 3.3). Since simple SELECT needs only two rows to cover sufficiently large number of cases, here we will set number of rows for table `Emp` as two. We determine number of rows to be modeled for each table only in the first pass of execution.

(c) **Analyze Constructs** If traceline has an entry for the SQL or language construct, we extract information about variables and tables, storing them in node of our data structure. For SQL constructs, if we already have the model for the table we move directly to *e*, otherwise we prepare model for the new table in *d*.

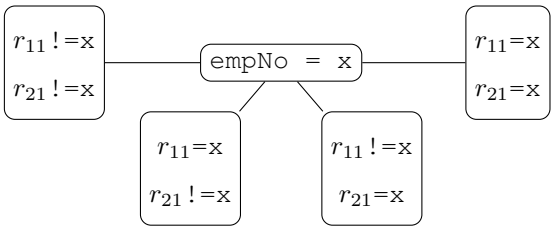
(d) **Model Table** Line 4 in 'TraceLog' tells that SequentialScan has been performed on the table `Emp`. As this table is accessed for the very first time, we do not have its model. We get the table

identifier from traceline and query database to extract the structure and integrity constraints of the table. We model table with estimated number of symbolic rows as shown in Table 4.1b. Here, we model table for two rows as calculated in *b*. In presence of foreign key constraints on the table, we will also model involved tables by extracting their structure and constraints. Long chains of foreign key constraints require a large number of tables to be modeled. We also store information about modeled tables (accessed and/or defined) in respective node of our data structure.

Table 3.3: Model for 'Emp'

eno	eSalary	eExp	eDept
r_{e11}	r_{e12}	r_{e13}	r_{e14}
r_{e21}	r_{e22}	r_{e23}	r_{e24}

Figure 3.5: Cases for Sequential Scan



(e) **Gather Constraints** For *SQL constructs*, we define constraints over symbolic rows of relations by using our models for DML statements. Adding integrity constraints ensures that data generated for tables will not violate constraints imposed on the structure of the table. Here, we draw inspiration from numerous works on symbolic execution that treat a simple IF condition as a choice point leading to two paths depending on whether the condition or its negation is true. Nonetheless, SQL statements define a choice point leading to two or more paths that too are not related to each other by simple negation due to presence of WHERE clause as well as integrity constraints on tables. In our example, sequential scan is performed on the table involving three cases for processing of sequential scan. (i) *None of the rows match*, (ii) *Only one row matches* and (iii) *Both rows match* scan condition. Case (i) and (iii) corresponds to one condition whereas case (ii) corresponds to two conditions.

Figure 3.6: ChoiceSet for SQL constructs in Example Code

```
ChoiceSet
{
Case 1: Not(r_e11==x),Not(r_e21==x)
    Result:[ ]
Case 2: (r_e11==x),Not(r_e21==x)
    Result:[r1]
Case 2: Not(r_e11==x),(r_e21==x)
    Result:[r2]
Case 3: (r_e11==x),(r_e21==x)
    Result:[invalid]
}
```

Figure 3.7: ChoiceSet for SQL & Language Constructs

```
ChoiceSet
{
// to explore if part
Case 1: Not(r_e11==x),Not(r_e21==x),
     (experience1>=4) Result:[]

 // to explore else part
Case 2:Not(r_e11==x),Not(r_e21==x),
     Not(experience1>=4) Result:[]
.....
}
```

All four conditions in Figure 3.5 are constructed. We add these to ChoiceSet of node as choices, shown in Figure 3.6. For each choice, we also store expected result of SQL statement that satisfies condition in 'Result'.

In case of *variable declarations*, we create symbolic object of the form `'var-name$seq-no'` and store it in current node, where seq-no is consecutive number assigned to variables and var-name is variable name. Thus, line 4 of TraceLog results in addition of symbolic object 'experience1' for 'experience' to current node. In case of *language constructs*, e.g. IF statements (line 11 in Figure 3.3), we impose condition on already defined symbolic object. Furthermore, we simply append constraints of language constructs to each of our choice in 'ChoiceSet' (Figure 3.7) to lead the execution of stored procedure to within IF statement. Note that choices in 'ChoiceSet' are generated lazily, i.e. choices are only available for the path under exploration at any point in time. When we reach end of path then we back track to explore other path. At that time our algorithm generates next choice.

(f) **Generate Test Case and Table Data** If traceline generates only one choice, which means only one path out of node, then we are certain about program flow and we simply process next traceline. But in the case when ChoiceSet of node has more than one choice (path)

out of node, then we are not sure about the path taken by the current trace, so we stop analyzing TraceLog and solve constraints to generate test cases. In order to generate test data, we pick choice from node. Then we express choices (constraints) in terms of SMT-Lib language. Finally, we check the satisfiability and solve the constraints to generate test data. Our symbolic executor then generates files for automatically setting up database and then executing stored procedure. Dataset for table and program input is given in Table 4.1d.

Test case: UpdateSalary(2)

Table 3.4: Dataset for Emp

eno	eSalary	eExp	eDept
1	3600	4	1
2	3800	6	7

(g) **Update Table** We setup tables with data generated in the previous step and re-execute the stored procedure with new set of inputs generated as test cases. For above example, we populate table `Emp` and move to step *a* for further execution.

We will keep exploring this path while generating test cases to lead our execution to the end of program through this path. On reaching the termination point, we will backtrack to explore other paths. Note that we explore the next choice either when it is marked as unsatisfiable by solver or when the program termination point is reached by the previous choice. We explore all choices in ChoiceSet for each node until tree is fully explored.

3.3.5 Handling of advanced SQL features

Joins in SQL statements

Joins—a commonly used operation in SQL statements— enable extraction of information from multiple tables by combining their columns. Output of join query involves access to multiple rows of same or different tables. We modeled execution of Cross, Inner and Outer joins. Number of resultant rows after join operations is a variable that depends on the number of rows in each table and number of rows that satisfy WHERE clause. We model resultant rows based on join type. For

two tables A and B with number of rows r_a and r_b respectively, number of resultant rows after join operation is given in Table 3.5. Number of resultant rows for Inner join is given as $(r_a \cap r_b)$, i.e. only matching rows will be selected. In case of Left Outer join, i.e. $(r_a \cap r_b) \cup r_a$, result of inner join is combined with rows from table A with null values in the columns of table B. $\cup r_a$ operation represents combination of non-satisfying rows from table A.

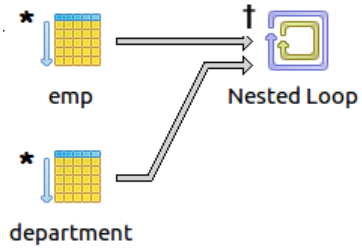
Table 3.5: Number of Resultant Rows Corresponding to Different Type of Joins

Join Type	**Rows in Output**
Cross Join	r_a x r_b
Inner Join	$r_a \cap r_b$
Left Outer Join	$(r_a \cap r_b) \cup r_a$
Right Outer Join	$(r_a \cap r_b) \cup r_b$

When two tables are joined to get information, we need constraint model that generates database state which works well with join type. We modeled join operation to achieve this task. For example, *select A.empSalary, A.empExp, B.bonus into salary, experience, increment from emp A inner join department B on A.empDept = B.deptId* will appear in TraceLog as follows.

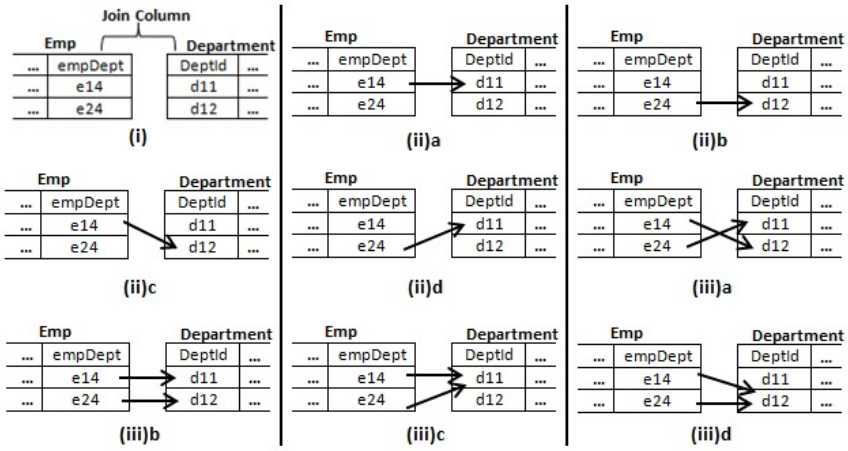
T_NestLoop JOIN_INNER TargetList TARGET_ENTRY Col 23 INNER.2 AS salary TargetList TARGET_ENTRY Col 23 INNER.3 AS experience TargetList TARGET_ENTRY Col 23 OUTER.1 AS increment Conditions 65 Col 23 INNER.4 Col 23 OUTER.0 Into 23 salary 23 experience 23 increment

Figure 3.8: Plan of inner join: $\star$ Tables emp and department are scanned for matching rows. † The resultant rows are joined using Nested Loop.



The plan for join query can be visualized as in Figure 3.8. During execution individual table is scanned and rows matching the condition(s) (if any) are filtered. Next, the resultant rows are

Figure 3.9: Possible Cases for Inner Join



joined in a Nested Loop. Here, it will lead to nine choices given in Figure 3.9. We extract join column from traceline and prepare conditions. Inner join is similar to WHERE clause, thus we model both tables with two rows. Left Outer join includes rows from both relations that satisfy WHERE clause in addition to all rows of the table on the left side of the WHERE clause. That essentially means that table on left side of the clause must contain number of rows greater than the number of rows in relation on the right side of the WHERE clause to test all cases. In this case, to model Left Outer join, we need three rows in table A.

Aggregate functions

Aggregate functions perform operations on a set of rows of a column to return a single result. Result of aggregate functions is often used in business logic to take decisions. In this version of paper, we extend coverage for SQL grammar by supporting aggregate functions in our symbolic executor. In PostgreSQL, AggNode is used to generate and return a row of result comprised of a single value and we model its execution. We express aggregate function as a symbolic expression and map it to variable storing result of a query. We also impose condition on table rows to satisfy aggregation function listed in SELECT statement. For example in case of Max, we impose condition on aggregated column to return satisfying rows.

Language constructs

The PL/pgSQL language constructs such as IF condition, FOR loops over SQL query results, assignments, variable initialization from SQL results, function start and return statements are supported. FOR loops over SELECT statements are also supported. The statement at the start of the FOR loop acts like an assignment statement and assigns a row from the FOR loop SELECT query results to the variables on which the loop runs. FOR loop works with currently modeled data types.

3.3.6 Processing of SQL

Execution of PL/pgSQL procedure can be divided into execution of 1) PL/pgSQL Language constructs and 2) SQL statements. Because of same type system, both rely on the same expression processing subsystem as shown in Figure 3.1 to process conditions and expressions. Expressions can also contain function calls allowing recursive procedure calls.

Everytime PL/pgSQL encounters an SQL statement it calls the SQL processing system to get the results. PostgreSQL prepares multiple possible execution plans with estimate of cost of executing each plan. An optimal plan represented as tree is selected for execution.

For example, *Select * from table1 t1, table2 t2, table3 t3 where t1.col1 = t3.col1 and t1.col2 = t2.num1 and t2.num2 ¿ 2* joins three tables and places extra conditions on column of *table2*. The plan for this query is shown in Figure 3.10. During execution, system scans *table1* and *table3* discarding the rows that don't meet condition. Results of the two scans are joined using a nested loop based join. Latter these results are joined with output of scan of *table2*.

We treat the nodes in the plan tree as basic building blocks of the program and modeled their execution in our symbolic executor. When a sequential scan executes, we extract the table identifier, the columns that appear in the output and condition(s). Extracted conditions decide the results of the scan. In the absence of condition, all rows are selected as result. Whereas, in presence of 'WHERE' clause, we impose the conditions on the table model such that each row in a table can either satisfy or dissatisfy the scan condition.

Here we draw inspiration from numerous works on symbolic execution that treat a simple IF condition as a choice point where system can take any of the two paths depending on whether

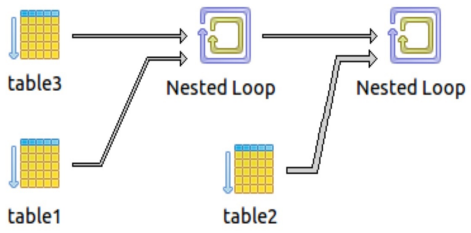


Figure 3.10: Plan for Join Query

the condition or its negation is true. Here we have a larger number of conditions that are not related to each other by a simple negation. Each of the conditions, if true, has a corresponding result model containing the rows selected by that condition. Therefore, we define choice as a set of conditions with corresponding result model. 'ChoiceSet' is the set of all choices at the node. During exploration of paths, at each node, we select a choice from ChoiceSet one by one, append table integrity constraints as well as conditions from already processed nodes along this path. We then solve it to generate corresponding result. However, if solver gives no solution for the condition, it means that choice corresponding to condition is not possible. This is a possible limitation but in our evaluation we did not encounter any case where solver timed out. If solver times out, we consider path to be unreachable. When we reach a node which needs the results of the earlier node, we can easily provide symbolic models of the results processed earlier. In example above, second NestedLoop requires results from the first NestedLoop and a Sequential Scan. Since output model of first NestedLoop is compatible with general result model, plan can be symbolically modeled. This means our grammar coverage of SQL is not based on syntax, rather it is based on the underlying execution plan of SQL.

3.4 Models for symbolic execution

In this section, we present our models required for processing of our symbolic executor. We model tables, data types, constraints and expressions. These models are then translated to SMT-Lib language by our tool and then solved using Z3 solver.

ALGORITHM 1: Test Case Generation Algorithm

```
Data: Solver = New Path Condition Stack, State = New ChoiceSets Stack, T= Test case
      with random values
Result: Test Cases: input and database state
Estimate table rows to be modeled;
while T is not NULL do
    TraceLog = Execute T on database;
    for each Line in TraceLog do
        if (Line is already processed) then
        State.Advance() // prepare new stack frame;
        ChoiceSet = ProcessTraceLine(Line);
        State.AddChoiceSet(ChoiceSet);
        if (ChoiceSet.size == 1) then
            Solver.Add(ChoiceSet.getCondition()) //Add the condition from the only choice
            in Solver;
    while true do
        Condition, StateAdvanced = State.NextChoice();
        if Condition == NULL then
            Terminate = State.BackTrack();
            if Terminate then
                T = NULL;
            else
        else if IsStateAdvanced then
            Solver.Add(Condition);
        else
            Solver.ReplaceTopFrame(Condition);
        IsCondSat, Result = Solver.solveCondition() the path condition stack in solver;
        if (IsCondSat) then
            T = MakeTestCase(Result);
```

3.4.1 Modeling of relations

Relations in database are two dimensional objects with fixed number of columns but variable number of rows. This makes modeling a relation a non-trivial task. There must be an adequate number of rows in modeled table to test all scenarios. Number of rows to be modeled in a relation is defined by the complexity of a query.

Table 3.6: Number of minimum symbolic rows in table model, where r_1 and r_2 is number of rows in first and second table

Join Type	r_1	r_2
Cross Join	$r_1 \geq 2$	$r_2 = r_1$
Inner Join	$r_1 = 2$	$r_2 = r_1$
Left Outer	$r_1 \geq 3$	$r_2 = r_1 - 1$
Right Outer	$r_1 \geq 2$	$r_2 = r_1 + 1$

Table 3.7: Data type models

Data type	**Solver type**	**Model summary**
Integer	Integer	Direct Mapping
Numeric	Real	Direct Mapping
Boolean	Boolean	Direct Mapping
Character	Integer	Integer represents ASCII value restricted to A-Z, a-z, 0-9
Text	Integer	Dictionary lookup value
Date	Integer	Offset from base date

For example, queries involving joins need to have a certain number of rows in the left or right relation to cover the possible scenarios. Depending on the join type, we assess number of rows to be modeled (see Table 3.6). Our model for the tables is a 2-D array of symbolic variables with data elements. Insertions and deletions in the tables are also modeled which can change the number of symbolic rows in the table as procedure is being executed.

3.4.2 Data type models

Attributes in the tables are of a specific data type. We treat attributes as symbolic variables and use Z3 to generate concrete values for these attributes. Our tool supports integer, boolean, numeric, character, text and date data types. First three data types map directly to corresponding Z3 solver type. We provide support for character, text and date data types by modeling them as integers as shown in Table 3.7.

3.4.3 Modeling of relational constraints

We have basic underlying models for primary key(unique and not null), foreign key and check constraints in the system. The modeling of a foreign key constraint requires addition of another table model. For recursive foreign key constraints, we model chains of foreign key relations which we disable only if we detect a cyclic relation during the processing of foreign key constraints. We also model execution of aggregate functions. Modeling of these functions and relational constraints is given in Table 3.8.

3.4.4 Model for expression processing

While processing constraints we come across a variety of expressions that we modeled for easy translation to SMT-Lib language for solving. We have modeled binary and boolean operators, testing for NULL values and Coalesce expressions. Operands of these expressions can be table columns, variables, constants or expressions. Boolean operations and binary conditional operators map directly to the Z3 solver API. Coalesce expressions are modeled such that they return the first not null value.

3.4.5 Model for sequences and special functions

The 'nextval' function relies on our model for sequences to output a symbolic expression. We model the sequence object as symbolic variable of type integer with starting value as 0. We return the expression *start_value* + *integer*, whenever nextval function is called for this object. To model special functions such as date and time, we treat the current date at start of symbolic executor as the base date.

Table 3.8: Constraint Models

Symbol	**Description**
a	attribute of table
pk	Represents primary key column of the table
fk	Represents foreign key column of the table
nnull	Represents Not null column value
agg	Represents column on which aggregate function is applied
expr	Holds value of expression; used in case of return expressions.
col-list	List of columns on which aggregate or check is applied
CHK(col-name)	Apply check constraints on the column, col-name
a_{ij}	Represents ith attribute in jth row of the table
pos_{key}	Position of the key column, where key = pk , fk, nnull, agg
primary	(for all i)((for all j)(and($a_{ipos_{pk}} != NULL), (a_{ipos_{pk}} != a_{jpos_{pk}})$))
not null	(for all i)(($a_{ipos}{}^{nnull} != NULL$))
foreign key	((for all i)($a_{ipos_{fk}}$ == (for all i)((for all j)and(($a_{ipos_{pk}} != NULL$), ($a_{ipos_{pk}}$!= $a_{jpos_{pk}}$)))))
unique	((for all i) ((for all j)(a_ipos_pk != a_jpos_pk)))
check	((for chk in (col-list))((for all i) CHK(a_ipos_pk)))
max	((for all pos_{agg} in $col - list$) ((for all i) ((for all j >i) ($a_{ipos_{agg}} < a_{jpos_{agg}}$))))
min	((for all pos_{agg} in $col - list$) ((for all i) ((for all j >i) ($a_{ipos_{agg}} > a_{jpos_{agg}}$))))
sum	((for all pos_{agg} in $col - list$) ((for all i) (($a_{ipos_{agg}}$ >0 and $expr_{ipos_{agg}} += a_{ipos_{agg}}$))))
Avg	((for all pos_{agg} in $col - list$) $expr_{pos_{agg}}$ = (((for all i) (($a_{ipos_{agg}}$ >0 and $expr_{pos_{agg}} += a_{ipos_{agg}}$))))) / i

3.5 Evaluation

We assess the effectiveness of our implementation of Symbolic Executor(SE) on multi-dimensional on-line procedures as well as on our crafted examples through experimentation. We assess the constraint modeling, scalability ability and cost effectiveness of our symbolic executor. Our key findings after the experimental evaluation shows that our symbolic executor

- is able to model constraints as well as tables with long chain of foreign key references.
- performs at par often providing better performance than existing approaches for automated test case generation of database applications.
- is reasonably scalable for solving generated constraints.

3.5.1 Settings

We implemented our technique using python. Given a set of stored procedures, our tool returns set of test-cases, where each test-case is comprised of test-data(database content) and test-input. In parallel to generation of test-cases, our tool executes these test-cases automatically by first setting up data using test-data then running test-input file. Outcome of the execution is recorded in our test-table within the database. We experimented our tool in a virtual machine with Ubuntu 16.04 as guest OS running on Intel Core i5 processor at 1.9GHz with 8GB of memory.

3.5.2 Testbed for evaluation

Our subject applications for empirical evaluation are partitioned into three groups.

- Firstly, to assess constraint modeling ability, we selected open-source projects that involved use of stored procedures with a variety of procedural constructs and database operations. These projects included Schemaverse[3], dvdrental, TheaterTicketing[4], TreeNode[5], AlgoSim[6],

[3] https://github.com/Abstrct/Schemaverse.git
[4] https://github.com/Yizhe-FAN/ParisVII_ProjectDataBase.git
[5] https://github.com/LSostaric/PostgreSQL-Node-Tree.git
[6] https://github.com/ashenoy95/plpgsql.git

and PostBooks. SchemaVerse is a space-based strategy game, implemented using a PostgreSQL database. dvdrental is a PostgreSQL sample database used to manage dvd stock. TreeNode has PL/pgSQL functions for managing tree nodes. AlgoSim is implementation of popular algorithms using databases.

- Secondly, to perform comparative analysis, we used open-source project RiskIt[7]. RiskIt is insurance quote system, implemented using Java. and used by other approaches for evaluation of their techniques. We converted procedures of RiskIt to PL/pgSQL stored procedures to perform comparative analysis of our technique. RiskIt contains 17 procedures and 11 tables. We further added 30 methods to test scalability.

- Lastly, for evaluating scalability of our SE, we use EDB[8], an open-source sample database whose schema is comprised of 10 methods. EDB is used to manage employee's records in an organization. We further crafted methods to address different construct (SQL and language) of various complexity (from simple queries to complex ones including joins). We also include variable number of queries to assess the scalability ability of our tool. In addition to above, we also evaluated scalability of our technique on relatively larger application, an open source Accounting and Enterprise Resource Planning (ERP) system, PostBooks which has a significant amount of its business logic written in functions in the database.

Altogether, our test bed includes 614 stored procedures with schema of 319 tables. We performed experimental evaluation on only 588 stored procedures as remaining had additional features that are not supported by our technique. accessing 319 tables. Summary of schema of test applications is given in Table 3.9. In the evaluation part of conference version, we made use of subset of stored procedures of PostBooks.

3.5.3 Results and discussion

Goal of our evaluation is to test effectiveness of constraint modeling, performance analysis and scalability of SE for automated test case generation. This section quantifies our key findings.

[7] `https://sourceforge.net/p/riskitinsurance`

[8] `https://www.enterprisedb.com/`

Table 3.9: Number of Constraints(Foreign key(fk), Check (chk), Unique) and Tables Modeled for Stored Procedures (SP)

No	Test Program	Tables	SP	Tables modeled	Constraints			Time
					Fk	Unique	Chk	
1	SchemaVerse	19	30	15	14	16	54	17597ms
2	dvdrental	18	8	12	12	14	70	21778ms
3	TreeNode	4	2	2	0	2	4	577ms
4	AlgoSim	13	6	5	0	0	0	1264ms
5	RiskIt	11	47	169	128	168	336	19539ms
6	EDB	3	10	21	10	32	42	2709ms
7	PostBooks	251	485	6440	11727	12457	42319	13m 55s

Database tables and constraint modeling ability

During symbolic execution of test applications, tables are modeled as soon as they are accessed by a stored procedure. Besides tables, we also model constraints associated with each table. Table 3.9 refers to the number of constraints and tables that are modeled. Our technique was able to model large number of tables and constraints. For SchemaVerse, around 54 check constraints are modeled, whereas and we modeled 183 check constraints for TheaterTicketing. PostBooks— a comparatively larger application—uses a very large number of constraints to ensure data integrity. In particular, the schema has over 400 foreign key constraints in 251 tables. This means that long chains of tables related by foreign keys are common. Even if a procedure directly uses few tables, we have to model the tables it references to be able to set up data properly for execution of the procedure. In 71 procedures SE ended up modeling over 30 tables and the constraints associated with them.

Performance comparison with other techniques

We used RiskIt to compare performance of our technique with two existing approaches on test generation for database applications. First is SynDB [1], which transforms database application

Table 3.10: Performance analysis of SynDB [1], RSE [2] and SE(our tool)

No	Procedures	Tables modeled	SQL queries	Times(Seconds)		
				SynDB	RSE	SE
1	getOneZipCode	1	2	21.3	-	1.821
2	createNewUser	8	7	-	11.294	2.685
3	getAllZipCode	5	2	27.9	-	3.282
4	filterEducation	2	1	13.1	-	1.149
5	Deleteuser	9	16	-	5.039	1.363
6	filterOccupation	4	1	17.6	-	5.559
7	filterZipCode	1	1	14.2	-	0.327
8	calculateIncome	0	0	-	-	0.107
9	calculateunemp--loymentrate	3	2	42.7	-	1.253
10	calculateRange	0	0	-	-	0.106

code to normalized program code and then applies classical symbolic execution on transformed code. Second is Relational Symbolic Execution (RSE) [2], which applies symbolic execution on program code and uses SQL DDL database schema for generating constraints on database tables. RiskIt application has 17 procedures that are data dependent and allow dynamic query generation. Out of these 17 methods we choose 10 methods to perform comparative analysis for two reasons. Firstly, they include diverse SQL statements and secondly, these methods are used by other approaches for evaluation of their technique. For fair comparison, we recorded the total execution time on similar machine. Results of other approaches are taken form [1] and [2]. Execution time by SE and other approaches is given in Table 3.10. From the Figure 3.11 and 3.12, it is evident that our technique is more efficient as compared to other approaches. In comparison with SynDB, the difference in execution time is much larger than that with RSE.

Figure 3.11: Comparison with SynDB

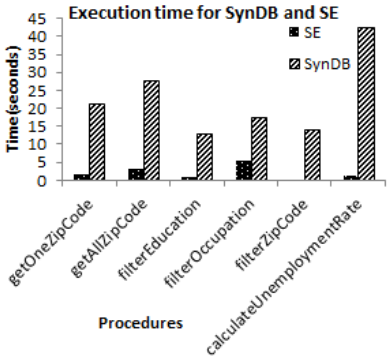
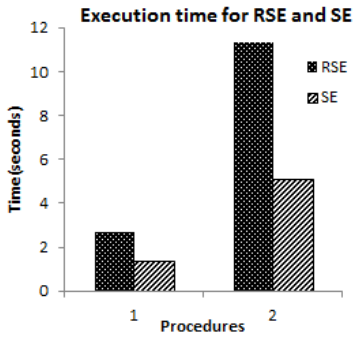


Figure 3.12: Comparison with RSE



In case of SynDB, such a performance difference is mainly attributed to the translation of SQL statements into native program code which increases the possible explorable paths. Code transformation along with increase in number of explorable paths altogether causes marginal difference in execution time. Our technique considers database constraints to be part of application code and applies symbolic execution on both (database and language constraints) simultaneously. However in case of RSE, performance difference is due to the fact that it generates database constraints by parsing and translating SQL DDL database schema whereas we extract this information directly from query nodes of PostgreSQL. Also as RSE is unable to handle character strings— a common data type in SQL—it evaluated its performance on two methods from RiskIt.

Testing scalability

Number of rows in table: In order to evaluate scalability of our technique, we used EDB. We further crafted methods to include different number of SQL statements as well as different types of SQL DML statements. Keeping number of rows constant while modeling tables can result in missing some test scenarios. A typical example for variable number of rows is joins. Even in presence of statements involving results from count operation may require certain number of rows in a table to test that particular scenario. Thus, number of symbolic rows to be modeled in a table is not a constant value in our symbolic executor rather it depends on the complexity of a query, in particular, joins. That essentially means that number of rows can vary. We ran procedures of EDB with variable number of rows. Table 3.11 shows results of our experiments with variable number of rows. Total execution time represents the time taken by our tool for gathering, solving constraints

Table 3.11: Total time(ms) taken to generate test cases by various number of rows

Procedure	#query	Test cases for r rows				Total time(ms) for r rows			
		r=2	r=3	r=4	r=5	r=2	r=3	r=4	r=5
emp_query	1	3	4	5	6	509	857	1062	2066
select_emp	1	2	13	21	31	1075	4030	10004	20020
get_dpt_name	1	3	4	5	6	173	386	601	1030
hire_clerk	2	1	1	1	1	32	36	38	37
fire_emp	2	3	4	5	6	179	361	775	1200
hire_emp	2	3	4	5	6	141	385	855	1300
hire_emp2	4	3	4	5	6	221	354	826	1160
hire_emp3	5	3	4	5	6	169	362	874	1120
update_emp_sal	2	5	7	9	11	1060	2036	5029	9450
Total	20	31	45	61	79	3639	8857	20103	37475

and then to generate test case comprised of both program input and database state. Figure 3.13 shows that number of test cases linearly scale with number of symbolic rows in a table model. For five rows in each table, it generates 79 test cases. Time taken to solve constraints by our technique also increases with increase in number of rows in initial table model(Figure 3.14). This is due to the fact that constraints generated for table rows are permutations of number of rows resulting in increase in solver time.

Figure 3.13: Test cases vs table-rows

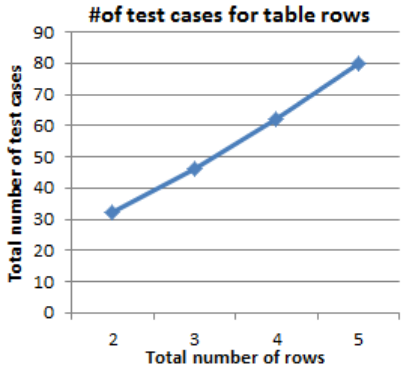
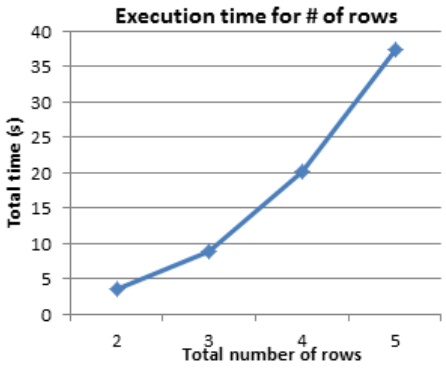


Figure 3.14: Execution time vs table-rows



Type of SQL statements in procedure: Type of SQL statement present in a procedure also greatly affect the execution time. Procedures in EDB involve DML statements of different complexity level. It is an establish fact that complex queries requires more execution time. Figure 3.15 shows that 'select_emp', despite having only one query, takes larger time to execute than 'hire_emp', which has 5 queries. The reason behind this boosted performance is due to the type of queries involved. 'select_emp' involves joins where as latter includes simple SELECTs and INSERTs. For concrete evidence, we selected procedures from PostBooks involving different number of joins and record their execution time (Table 3.12). Results shown in Figure 3.16 depict that as number of joining tables increase in the procedure, execution time also increase. 'actcost' involves only one join where as 'allocateforwo' has join on 6 tables. Although result show that type of join has great impact on execution time, nevertheless constraints within query also attribute to execution cost.

Figure 3.15: Execution time for EDB

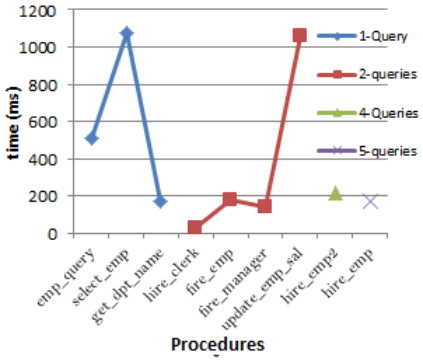
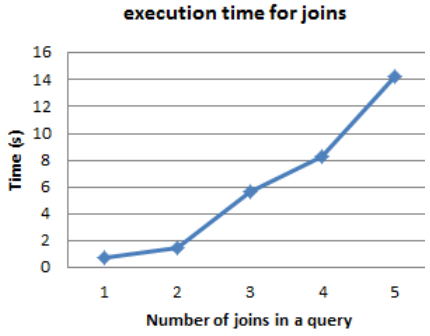


Figure 3.16: Execution time vs no. of joins



To test the impact of number of queries on execution time, we crafted EDB procedures to include large amount of SQL statements. Each procedure contains SELECT, DELETE, UPDATE and INSERT query. Single set of query is called multiple times and recorded results are plotted in Figure 3.17. After approximately 120 queries, there is exponential increase in solver time. Keeping in view that usually procedures are blend of SQL DML statement, it is fair to assume that our technique will generate test cases in reasonable amount of time.

Scalability limits imposed by Solver used in SE: We also evaluated scalability of Z3 solver used in SE on Postbooks against number of constraints generate as shown in Figure 3.18. Solver time increase linearly if number of constraints remains less than 63. For larger number of constraints

total time increases notably.

No	Procedure	#joins	time(s)
1	actcost	1	0.705
2	deletewo	2	1.44
3	calccobilltax	3	5.645
4	changepurchase-	4	8.231
	-opship	4	8.231
5	allocatedforwo	6	14.202

Table 3.12: Execution time for procedures involving different number of joins from PostBook

#Queries	Time(s)
40	24.1
80	76.8
120	142.7
160	297.3
200	483.3
240	1004.

Table 3.13: Solver time and total time taken to generate test cases for different SQL statements

Figure 3.17: Total execution time for different number of table rows

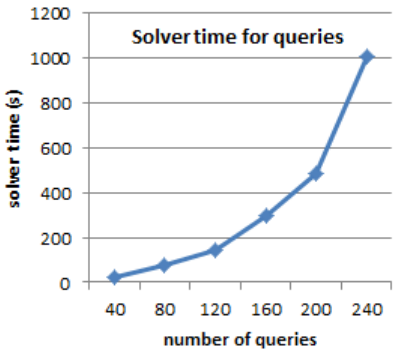
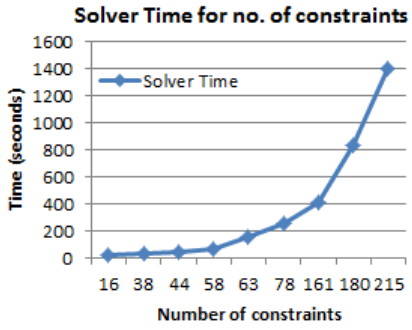


Figure 3.18: Solver time for generated constraints



Test generation cost

As it can be seen from the results that the performance of database applications does not rely on single factor rather it is a function of different variables most importantly complexity of a query. Other variables include number of attributes in a table, number of symbolic rows in a table and number and types of queries in the procedure. 'select_emp' and 'emp_query' both have same number of queries but there is a significant difference in time for test case generation. This is due to the fact that in both procedures number of selected columns in SQL statements is different.

Table 3.14: Exception Cases: Summary of Results for Constraint (Foreign key (Fk), Check (Chk)) Violations

No	Test Program	Exception cases	Constraint violations				User defined exceptions
			Chk	Fk	Invalid input	no data	
1	SchemaVerse	18	10	0	4	1	3
2	dvdrental	7	3	1	1	0	2
3	TreeNode	5	2	0	2	0	1
4	AlgoSim	1	0	0	0	1	0
5	RiskIt	42	12	2	14	0	4
6	EDB	33	10	2	7	5	9
7	PostBooks	196	35	9	115+	20+	27

Number of constraints to be modeled vary with the number of attributes in a table, requiring more time for test case generation for larger set of attributes in a table. Similarly, query involving more tables will take larger amount of time in order to model referenced tables (already discussed in 3.5.3). It can be concluded from experiments that complexity of a query is major factor that contributes to the cost most.

Discussion on generated test cases

After automated generation of test cases (procedure input as well as database state), we run each test case which is either executed successfully or thrown an exception. Different types of exception are logged in the table as a result of execution. These include user defined exceptions, constraint violations and no data found. Many of the procedures have user defined exceptions to give user friendly responses to the client. Constraint violations include foreign key, unique key and check constraints violations. Manual inspection of these exception cases enabled detecting a possible fault in the code. One such interesting example is of procedure createtodoitem from PostBooks, which throws an exception when called on inputs where inserted todoitem incdt_id does not reference any existing incdt_id as shown in Figure 3.19.

```
CREATE OR REPLACE FUNCTION createtodoitem( integer, text,..., integer,...)
DECLARE
  ptodoid   ALIAS FOR $1;
  pusername ALIAS FOR $2;
  pincdtid  ALIAS FOR $5;
  pcntctid  ALIAS FOR $16;
    _incdtid  INTEGER      := pincdtid;
  _result   INTEGER;
..
 BEGIN
 ...
   IF (_incdtid <= 0) THEN
    _incdtid := NULL;
  END IF;
...
  INSERT INTO todoitem ( todoitem_id, todoitem_username,...,
     todoitem_incdt_id,...)
       VALUES ( _todoid, pusername, ..., _incdtid,... );
...
```

Figure 3.19: Stored procedure resulting in exception case for foreign key constraint violation with given test case inputs `createtodoitem(2,'a',...6,...)`

Breakup of exception cases found is given in Table 12. Just like assertions provide an Oracle for test generation of normal programs, user defined exceptions and constraint violations provide an Oracle against which our technique can be used to automatically generate valid test cases. Reported exceptions contain a large set of no data found cases. The typical scenario for these exceptions is when a select statement tries to fetch data and no data is found but this case is not handled properly. Check constraints exceptions are mainly due to violation of constraints imposed on the column of the table. There is another kind of user defined exception which was over 40, where the symbolic executor generates test cases that violate NOT NULL constraint during UPDATE statements. Inspection of the code indicated that the UPDATE statements in these functions are directly using some of the input values of the procedure allowing the SMT solver to set them to any value to trigger a constraint violation. In some cases, programmer notified user with meaningful message.

3.6 Threats to validity

We compare our approach with two existing approaches on similar machines using limited number of procedures, which pose an internal threat to validity. Although test procedures cover most of the SQL and language constructs, testing larger applications would highlight performance differences between our approach and the others. However, even in the current scenario significant performance difference shows the effectiveness of our technique. Our dataset consists of applications that cover the supported grammar(SQL and language constructs), in spite of having limited number of procedures. For comparative analysis, we manually convert Java methods to PL/pgSQL stored procedures. Although procedures originally written in PL/pgSQL would be effective measure of testing, our testing on converted methods provides insight on the efficiency of our technique. Our testing for limited number of rows may miss some scenarios which require larger number of rows in at table.

With respect to external threat, we support large subset of SQL grammar as compared to [2], which is comprised of commonly used functions and operators. However, larger coverage would be a good representative of effectiveness of our technique. Although we handle text data type but

LIKE string operations are not supported. [2] also provides support for transactions which our tool does not.

3.7 Related work

Symbolic Execution. [93] and [83] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [96] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [86] defines symbolic execution for object-oriented code and uses *lazy initialization* to handle pointer aliasing. DART [30] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition to explore another path. To overcome the path explosion in large programs, SMART [97] introduced inter-procedural static analysis techniques to compute procedure summaries and reduce the paths to be explored by DART. CUTE [31] extends DART to handle constraints on references. EGT [98] and EXE [99] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. KLEE [87] is the most recent tool from the EGT/EXE family. KLEE has been shown to work for many off-the-shelf programs written in C/C++. Many recent research projects have proposed techniques for scaling symbolic execution by parallel and incremental execution [100–105]. **Testing Database Applications.** The testing of stored procedures is closely related to the testing of database-driven applications written in imperative languages. While we have not found any previous work on automated test case generation for stored procedures, the testing of database-driven applications has received significant attention in the past decade.

Treating database accesses as independent input relations: This approach focuses on generation of database state using SQL statements, independent of their interaction with program code. There are quite a few techniques that use test oracles in conjunction with program specification, particularly for database schema, to generate test cases. Given a test oracle (SQL SELECT statement and desired output,) [18] introduced reverse relational algebra for generating a test case where

as [19] used program specification for estimating resultant number of rows by modeling constraints in SQL queries. Then using SMT solver they generated one particular database state against each query. Many researchers test SQL queries based on the coverage criteria defined using specification. In this regard, [20] introduced new coverage criteria keeping semantics of SQL constructs in consideration. Later [21] worked on a constraint based approach to generate test cases for SQL queries that satisfied their proposed criteria written in Alloy [22]. Above mentioned approaches rely on program specification. In general, availability of test oracles is an issue, besides, above approaches exercise only particular behavior of query. Our work is concerned with exhausting many possible behaviors under some bounds, while being independent of program specification.

One of the first tools for automated test case generation of database-driven applications was AGENDA [25]. AGENDA generates test cases for transactions in applications by considering the database schema constraints. To model the conditions imposed by the transaction logic, it relies on user supplied constraints and is focused on specific kinds of tests. As far as we know, [10] were the first ones to apply the idea of symbolic execution on database-driven applications for increased branch coverage. They performed concolic execution using two constraint solvers, one for solving arithmetic and other for string constraints. Later on, [23] and [5] extended their approach for different coverage criterion. Although [10] supported a wide variety of constraints in WHERE clause including partial string matches with LIKE keyword, however, their supported SQL grammar was limited to queries using a single table unlike our work which supports joins with any number of tables.

Symbolic execution has also been applied on database application code with existing database state. In their recent work, [106] used the existing database state to generate program input. They gather constraints imposed on program variables by SQL statements and from results of a query. However, using existing database state may ignore some test case scenarios.

In recent years, program coverage has become important parameter for application testing. Even for database-driven applications, diverse variety of coverage criteria has been defined by researchers. Plain Pairwise Coverage (PPC) and Selected Pairwise Coverage (SPC) [107] testing makes use of existing database to detect faults due to the absence of a certain predicate. For

this, they extended code coverage by incorporating selected elements from SELECT query to define coverage criteria. Coverage criteria has also been applied to database integrity constraints. Although database semantics provide consistency and concurrency control but within a database itself schema constraints play an important part in ensuring data integrity and coherence. [108] employed coverage criteria on database schema integrity constraints on different granularities to test effectiveness of their approach. They defined nine different criteria to evaluate effectiveness of coverage criteria. These criteria are based on two different classes, constraint coverage and column coverage to test simple directive operations as well as complex integrity constraints such as multi column primary keys. Our algorithm works towards full path coverage by gathering constraint using control flow of the program. Our approach caters SQL constructs, database accesses and database integrity constraints. [109] presented incremental approach for test case generation that extracts constraints from test requirements, previous test case and initial database state. However, their tool has limited support for SQL grammar and does not support language constructs. It mainly focus on the SELECT statement involving joins and aggregates. [110] models test case generation for SQL queries as search based problem. They extract information from query plan of database to define fitness function, that is used to define coverage criteria. Their approach generate test cases that test SQL queries only whereas our algorithm generates test cases for entire stored procedure.

Much work has been done for testing of Java code interacting with database. Given a database schema, a finite set of paths from the control flow graph, and program variables, [26] generate Alloy [22] relational model constraints. It uses Alloy analyzer to solve these constraints to generate test cases. Their algorithm works by generating a symbolic variable for each value taken by the method variables or database tables during path exploration. In later work, [2, 27] gathered procedural symbolic constraints along with SQL constraints (embedded in program code) and solved them to generate database and program input. These techniques rely on program specification and do not handle complex operations such as null test, joins and string data type.

Using program specification for database generation Another approach uses declarative specifications in Alloy [22], solving them with Alloy Analyzer. The solutions are often converted

back automatically to INSERT queries that can populate a database. While Alloy is a powerful language, modeling imperative constraints mixed with queries in a stored procedure is difficult, resulting in a substantially reduced SQL subset being modeled. In contrast, our technique of instrumenting query nodes in the database query execution engine results in both declarative queries and imperative constraints being converted into a series of imperative sequential tasks on which standard symbolic execution techniques can be applied. While we do not support the entire SQL grammar, our limitations are not fundamental in nature and the technique can be easily extended to other SQL statements. [6] presented a framework, for testing the correctness of database management system, which uses Alloy [22] to model a subset of SQL queries by automatically generating SQL queries, their expected results, and database state when executed on a database management system. They have modeled SQL queries and database schema using Alloy that uses SAT solver to populate tables.

Transformation of database access into native application code Another popular approach [1, 91] transforms database application code to native program code by mapping database interactions onto program variables, then appling traditional symbolic execution on it. Code transformation technique is however time consuming and it may increase the number of paths to be explored.

Testing of database applications has gained much attention in recent years. Commercial IDEs like Visual Studio[9] have limited support for unit testing of stored procedures as database state and procedure inputs are not generated automatically. Another paper addresses SQL injection attacks in stored procedures [37]. It performs static analysis to instrument SQL statements in procedures, and dynamic analysis to compare statements to what was observed statically. However, this technique is specific to SQL injection and cannot be extended to generic test case generation.

[9] `https://www.visualstudio.com`

Chapter 4

Effective Partial Order Reduction in Model Checking Database Applications

4.1 Introduction

With the advancement of computer technology, highly concurrent systems are being developed. Relational Database Management Systems (RDBMS) are widely used for storing and managing data for applications. Typically, a single database server serves multiple client applications. RDBMS store data in tables (relations). Multiple clients can access the same tables at the same time. The accesses can be writes to the database tables (inserts, update and delete statements) or reads from the database tables (select statements). Moreover, due to these operations database state is changing continuously. At a given time, multiple clients are sharing the same database state. Multiple clients are accessing these tables for different operations. Their output (or result) of access depends on the state of the database at that time. Order of execution of database operation depends on the arrival of client request, which could be different for different cases depending on factors like network traffic etc. Many different schedulings (order of processes) of requests are possible in such situations.

4.1.1 Deriving Problem

Distributed applications, particularly web applications, often depend on a centralized database. The results of database operations depend on the state of database at that time and often also on the order of execution of operations performed by concurrent clients. Verification of such applications requires modeling all these possible orders so that the user can determine which are incorrect orderings and can prevent them with transactions or business logic. However, straightforward exploration leads to state space explosion. In a given schedule, if two operations share database state, they are dependent. Result of the database access truly depends on the state of database at that time. Although database semantics inherently ensure atomicity and have strict concurrency control [111] but, based on the dependence of two operations, the outcome of database access for one client can be non-deterministic. This can lead to the overall unpredictable behavior of an application. Since the number of possible schedules is permutations of the number of active processes, with an increase in the number of processes, the number of different schedules grow exponentially. The number of schedules is the most crucial factor in handling state space explosion for concurrent programs.

4.1.2 Solution to the Problem

Partial order reduction prunes orderings that are equivalent to other orderings already explored. We present a novel technique of Effective Partial Order Reduction (EPOR) for model checking software of Java applications sharing database state. EPOR improves upon prior work by performing a more precise analysis and supports many more operations. The key idea behind EPOR is that monitoring the effect of database operations inside database implementation gives a more precise view of operation dependencies than what can be achieved from an external view. Like prior work, EPOR also relies on Java PathFinder model checker for model checking Java applications. However, unlike prior work, there is additional instrumentation inside the database that enables precise analysis, allowing support for more constructs. Our results improve upon earlier work by achieving a significant reduction in the number of states explored and thus allow more effective model checking of database applications with concurrent operations.

Our key idea is to observe the effect of SQL statements by instrumentation inside the database engine. Instead of analyzing SQL statements, we analyze which rows are read from and written to during execution of a particular SQL statement and if needed, temporarily observe the effect of SQL statements applied in the opposite order. Analyzing the effect of SQL statements inside the database reduces the need to parse and process every SQL statement separately and the automatic reexecution based analysis enables much precise partial order reduction than what was proposed in DPF.

We are using PostgreSQL database engine[1] and Java PathFinder (JPF) [11] model checker for our implementation. We consider concurrent Java processes interacting with a database through SQL statements. We model processes as threads in a multi-threaded application and use JPF to generate thread schedules for the application. We find commutative paths by analyzing database accesses and interdependency of PostgreSQL statements executed by multiple threads. The actual queries are executed in the database, which is run along with the model checker to check record sharedness and to mark dependency among records of same tables. Rows with possible read write conflicts, e.g., when an INSERT query affects a SELECT query only if specific data is inserted are identified by rerunning the SELECT to identify precise dependency. The results obtained are used by our dependency analyzer to mark statements dependent for exploration of other interleavings.

4.1.3 Contribution

We make the following contributions:

- **More Precise Partial Order Reduction** Our Effective Partial Order Reduction technique (EPOR) is more precise as it finds dependencies among queries by executing them and observing them inside the database.

- **Instrumentation of PostgreSQL** We instrumented PostgreSQL query nodes to extract unique row identifiers to perform precise dependency analysis and enable the dependency analysis of many SQL operations.

[1] `http://www.postgresql.org`

- **Implementation** We implemented partial order reduction for Java applications using PostgreSQL. Our algorithms for partial order reduction are adaptable to other database systems as well. Based upon ideas in DPF, we also implement database state restoration with JPF backtracking. We exploit depth-first search mechanism of JPF to map JPF state identifiers to database save points for efficiently restoring database state while backtracking through program states.

- **Evaluation** We evaluated our technique on PetClinic4[2], which is an official sample distributed with Spring framework. We also evaluated our technique in comparison with DPF and observed a significant reduction in state-space size in some cases.

4.2 Motivating Example

To explain dependent database accesses i.e., dependency of PostgreSQL statements on each other, which form the basis of partial order reduction, we consider a simple example of an Employee Management System (EMS). Our database schema for this example is a single table COMPANY, which contains records of the employees of the company. It has four attributes named ID, Name, Location, and Salary. We have a primary key constraint on the ID column. To illustrate this example, three rows of table COMPANY are shown in Table 4.1a. Multiple operations can be performed on this table. In our application, we consider only two operations: `addBonus(int maxSalary, int Bonus)` and `updateSalary(int increment, String loc)`. Function addBounus basically adds bonus amount to the employee's salary if salary is less than given threshold value maxSalary. So inputs for this function are Bonus and maxSalary. The other function, updateSalary, increments the salary of the employees based on the location. Location(loc) and increment are two input parameters for this operation. We have mapped these two operations on two threads such that each operation is performed by a different thread. Code snippet for both operations, executed by different threads, is shown in Fig. 4.1 and Fig. 4.2.

Consider ThreadSchedule-I given in Fig. 4.3. The SQL statements accessing the database by

[2] `http://static.springsource.org/docs/petclinic.html`

```
1  void addBonus(int maxSalary, int Bonus){
2    int id = 0;
3    int newSalary = 0;
4    int bonus = Bonus;
5    Connection conn = DriverManager.getConnection (url, "postgres","");
6    Statement stmt = conn.createStatement();
7    ResultSet rs = stmt.executeQuery("SELECT ID, Salary FROM Company WHERE
         Salary<maxSalary");
8    if (rs.next()) {
9      id = rs.getInt("ID");
10     newSalary = rs.getInt("Salary");
11     newSalary = newSalary+ bonus;
12     stmt.executeUpdate("UPDATE Company SET Salary="+newSalary+" WHERE
           ID="+id);
13     newSalary = 0;
14   }
15   stmt.close();
16   rs.close();
17   conn.close();
18 }
```

Figure 4.1: Example Code for identification of data dependence problem when executed by two processes/ threads. It adds bonus to Employees Salary, if their salary is less than maxSalary, passed as input parameter

Thread 1 for addBonus operations are statements 7 and 12. Statement 7 is retrieving IDs of those employees who have salary less than maxSalary passed as input to the function and then on line 12, there is SQL statement for modifying the salary of previously retrieved employees by adding bonus to it. Thread 2 is performing updateSalary operation and it accesses the database through statement 4 where, based on the location, salary of employees is incremented by value passed to this function as integer.

When ThreadSchedule-I is executed with test inputs (input 1) [maxSalary=12000, Bonus=500]

```
1 void updateSalary(int increment, String loc){
2   Connection conn = DriverManager.getConnection(url, "postgres","");
3   Statement stmt = connection.createStatement();
4   stmt.executeUpdate("UPDATE Company SET Salary=Salary+"+increment+" WHERE
        Location="+loc);
5   stmt.close();
6   conn.close();
7 }
```

Figure 4.2: Example Code for identification of data dependence when executed by two processes/ threads. It updates the salary of the employees based on their location

```
Thread 1: Line 7   addBouns: SELECT
Thread 1: Line 12  addBonus: UPDATE
Thread 2: Line 4   updateSalary: UPDATE
```

Figure 4.3: ThreadSchedule-I (Thread 1 → Thread 2)

```
Thread 2: Line 4   updateSalary: UPDATE
Thread 1: Line 7   addBouns: SELECT
Thread 1: Line 12  addBonus: UPDATE
```

Figure 4.4: ThreadSchedule-II (Thread 2 → Thread 1)

and [increment=2000, loc='Texas'] for addBonus and updateSalary respectively. Thread 1 selects an employee of table COMPANY with ID 2 and modifies its salary by adding 500 to it. After completion of addBonus operation, Thread 2 performs updateSalary operation based on the location. This time Thread 2 will modify the salary of employee with ID 2 changing the state of database again. Final database state after the completion of ThreadSchedule-I, is given as in Table 4.1b.

In the other schedule, given in Fig. 4.4, Thread 2 performs updateSalary operation followed by addBonus operation by Thread 1. Given the initial state of database in Table 4.1a and same input as before, Thread 2 updates the salary of employee with ID 2 to 12500. Then Thread 1 executes addBonus. After the execution of both operations for ThreadSchedule-II, state of the table COM-

Table 4.1: Initial State of COMPANY table from Example

(a) Initial State

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	10500
3	Alice	Norway	14000

(b) AFTER EXECUTION OF THREAD SCHEDULE-I WITH INPUT 1

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	13000
3	Alice	Norway	14000

(c) AFTER EXECUTION OF THREAD SCHEDULE-II WITH INPUT 1

ID	Name	Location	Salary
1	Bill	California	16000
2	Bob	Texas	12500
3	Alice	Norway	14000

(d) AFTER EXECUTION OF THREAD SCHEDULE I & II WITH INPUT 2

ID	Name	Location	Salary
1	Bill	California	18000
2	Bob	Texas	11000
3	Alice	Norway	14000

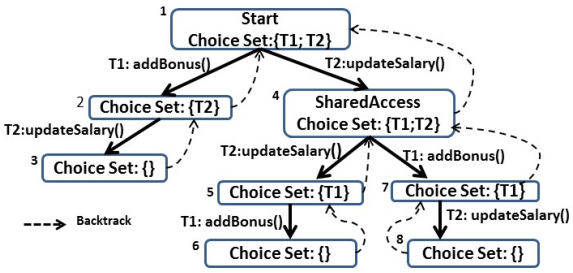


Figure 4.5: State Space of the Program with Dependent Statements

PANY is given in Table 4.1c. It is evident from output of ThreadSchedule-I and ThreadSchedule-II given in Table 4.1b and Table 4.1c respectively that final salary of 'Bob' is non-deterministic. Since both threads are accessing the same rows of the table, the final state of the database depends on the schedule in which two operations are performed. Such operations are dependent operations. In order to test dependent operations, we have to consider all possible schedules involving all active processes at that time. The state-space of the program with dependent statements is shown in Fig. 4.5.

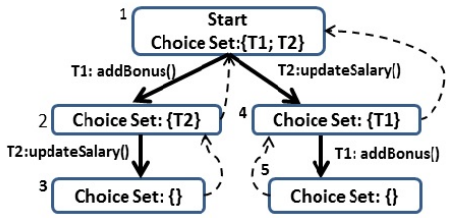


Figure 4.6: State Space of Program with Independent Statements

In order to differentiate between dependent and independent processes, lets consider the same operations addBonus and updateSalary with test inputs (input 2) [maxSalary=10000, Bonus=500] and [increment=2000, loc='California'] respectively. Given the same initial state as in Table 4.1a, consider two threads executing ThreadSchedule-I and ThreadSchedule-II (Fig. 4.3 and Fig. 4.4).

With ThreadSchedule-I, only one row of the table COMPANY with employee ID 2 is modified when Thread 1 executes addBonus operation. Then Thread 2 performs updateSalary operation updating the salary of employee with ID 1. After completion of the execution of both threads, table COMPANY will be modified to Table 4.1d. We can see that both Thread 1 and Thread 2 operate on different rows of the same table. Now, consider ThreadSchedule-II (Fig. 4.4) where Thread 2 modifies the salary of employee with ID 1 whereas Thread 1 adds bonus to the salary of employee with ID 2. After the successful execution of both operations, it is evident from state of table COMPANY given in Table 4.1d that both threads accessed different rows of the table. The state-space of the program with ThreadSchedule-I & II is given in Fig. 4.6.

As can be seen from the results, no matter in which order these processes are executed, the final state of database remains the same for this input. Such processes are independent processes and there is no use to explore both schedules as it would only increase the state-space of the system under test. The naïve approach to consider all possible schedules of database accesses by different threads will lead to state explosion, especially for a program with larger number of concurrent operations. The state space of program with ThreadSchedule-I in Fig. 4.3 using naïve approach can be given as in Fig. 4.7. In our work, we explore different schedules based on dependency relation of operations on each other. Hence, we reduced the state space to be explored by executing only one interleaving of processes, if they are independent. In example given above, state space of

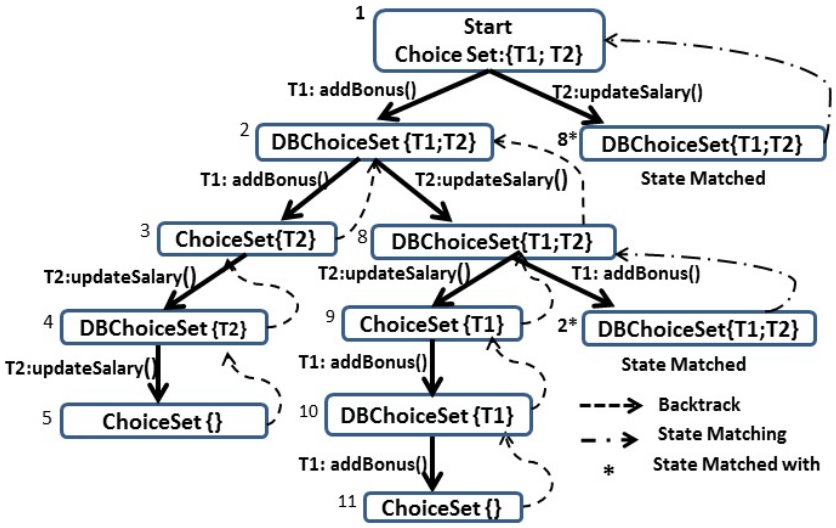


Figure 4.7: State Space of Program due to DB Access Choice Points

program for dependent operations is reduced to Fig. 4.5, which is further reduced to the Fig. 4.6 for independent operations, exhibiting significant reduction in state space of program.

4.3 Background

4.3.1 PostgreSQL

We have used an open source database PostgreSQL. Its relational schema is a finite set of relations used to store data for the applications. Each relation or table has certain attributes. A record is an ordered list of attribute values and representation of a row of the table. In PostgreSQL, every record in relation has a unique row identifier named `'ctid'` which is a system attribute and its value is assigned by server itself, once the record is inserted for the very first time, or whenever it is updated. `'ctid'` is mapped to $RowId_i$ which uniquely identifies the i^{th} record of the relation r.

Our program communicates with PostgreSQL using PL/pgSQL which is PostgreSQL implementation of the declarative structured query language (SQL). PL/pgSQL provides a set of statements to perform certain operations on the relations of the database. We focus on a simple state-

ments that allow querying, insertion, deletion, or updates in the tables of the database. SELECT, INSERT, DELETE and UPDATE statements are used to perform these operations. In general, SELECT statement is a read only operation, as it returns records in a relation that satisfies the condition c specified in the 'WHERE' clause, whereas INSERT, DELETE and UPDATE statements performs write operation as they modify the state of the database. DELETE and UPDATE statements modify the database state if condition c in 'WHERE' clause is satisfied.

4.3.2 Java PathFinder

Java PathFinder is a model checker that explores different interleavings of threads by selecting a thread non-deterministically from a set of live threads. In order to explore all possible schedules, JPF uses the choice generator (to generate non-deterministic choices) on thread operations. JPF generates schedules on the basis of calls to initialization and termination of these threads by executing the statements of the thread until the thread finishes the execution.

JPF executes Java byte code instructions from the application on its own Virtual Machine (VM). JPF has listeners which get notified by the VM whenever a specific operation is performed. These listeners can be used to perform a specific operation and/or to further control the execution of the program. JPF explores states of the program along all possible paths and whenever end of the path is reached, it backtracks to explore other unexplored paths.

4.3.3 Partial Order Reduction

Partial Order Reduction (POR) is an optimization technique that exploits the fact that many paths are redundant, as they are formed due to the execution of independent transitions in different orders. In the case of database accesses, two operations are independent if both operations produce the same result and database state, regardless of the order in which they are executed.

Transition is the sequence of statements executed by a thread without interruption. There are different approaches to define transitions in terms of database accesses that range from coarse to fine-grained approaches. The coarse-grained approach considers a set of database accesses (which might be multiple) by a single thread as a transition, taking thread as a component. It explores

all possible interleavings of components.The naïve, fine-grained approach on the other hand, considers every database access as a transition. It trivially considers any two database accesses to be dependent and exhaustively explores the entire transition graph, which gives rise to the problem of state space explosion.

POR techniques identify dependent transitions and explore only a subset of the paths that are executed by the naïve approach. Thus we need to find all such cases where database access is independent to avoid checking of redundant paths. POR can be applied at different granularity levels from table, record to attribute (cell) level. We applied partial-order reduction at record level of the table using $RowId_i$. However, attribute level can be easily done by comparing columns of the tables as well.

4.4 Technique

We have modeled concurrent operations with a multi-threaded application. To identify when a thread is about to execute a statement that involves database access, we identify database access as an invocation of Java Statement class methods. Whenever we find such an invocation, we perform a dependency analysis. In the case of dependent database accesses, we mark the accesses as shared. Doing this for threads makes sure that if two threads have shared database accesses, then we will explore the other interleaving as well to test the non-deterministic behavior of the application. Our partial order reduction process has four parts:

1. Identify database access during state exploration.
2. Get instructions that are scheduled for execution and query database.
3. Perform dependency analysis and mark dependent database accesses.
4. Add JPF choice point (to explore alternative choices) for dependent database accesses.

The flow of our partial order reduction process is given in Fig. 4.8.

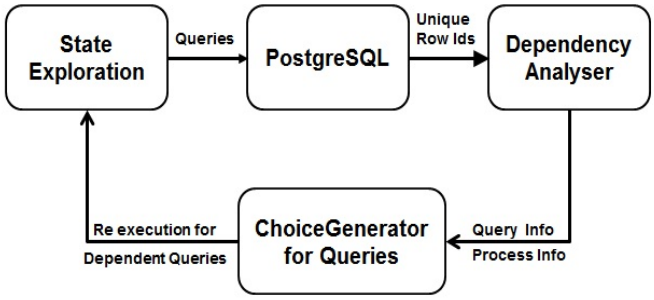


Figure 4.8: Partial Order Reduction Process

4.4.1 EPOR Handling of DB Operations

We run SQL statements in PostgreSQL to extract $RowId_i$ and then find the dependencies by analyzing the output of these queries. For each SQL statement Q executed on the database, where Q $\in$ { S = Select , D= Delete, I = Insert, U = Update }, we define the semantics of the output of these database accesses as follows:

1. The **SELECT** statement, returns a set of unique row identifiers of the rows that satisfy the condition c in 'WHERE' clause or all rows present in the relation in absence of the condition c. This can be given as $rowId(S) = \{RowId_i | i \in \{1, 2, ..., n\}\}$.

2. The **INSERT** statement returns a unique row identifier of newly inserted rows if insertion is successful, that is, $rowId(I) = \{RowId_i\}$.

3. The **UPDATE** statement returns the updated unique row identifier of the record, if it satisfies the condition c in the 'WHERE' clause, represented as $rowId(U) = \{RowId_i | i \in \{1, ..., n\}\}$.

4. The **DELETE** statement returns the unique row identifier of the deleted record if it satisfies the condition c in the 'WHERE' clause, represented as $rowId(D) = \{RowId_i | i \in \{1, ..., n\}\}$.

Table 4.2: Dependency Relations for Database Accesses

DB Operation	Dependence Relation	Execution Order	Conditions Used by EPOR to compute dependence	Dependence Relation (DPF)[a] [51]
(S_1, S_2)	No	-	-	-
(I_1, I_2)	No	-	-	-
(D_1, D_2)	No	-	-	-
(S_1, U_2)	Yes	(S_1, U_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$rowId(S_1)	Conditional Dependence
(U_1, S_2)	Yes	(S_2, U_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Conditional Dependence
(S_1, I_2)	Yes	(S_1, I_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$ rowIdS_1	Always Consider Dependent
(I_1, S_2)	Yes	(S_2, I_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Always Consider Dependent
(S_1, D_2)	Yes	(S_1, D_2, S_{1a})	rowId(S_1) $\cap$ rowId(S_{1a}) $\neq$ rowIdS_1	Conditional Dependence
(D_1, S_2)	Yes	(S_2, D_1, S_{2a})	rowId(S_2) $\cap$ rowId(S_{2a}) $\neq$ rowIdS_2	Conditional Dependence
(U_1, U_2)	Yes	(U_1, U_2)	rowId(U_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(I_1, U_2)	Yes	(I_1, U_2)	rowId(I_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(D_1, U_2)	Yes	(D_1, U_2)	rowId(D_1) $\cap$ rowId(U_2) $\neq \emptyset$	Conditional Dependence
(D_1, I_2)	Yes	(D_1, I_2)	rowId(D_1) $\cap$ rowId(I_2) $\neq \emptyset$	Conditional Dependence

[a]Conditional Dependence are less precise except at attribute granularity (See Section 4.5)

4.4.2 Instrumentation of PostgreSQL

To check the dependency conditions, we have extracted unique row identifiers of the records of the relation through instrumentation of PostgreSQL. After getting two database accesses for same relation by different threads, we actually execute queries by connecting to PostgreSQL server. Whenever PostgreSQL encounters an SQL statement, it prepares many possible execution plans to process the statement. It selects the optimal plan after estimating the cost of each plan and then executes it. During execution of the plan, in the initialization phase of the list of query nodes, we append an extra node to the list to get $RowId_i$ of the records returned by the query. After complete execution of the query, we gather $RowId_i$(s) of the accessed records returned by the query.

4.4.3 Detection of Operational Dependencies

For database applications, we track the dependencies among SQL operations at record level. We have defined conditional dependencies among SQL operations as in Table 4.2, in form of pair of database operations. Database accesses are represented by the first letter of the SQL operation. For

example, (S_1, U_2) defines sequence of database operation, i.e., SELECT by first process followed by UPDATE by second process. Dependence Relation 'Yes' or 'No' represents that both accesses are dependent or independent of each other, if they satisfy the condition given in 'Conditions used by EPOR' column, with the given execution order. The last column of Table 4.2 lists dependency relationship defined by the DPF [51]. Conditional dependence means that if the condition is satisfied by the function defined in DPF, only then, the pair of operations is dependent otherwise independent. Symbol '-' shows that there is no need to specify condition as both operations are independent of each other.

There are certain sequences of database operations that are always independent, e.g., two deletes represented as (D_1, D_2) are always independent, as the combined effect of both deletes remain same, irrespective of their order of execution. Similarly, two inserts (I_1, I_2) in a database does not result in non-deterministic database state. Since the output of the insertion operation is an addition of a new record in the table, so after execution of (I_1, I_2) or (I_2, I_1) database will have maximum two new records. It is quite obvious for two statements querying database that both will be independent statements as they are not affecting the resulting database state.

In case of two updates (U_1, U_2), database access could be dependent on the order of the execution. For example, there could exist one record R_o which satisfies the condition c_1 of U_1 and get modified such that updated value of R_o is R_u. When U_2 is executed, the modified value R_u does not satisfy the condition c_2 of U_2, whereas the original value of record R_o satisfies the condition c_2 of U_2. In this situation, the order of execution of U_1 and U_2 can lead to different database states. Thus, this operation is not independent and in order to test it against given specifications, we need to explore both schedules, that is, U_1 followed by U_2 ($U_1 \rightarrow U_2$) and U_2 followed by U_1 ($U_2 \rightarrow U_1$). In order to identify the dependent operations, we check the conditions specified in Table 4.2. For example, if two updates are dependent then they must be updating the same records. So intersection of unique row identifiers returned by each of these two queries will not be an empty set.

In operations that involve one SELECT statement, e.g., (U_1, S_2), execution order (S_2, U_1, S_{2a}) shows firstly we run SELECT operation followed by UPDATE, then we rerun SELECT given as

S_{2a} to apply dependency condition.

4.4.4 Implementation Details

As we mentioned earlier, we model parallel processes as different threads. Our partial order reduction technique builds on top of JPF. We trapped calls to methods of Statement class by intercepting all method invocations that access the database. JPF notifies our listener when it encounters executeQuery, executeUpdate or execute method of the Statement class. We extract the statement being executed by the call and parse it to obtain the list of tables accessed and access type of statement (READ or WRITE). We have defined a list structure to store the information we extracted. Every time we encounter these methods we traverse through our structure to find a pair of threads t_i and t_j, such that the instruction to be executed by each thread is execute method call of a Statement object. It then compares thread information along with their access type to perform dependency analysis. If both operation/queries performed by threads are dependent i.e. state of the database can be non-deterministic, then we mark the set of statements as shared in our data structure as

described in algorithm 2.

ALGORITHM 2: Marking Dependent Transition Algorithm

```
define C_dep as conditional dependence;
L ← dbAccessInfoList;
T_info ← Get information of current thread t;
M ← Invoked method from T_info;
if (M is a method of Statement class) then
    Q ← GetDBStatement(T_info);
    Rel ← TablesAccessed(Q , T_info);
    Access ← AccessType(Q);
    for each element E in L do
        if (E ≠ T_info) AND (E.Rel=T_info.Rel) AND ((E.Access OR T_info.Access) ≠
        READ) then
            (R_i, R_j) ← ExecuteQuery(E.Q, T_info.Q);
            C_dep ← DependencyAnalysis(R_i, R_j);
            if (C_dep) then
                MarkDBAccessShared(E, T_info);
    if ( ! (isPresent(T_info, L_d))) then
        add T_info to L;
```

In order to call methods of an object, Java uses the bytecode instruction 'INVOKEVIRTUAL'. We intercept the execution of 'INVOKEVIRTUAL' instruction every time it is called. For Statement methods, we check, for the running thread, if there exists any database access shared with any other thread. In presence of shared database access with one or more threads, we add a choice point for the model checker at this point, with all threads sharing database access as possible choices. This makes sure that the model checker makes schedules by pre-empting threads on these method

calls. Pseudocode is shown in algorithm. 3.

ALGORITHM 3: Partial Order Reduction Algorithm

```
define T_curr as active thread;
B ← ProgramByteCode;
L ← dbAccessInfoList;
L_d ← dependentAccessInfoList;
I ← read Instruction from B;
while (B ≠ end of Bytecode) do
    if (I = 'INVOKEVIRTUAL') then
        T_info ← ThreadInfo of T_curr;
        if (L ≠ 'NULL') then
            if (isPresent(T_info , L_d)) then
                if (!(ChoicePoint)) then
                    add ChoicePoint;
            else
                if (isDependent()) then
                    add T_curr to L_d;
        else
            add T_curr to L;
    I ← read next Instruction from B;
```

4.4.5 Backtracking and Database State Restoration

Execution of queries in a sequence will change the state of the database. For example, UPDATE followed by SELECT ($U \rightarrow S$) will change the database state. If SELECT and UPDATE are dependent accesses then we will explore other interleaving as well. Before exploring $S \rightarrow U$, it is important to restore the database to its original state i.e. state that was before execution of $U \rightarrow S$. For backtracking the database state, we exploit the savepoint and rollback mechanism of the database, so we can roll back the database state at a backtrack point. For each state in JPF, we

set a savepoint. We have mapped JPF state identifiers to savepoint in order to insert savepoint in database. When JPF backtracks restoring the memory state, we rollback to the respective savepoint, by using mapping of state identifiers.

4.4.6 Challenges and their Solution

There are significant challenges that we face during implementation and testing of database applications. We customize JPF to address all mentioned challenges and enable the exploration of database applications.

- One of the challenges that we faced during implementation phase was that JPF cannot simulate classes that use native code for their functionality, unless a model is written for the classes to translate objects sent and received to native code. Database accesses need drivers and in order to access and use these drivers, Java uses classes, that are not modeled in JPF. In order to get around this problem, we wrote wrappers classes required to communicate with database. The wrappers provide the same interface as the Java SQL classes but do not connect to the database like the actual classes. Instead these classes simulate database access.

- JPF sees accesses to shared memory objects as the only source of non-determinism. Since database does not load in its memory during program execution, JPF is unable to identify non-determinism in the processes that are accessing database. We have addressed this challenge by defining our own data structure in JPF to store information about shared database accesses.

- JPF does not see shared access to database. So, it does not consider transitions due to shared database access, as the scheduling points. Shared database accesses are the key scheduling points for database applications. Thus, in order to explore their possible schedules, we add a choice generator at these points.

Table 4.3: Number of States Generated

No. of DB Operations	No. of States		No. of States	
	Dep	Indep	Dep(DPF)	Indep(DPF)
2	12	10	15	9
3	14	12	21	14
4	33	22	27	18
5	59	44	70	38
6	56	40	183	40
7	111	53	257	48

4.5 Evaluation

4.5.1 Configurations and Benchmarks

Each thread of our implementation has its own local variables and can communicate with database through SQL statements. We experimentally evaluate the effect of Partial Order Reduction on the state space of program. We have shown that the number of states explored decreases from naïve exhaustive exploration, with much reduction. We have also compared our results with Database PathFinder (DPF)'s POR technique and shown that for record level granularity, our technique is more precise, as in some cases it gives better reduction in number of states and instructions executed.

All experiments were performed on a machine with a 3-core Intel Core i3-370M processor and 4GB of main memory, running Ubuntu Linux 14.4 and PostgreSQL 9.3.

Our set of benchmarks includes one live Java application PetClinic4, which is an official sample distributed with the Spring Framework[3] and implements an information system to be used by a veterinary clinic to manage information about veterinarians, pet owners, and pets. Secondly, our own created example of Employee Management System. We have implemented several operations to check effectiveness of our technique. Two operations are presented above in Section 4.2.

[3] `http://springsource.org/`

Table 4.4: Summary of Results for PetClinic for Dependent and Independent Operations

No. of Queries	Dep. Queries	Indep. Queries	States EPOR	States DPF	Time EPOR(ms)	Time DPF(ms)	Memory (MB)	Instructions EPOR
2	0	2	10	10	1200	1000	69	5277
2	2	0	12	12	1200	1000	69	5506
3	2	1	12	14	1400	1000	69	5634
3	0	3	10	14	1600	1000	69	5405
4	0	4	22	24	4000	4000	69	9136
4	2	2	26	29	6000	5000	99	9983
4	3	1	33	32	8000	7000	99	10792
5	2	3	50	68	19000	18000	99	18793
5	5	0	53	70	20000	19000	99	18826
7	2	5	100	182	72000	48000	129	40564
7	5	2	111	196	74000	52000	299	44888

PetClinic is basically a web based application. We modeled it by designing an interface through which we send commands to Java code containing application logic. Since PetClinic does not include concurrent cases (test case for two or more threads), we created concurrent test cases by combining operations in such a way that each operation is executed by one thread as one isolated process. Since, Java PathFinder works on a particular input, so we had to create set of inputs for our test applications. In order to have unbiased results we considered a range of values as input for test cases.

4.5.2 Discussion

We executed a number of threads with varied number of database accesses per thread for a range of input values. We encountered dependent and independent database accesses during execution. For independent operations number of states explored were less than that of dependent operations. It can be seen from results given in Table 4.3 that with increase in number of database accesses,

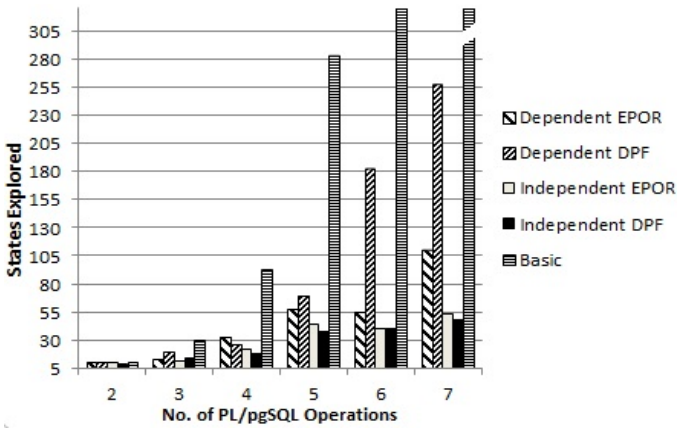


Figure 4.9: State space of Program for Dependent and Independent DB Accesses

the state space of program also increases for dependent operations.

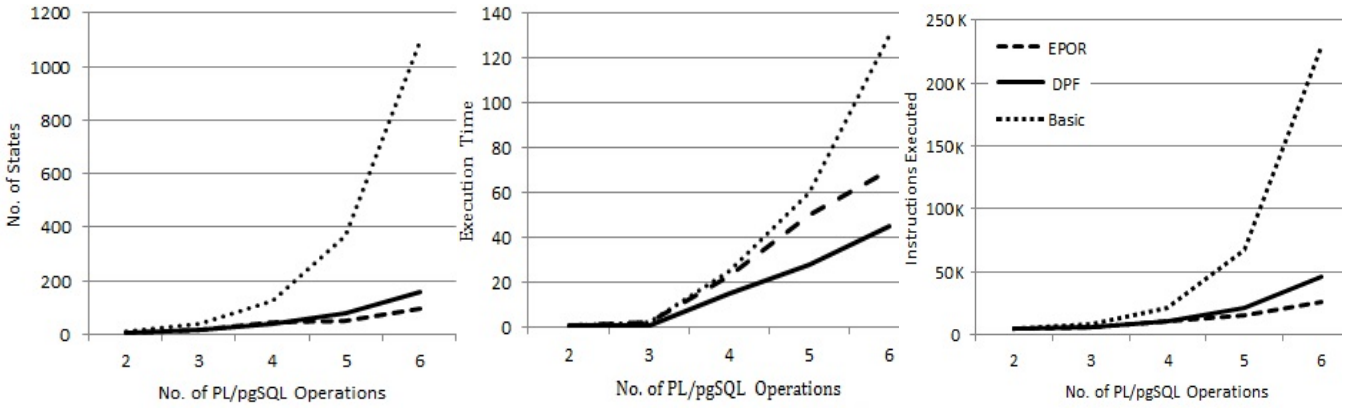
In order to evaluate performance of our technique, we compared EPOR with DPF for dependent and independent database accesses. We have seen that DPF approach takes pessimistic decisions about database accesses especially when it encounters a pair of INSERT and SELECT statement by considering it to be dependent access. However, we can also note that most of the times the number of states generated by our algorithm for independent operations were more than those generated by DPF. It is due to the fact that our algorithm does not consider disjoint attributes of same row as independent. So there is a possibility of marking accesses as dependent even when two disjoint attributes of same row are accessed, eventually generating more states. It is evident from Fig. 4.9 that increase in number of states for basic approach is exponential. There is significant decrease in states explored by EPOR for dependent operations which shows that our algorithm returns more precise results at record level granularity. Although it is quite trivial to implement same logic at attribute level, since we cannot improve on attribute level as compared to DPF, we restricted our scope to the record level granularity.

Summary of results for PetClinic given in Table 4.4 shows that our algorithm requires less time for the verification of independent accesses than that required by dependent database accesses. Hence, for programs with larger number of independent accesses as compared to dependent accesses will give significant reduction in state space of program.

Result of our second example Employee Management System depicts significant reduction in

Table 4.5: Summary of Results for Employee Management System

No. of DB Accesses	Independent Operations Time (ms)	Dependent Operations Time(ms)	States Indep.	States Dep.	Speedup	Reduction
3	2000	3000	18	23	1.5	1.27
4	23000	85000	42	66	3.7	1.57
5	50000	168000	52	86	3.36	1.65
6	70000	189000	69	93	2.7	1.34



(a) Number of States Explored

(b) Execution Time for State Space Exploration

(c) Number of Instructions Executed

Figure 4.10: Comparison of EPOR, DPF and naïve approach

number of state of system under test. In general, our technique gives on average, 1.4x reduction in state space of program. If we compare effectiveness of our algorithm for dependent and independent database access, we can see that if our program contains independent accesses then verification takes less time giving average speed up of 2.8x. With the increase in number of independent database accesses as well as number of concurrent operations, our technique will improve on overall speedup and reduction in state space of program as shown in Table 4.5.

While comparing with DPF, we present result of experiments that involves states explored, the number of instruction executed and the execution time. Results are given in Table 4.6. The number of instructions executed by DPF is more than our algorithm due to the fact that many independent

Table 4.6: Model Checking Employee Management System using DPF, EPOR and Basic Approach

No. of Queries	Ins. EPOR	Ins. DPF	Ins. Basic	States EPOR	States DPF	States Basic	Time (ms) EPOR	Time (ms) DPF	Time (ms) Basic
2	4154	4154	5050	4	6	12	1000	1000	1000
3	5671	5622	8378	18	16	40	2000	1000	2000
4	9874	9867	20996	42	38	122	23000	15000	25000
5	15209	20488	66940	52	78	366	50000	28000	60000
6	25681	46001	229550	93	158	1096	70000	45000	130000

accesses are marked dependent by the DPF algorithm, in Fig. 4.10c. However, our algorithm takes more time as given in Fig. 4.10b, for execution as number of processes increase because EPOR works on record level only and also due to re-execution which would only pay off for larger programs. For programs with less than 75 per cent database operations as dependent and remaining as independent will perform better in terms of execution time and reduction in unnecessary states of program due to the fact that for independent accesses only one schedule out of many will be explored.

Since our code for JPF is generic, our technique can easily be adapted to other database systems with instrumentation. Instrumentation of RDBMS is trivial if source code is available. Moreover, in presence of transactions, we can track transaction's commit in execute method and then apply EPOR on all queries present in the transaction by considering it a single operation.

Chapter 5

Testing NoSQL based database-driven Applications

5.1 Introduction

Many business enterprises use relational database management systems at the back end to manage data for the applications. New developments in the automation of computer technology led to use of data in huge amounts. Massive amounts of data is coming from different sources such as IoT devices, social media, streaming applications, cloud computing servers etc. Every second large amount of data is collected and stored for later analysis. Large organizations make use of this data to predict demand, make recommendations etc. Business decisions rely on the data stored at the back end. This data is unstructured in nature, hence traditional database management systems that deal with structured data are unable to store and manage this data efficiently [112]. This massive change in way of utilizing data in decision making has changed the paradigm in which data is stored or accessed. Scalable and more flexible databases are designed to address challenges posed by applications that make use of this data. NoSQL applications, as their name depicts, use no SQL for data manipulation and are built around these scalable databases. These applications are capable of storing records of variable length, allowing large amounts of data to be stored using relatively less space. Furthermore, NoSQL applications are schema less making them flexible

enough to adapt to the changes in the design of applications, even at runtime. NoSQL applications are gaining popularity and becoming an alternative to relational databases.

NoSQL applications have already been adopted in many organizations to make best use of Big-Data. NoSQL based database-driven applications are used in many scenarios. A typical scenario is an application that makes runtime decisions based on data collected from IoT devices, say a temperature sensor to address challenges in designing smart buildings. From single to multi user, there is a variety of applications that make use of this data stored in NoSQL databases. These applications manage huge amounts of data in document format using variable data models. Thorough testing of such applications is required for correct working of system. Due to scalable nature and complex structure of data, testing NoSQL based applications has become a challenging task. Though there are many techniques for testing database-driven applications, none of them address NoSQL databases. Much research attention is required in this field. Some work has been done on performance testing of NoSQL databases, some of which is related to inter-comparison of NoSQL database data models.

Symbolic execution, introduced in 1976 [82, 83], is powerful program analysis technique for automated test input generation. Symbolic execution works on the program's source code, and, by exploring its control flow graph, it generates path conditions for each path in the control flow graph. Path conditions are formed by collecting and combining constraints along each path. Each path condition is then solved using solvers to generate test inputs. These test inputs ensure execution of that specific path. Hence, symbolic execution aims at full path coverage of the program. Many state of the art constraint solvers are used to solve path conditions. A lot of progress has been made in improving the scope of these solvers. Model checking is another verification technique that explores the state space of program systematically for verification of properties. In case of an error it provides the user with error trail, i.e. a full path in program leading to that specific error.

In this work, we propose a technique for automated test case generation of NoSQL based database-driven applications written using Java. Focus of our research is document based applications that use MongoDB[1] [113] at the back end to store and manage data. We adapt dynamic

[1] `http://www.mongodb.org`

symbolic execution in conjunction with model checking to explore control flow graph of the program while generating path conditions.These path conditions are then solved to generate a test case. A test case for database-driven applications not only has program inputs but also contains the corresponding database state. We collect program as well as database constraints along path and form path conditions. We will evaluate our technique on sample programs and on-line applications.

5.2 Background

5.2.1 MongoDB: NoSQL database

There are many NoSQL databases used to address challenges posed by unstructured and complex data. Unlike relational databases, they do not enforce inter-relationship constraints. NoSQL databases are built upon different data models. These include key-value pair, document based, column based and graph based data models. MongoDB is a database system for managing large amounts of data for NoSQL applications. MongoDB supports document oriented databases where data is stored in documents. It can have nested or embedded documents. These documents have a unique ID in a collection. MongoDB, being NoSQL database stores data in JSON/BSON like structures. Unlike relational databases, MongoDB offers scalability by expanding horizontally. It can store structured, loosely structured or even unstructured data. Applications that use MongoDB usually create data models dynamically. These applications create schema on the fly without defining any thing prior to their implementation. Attributes of data are represented as a field. These fields, along with their values, are stored in JSON format as shown in Figure 5.1. This document contains three fields: `_id`, `name` and `salary`. Field names beginning with "_" character acts as unique identifier for a document. Here, `_id` uniquely identifies document in a collection. Usually, it is system generated number. Each document keeps record of field names and value stored in it. Different fields are combined together in a document which is in fact a record in a relational model. All documents form a collection representative of the table in traditional relational database system. This data model is quite flexible and can store arrays and other complex structures. It also has the ability to store hierarchical relationships. In order to link documents, embedded documents

are used.

```
"_id" : ObjectId("33a52bb7830b8c9b233b4fe6"),
"name" : "Adam",
"salary" : "3000"
```

Figure 5.1: Representation of a document in JSON format. Fields and values are shown as key value pair.

Each collection in NoSQL database can have multiple documents in it. Unlike the relational model, each document can store a variable number of fields in it. This gives flexibility to NoSQL databases and, on the other hand, saves space by not keeping fields of those data elements that have nothing as a value.

5.2.2 Symbolic Path Finder

Symbolic Path Finder (SPF) – an extension of classical Java PathFinder [114] [115, 116] – is an open source symbolic model checker designed for Java applications. It systematically works on byte code of Java program. It combines symbolic execution with model checking. It also provides constraint solving for automated test input generation. SPF offers inherent advantages of symbolic execution by providing high code coverage (e.g. path coverage) along with error benefits of model checking by detecting errors in highly concurrent environment. It also merges functionality for automated test input generation through constraint solving.

The key idea is to execute programs on symbolic input, which covers all possible cases of concrete input. Variables are represented symbolically using expressions with constraints on them that are extracted from program code during state space exploration of the program. These constraints are joined together to form path condition. Then, path conditions are solved to generate concrete input. It guarantees execution of that specific concrete path (for which constraints are solved), when program is executed using that concrete input.

SPF works by configuring itself using information described in configuration files. For example, In order to execute some method symbolically, user needs to specify information about

method in configuration file. SPF can treat whole class symbolically or it also gives an option to make certain variables or functions symbolic by providing information in configuration file. Even for symbolic program variables, range of input values can also be provided to the tool either using configuration files or through annotations.

5.3 Illustrative example

Consider example code given in Figure 5.2. This sample method is written in Java and has database calls to MongoDB. This code modifies employee's salary depending on the current salary of the employee. In order to modify database state, it interacts with MongoDB.

In Figure 5.2, lines 6-8 refer to database interaction, where database connection is established and then database and collection within the database are accessed. In order to search for a particular set of records meeting the search criteria, BasicDBObject is created which holds the search criteria given in line 10-11 of the code.

5.3.1 Loading configuration

In order to symbolically execute this application code, we define input parameters of method as symbolic in JPF configuration file. JPF loads configuration from application and project configuration files before analyzing the byte code of the Java application.

5.3.2 Systematic analysis: state space exploration

We explore the state space of program using JPF-core utilities. JPF creates states and maintains objects within these states (see Figure 5.3a). For variables that are to be treated as symbolic, it creates symbolic variables and stores them in state. These variables are then accessible from within all states that we define later on. Each state leads to another state (see Figure 5.3b). States have choice generator that stores number of choices leading out of the state. In case of simple language construct, say IF condition, there are only two paths out of the state, that is condition holds or its negation holds. In this case it will have only two choices as shown in Figure 5.3d. In

```
void addBonus(int maxSalary, int Bonus){
  int id = 0;
  int newSalary = 0;
   int bonus = Bonus;

  MongoClient mongoClient = new MongoClient();
  DB database = mongoClient.getDB("myMongoDb");
  DBCollection collection = database.getCollection("company");

  BasicDBObject searchQuery = new BasicDBObject();
  searchQuery.put("salary", new BasicDBObject("$lt", maxSalary));

  DBCursor cursor = collection.find(searchQuery);

  for(DBObject result:cursor)
  {
    String id = result.get(_id);
    newSalary = result.get(salary);
    newSalary = newSalary + bonus;

    BasicDBObject updateThisQuery = new BasicDBObject();
    updateThisQuery.put("_id", id);

    BasicDBObject newDoc = new BasicDBObject();
    newDoc.append( $Set , new BasicObject().append(Salary, newSalary));

    collection.update(updateThisQuery, newDoc);
    newSalary = 0;
  }
  mongoClient.close();
}
```

Figure 5.2: Example Code of database application using MongoDB to modify documents in collection. It modifies employees salary by adding bonus to it, if current salary is less than threshold salary passed as an input parameter

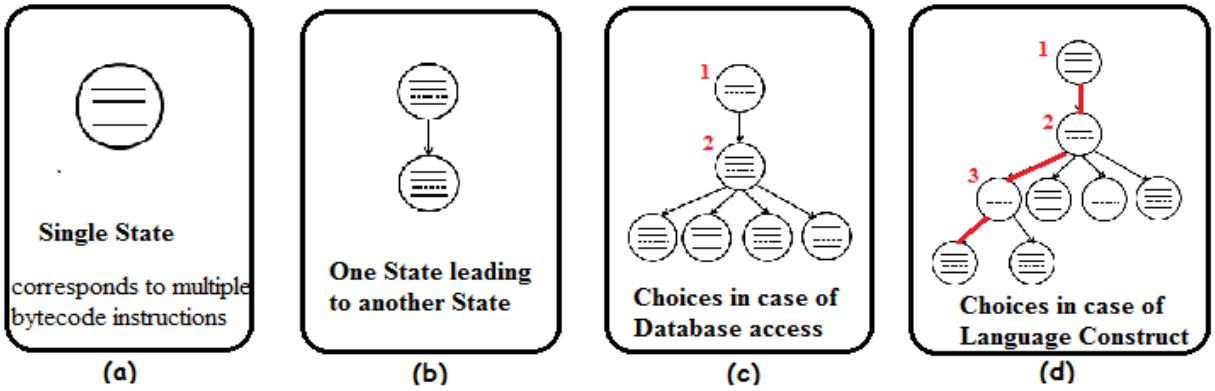


Figure 5.3: State space exploration of NoSQL based data driven applications. (a) Single state corresponds to set of byte code instructions executed without transition. (b) One state leading to another single state. (c) Multiple paths out of the state in case of database access. State 2 is a choice point from where execution can take multiple paths. (d) State 3 is also a choice point due to language construct. Path condition is formed by gathering constraints along the states, e.g. along states joined with red lines.

case of database accesses number of choices or paths leading out of the state depend on the number of symbolic documents in the collection. For two symbolic rows in the collection, we will have four possible choices as shown in Figure 5.3c.

5.3.3 Creation of symbolic collections

We intercept calls to the database in order to gather information about database, collection and documents. As soon as we reach line 6 of the code (see Figure 5.2), we intercept program execution to extract related information about the database. For existing databases and collections, we also extract field information to create respective symbolic fields. Our database is comprised of symbolic documents. Initially, we create two symbolic document for each collection. Later operations on database affect the number of symbolic documents. We store symbolic database within state. Here we create the symbolic collection “company” to store symbolic documents as shown in Figure 5.4.

```
{
     "_id" : id$sym1,
    "name" : name$sym1,
    "salary" : salary$sym1
},
{
    "_id" : id$sym2,
    "name" : name$sym2,
    "salary" : salary$sym2
}
```

Figure 5.4: Representation of a symbolic document in collection

5.3.4 Modification of path condition

While defining symbolic documents, we get information about fields from the database and create respective symbolic variables and impose constraint on them. Then, we modify path condition by appending constraints imposed on the fields of documents of a collection. Here, in Line 11 of the code in Figure 5.2, constraint is imposed on the field "salary' 'of the collection "company" using symbolic variable. Now we add condition "`salary < maxSalary$sym`" to the path condition as well.

5.3.5 Modification of choice generator set

Line 13 of the code in Figure 5.2, is a search query searching for documents in the collection. We treat 'find' methods as a choice point from where multiple paths lead out of the state. Since we have multiple symbolic rows, number of choices out of the state would be greater than two. In case of search query, we have different execution paths. Since we have two symbolic documents, we have four paths leading out of the state. For example in case of search method given in line 13, we will extract symbolic expression for the search criteria , i.e. `salary$sym1 < maxSalary$sym` and impose conditions on the document's fields. Here, we will have four different choices. (i) only first document satisfies the search criteria (see Figure 5.5), (ii) only second document satisfies the

```
salary$sym1 < maxsalary$sym
salary$sym2 >= maxsalary$sym
```

Figure 5.5: First document satisfies criteria

```
salary$sym1 >= maxsalary$sym
salary$sym2 < maxsalary$sym
```

Figure 5.6: Second document satisfies criteria

```
salary$sym1 < maxsalary$sym
salary$sym2 < maxsalary$sym
```

Figure 5.7: both document satisfy criteria

```
salary$sym1 >= maxsalary$sym
salary$sym2 >= maxsalary$sym
```

Figure 5.8: neither document satisfies criteria

search criteria(see Figure 5.6), (iii) both documents satisfy the search criteria(see Figure 5.7) and (iv) none of the documents satisfies the search criteria(see Figure 5.8).

At this point, we add choices to choice generator set and we mark it as a choice point. From here, now execution can lead to multiple paths. After declaring it as a choice point, we restart the execution of this instruction and JPF will automatically pick one choice from the choice set and will start exploring that particular path.

5.3.6 Solving path conditions

As soon as we reach end of the path, we get symbolic path condition expression from the state and send it to solver. We solve it to get test case.

5.3.7 Back tracking for further exploration

When JPF reaches at the end of the path, it backtracks to the state(s) that have unexplored choice. From there, it picks another choice from the choice generator set and starts exploration of program byte code till complete exploration of the state space of program.

5.4 Implementation Details

This section describes JPF-symbc functionalities that are required for symbolic execution of NoSQL based database-driven applications.

5.4.1 High level overview of symbolic execution of NoSQL based database applications

We execute byte code of our application using JPF, which systematically explores the state space of program whilst producing symbolic expression for path conditions. JPF loads configuration from JPF configuration files and Java class properties. We have modified listener to notify database accesses. On a database access, we perform two tasks. Firstly we extract information about documents fields to create symbolic variables and secondly, we mark current execution state as a choice point. Upon reaching end of the path, we solve path condition using constraint solver to generate test case and symbolic expression for path condition. High level overview of our technique is given in Figure 5.9

5.4.2 Symbolic Listeners

SPF runs byte code of Java program. It executes on top of the JVM and systematically explores byte code, making control flow graph of program in form of a tree like structure. Each node of the tree represents one state which is mapped onto one line of the program code. Listeners are objects that are notified whenever some VM or search event occurs. Symbolic listeners are used to define symbolic variables and constraint on them. In a typical application, meant to run on SPF, variables are declared symbolic either by using annotations in source code of program or by specifying in the configuration file of the program. In our case, we need to deal with the document's attributes in addition to the usual program variables. For this, we have intercepted exploration of the program state space whenever database event occurs. At this point we notify our symbolic listener. We have modified symbolic listener to extract information about database, collections with in it and documents as well. Here, we connect to the MongoDB to get information about collection and documents within that specific database. Using this information, we create symbolic database. In a case of search query, we also extract condition(s) imposed on the variable(s). Using this information, we impose condition on respective symbolic variable.

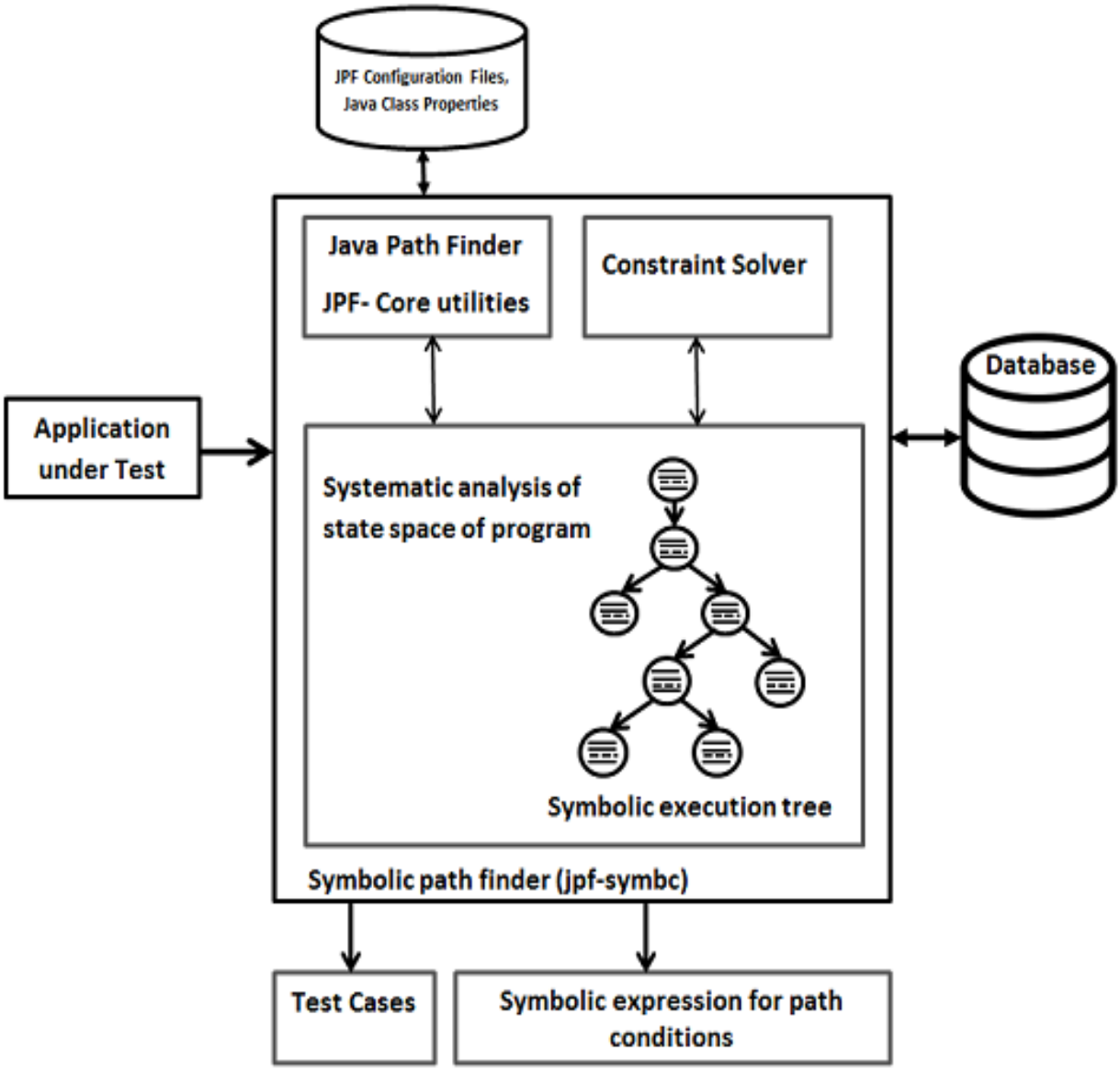


Figure 5.9: High level overview of our technique

5.4.3 Symbolic Choice Generators

During state space exploration of the program, each state leads to another state. There are some points in program when once state can lead to different scenarios. At this point, execution of program is not straight forward, as one state has more than one paths leading out of the state. This usually happens when conditional statements is encountered. This point is known as choice

point. Choice generators are used to create choices. Symbolic choice generators are used to create choices while creating and storing symbolic expressions based on the condition. Symbolic execution of database application is different from non database-driven application in a way that fields within documents are also treated symbolically. The constraint within search method is treated as a conditional statement and marked as choice point. A choice point is a place in state space of program where one state leads to more than one state. Here, in fact we explore both possibilities, that is condition and its negation both can be true. Thus, in case of find method, we intercept program execution and make current state as choice point. Unlike traditional conditional if condition, database constraints can lead to more than two options.

5.4.4 SPF for NoSQL database applications

Our approach aims to work on the byte code of Java application interacting with MongoDB. Algorithm for symbolic execution of NoSQL based database applications is detailed in Algorithm 4. We keep track of the program variables. As soon as program interacts with database, we intercept its execution to extract information about collection and documents. We further connect to database to get more information to create symbolic documents. We create two symbolic documents within a collection. Later on further operations on these documents can change the number of symbolic documents.We impose constraints on its fields and add the information in the respective state. Whenever new symbolic object is created, we modify path condition to append new constraints to existing path condition.

Similarly, while searching for particular document in the database, we extract constraint imposed on the specific field and express it as an expression. Then we append this expression to the path condition. Each state from this point will include this expression. Upon reaching end of the path, we get path condition from the state and using Z3 SMT solver, we solve it to get values for the database documents and method input variables. This makes up a test case. We backtrack to the state that have unexplored choices to explore other choice. In this way we generate test cases for method promising full path coverage.

ALGORITHM 4: Test Case Generation Algorithm for NoSQL based Database Applications

```
Data: Solver = New Path Condition Stack, State = New ChoiceSets Stack, T= Test case
      with random values
Result: Test Cases: test input and database state
for (all paths in the control flow graph of the program) do
    while (!pathFullyExplored) do
        T = getThreadInfo from current state;
        M = getMethodInfo from T;
        // get information about DB, collection ;
        if (Method is member of mongoClient) then
            DB = database accessed by client;
            DBCol = collection accessed by client;
        if (Method is member of DBCollection) then
            if (method is search query) then
                get search criteria;
                impose constraints on attributes;
                Mark choice point;
        if (Method is member of DBObject) then
            if (method is get or append or put) then
                Attr=get information about attributes of documents;
                Create symbolic database using DB, DBCOL and Attr;
                Add symbolic database to all states;
    print path condition;
    solve path condition and generate test case;
    backtrack to ChoicePoint to explore other path;
```

5.5 Related Work

NoSQL database application testing has gained less research attention as compared to SQL based database applications, however there are some tools available for the unit testing of such applications. NoSQLJUnit is NoSQL database application testing tool and is an extension of JUnit. It allows to load test data in database prior to application testing. It also allows to compare dataset after execution of data manipulation commands. Another state-of-the-art tool, Travis-CI, which supports a lot of NoSQL databases. It is a free SaaS platform. It checks your source code by compiling it. It also sets up the required databases to run tests. Public github repositories can be registered with Travis-CI [2] for testing.

NoSQLUNit is a JUnit extension that makes it very easy to manage life cycle and connections for a wide range of NoSQL databases like Cassandra, elastiscsearch, MongoDB, Redis and some more. It also enables you to load test data before test execution and compare data sets after test execution with few lines of code.

[2] `https://travis-ci.org/`

Chapter 6

Cross Platform Bug Correlation using Stack Traces

Crashing of program is an annoying experience for users. Whenever a program crashes, an event log is generated. Sometimes built in crash reporting programs send crash reports automatically to developing site whereas sometimes, user is presented with an option to report the crash himself. This reporting is often useful for the development team to diagnose and fix the problem. It happens quite often that code that crashes a program is not the reason for the crash itself but crash is due to the use of some faulty function or library of some other program. We propose a method to identify cross platform bug correlation to detect faulty functions, using function call stack information given in bug reports. We collect and process bug reports from multiple platforms to compute similarity based on our similarity metric between occurrence sequences of the function calls within different bug reports. For the solved bug we extract information about faulty function by analyzing its bug report to propose fix of the similar bug with same function within correlated bug report. If there exists fix of one bug in one application, it can be used to resolve similar bug in some other application. Using our technique we found similar bug reports. We were able to find cases for similar bug reports that have same reason for the cause of bug and were using same fix.

6.1 Introduction

Technology is ever evolving. Due to high competition in the software industry and keeping on going demands of customers in view, software companies release their product at a very early stage of development that makes it impossible to test applications properly. Due to this problem, such applications are highly prone to errors(bugs) that can even lead to program crash. Crashing of a program causes annoyance for its users. Software producers use different methods to know about these errors in order to find the root cause of the problem to fix the faults. Many companies are investing a lot of money, effort and time to get these bugs reported because eventually these reports will help developers to find and fix bugs. Mozilla has paid out over 1.6 million dollars in Bug Bounty Program[1] to its various researchers. Another way to report bugs is use of built in mechanism to report crashes automatically. Software such as Apple [53] and windows [54] use such mechanism to report crashes automatically to the development site. Some softwares provide an option to the user to report the error in form of a pop up window.

Fixing of a bug is not only a tedious task but also very expensive. Fixing of a bug has been an interesting topic for the researchers for almost one decade. It involves detailed analysis of structure of bug report [117], grouping of bugs, assignment of bugs to developers [118], [119] [120], fault localization and fault resolution. Grouping of bugs requires similar bugs to be placed in the same bucket. Then based on the number of bugs in each bucket, these bug reports are sent to the developing team for fault localization. When the faulty piece of the code is identified, developers resolve the bug. The process of reporting the bug to its resolution is resource constrained. Hence it gives rise to the need for automation of this process.

Many open source programs [52] have built in systems for bug reporting. These softwares receive millions of bug reports each day. Developers examine these bug reports to find the faulty piece of code in order to fix the bug. Crashes, due to these bugs appearing to happen in different locations might be correlated, i.e., due to same faulty function used in different places. Identification of correlated bugs is yet another challenging task that involves analysis of crash signature in particular stack trace of the bug report. Correlation between bug reports can be used to improve

[1] https://www.mozilla.org/en-US/security/bug-bounty/

```
1 nsCRT::strtok [mozilla/xpcom/ds/nsCRT.cpp, line 156]
2 nsMsgI18NParseMetaCharset [mozilla/mailnews/base/util/nsMsgI18N.cpp, line
    380]
3 nsMsgComposeAndSend::AddCompFieldLocalAttachments
    [mozilla/mailnews/compose/src/nsMsgSend.cpp, line 2431]
4 nsMsgComposeAndSend::HackAttachments
    [mozilla/mailnews/compose/src/nsMsgSend.cpp, line 2595]
5 nsMsgComposeAndSend::Init [mozilla/mailnews/compose/src/nsMsgSend.cpp,
    line 3381]
6 nsMsgComposeAndSend::CreateAndSendMessage
    [mozilla/mailnews/compose/src/nsMsgSend.cpp, line 4226]
7 nsMsgCompose::_SendMsg [mozilla/mailnews/compose/src/nsMsgCompose.cpp,
    line 965]
8 nsMsgCompose::SendMsg [mozilla/mailnews/compose/src/nsMsgCompose.cpp,
    line 1145]
```

Figure 6.1: Stack Traces from Bug Reports of Thunderbird

fixing of bugs.

It often happens that reason for crashing of two different applications is the single library function used by both of them. If fix is available for one application then it can be used to resolve issues in other application. The major focus of earlier approaches was on bug localization and bucketing of bugs [55] , [121] by finding similarities among bug reports from the same platform. There are techniques that address this problem but at a time for one product only. To our knowledge, none has addressed the problem for cross platform bug correlation, which can be helpful in diagnosing and resolving bugs of one application using fix of other application. Our key idea is to compute cross platform correlation between bug reports. We propose a method which takes all bug reports belonging to all products and then find correlated bug reports.

There are bug tracking systems that have on-line repositories that are used to store and keep track of each bug report. Bug reports once sent to the these systems are assigned to the developers for further processing.

```
nsCRT::strtok [d:\builds\seamonkey\mozilla\xpcom\ds\nsCRT.cpp, line 242]
MimeInlineTextHTML_parse_line
    [d:\builds\seamonkey\mozilla\mailnews\mime\src\mimethtm.cpp, line 192]
MimeInlineText_rotate_convert_and_parse_line
    [d:\builds\seamonkey\mozilla\mailnews\mime\src\mimetext.cpp, line 381]
convert_and_send_buffer[d:\builds\seamonkey\mozilla\mailnews\mime\src\mimebuf.cpp,
    line 123]
mime_LineBuffer
    [d:\builds\seamonkey\mozilla\mailnews\mime\src\mimebuf.cpp,line 245]
```

Figure 6.2: Stack Traces from Bug Reports of MailNews Core

Format of Bug Report Each bug report has unique identifier. It contains many fields including date of issue, status of bug report (Fixed, Verified etc), date of assigning, comment id, comments etc. Details of error and/or stack traces are part of the comment section. Each bug report can have multiple comments which can have zero or more stack traces in them.

Stack Trace is list of function calls with last function called on the top of the stack. Each line in the stack trace represents single function call and is called stack frame. Stack frame contains information about function, i.e., name, parameters, return type, line in the code at which function is located, file name, file path etc. Format of stack traces varies from application to application.

We make the following contributions

- **Automatic Extraction of Stack Traces:** We identified different types of stack frames and from the comments of the bug reports we automatically extracted stack traces of each type.
- **Similarity Metric for Bug Reports:** We define similarity metric for the stack traces in the different bug reports.
- **Clustering of Bug Reports:** We grouped similar bug reports from different platforms by tuning distance threshold to analyze faulty function.
- **Evaluation:** We evaluated our similarity metric on 877000 bug reports collected from Bugzilla[2].

[2] http://www.bugzilla.org/

Manual inspection of some of bug reports shows that bugs that are marked similar are due to the same fault.

6.2 Illustrative Example

In order to explain computation of similarity between bug reports, consider two stack traces extracted from two different bug reports from different products Thunderbird and MailNews Core as shown in Figure 6.1 and Figure 6.2. Here both stack traces have same format, i.e.,

`<FunctionName><FilePath><LineNumber>`

We extract information from each stack frame and store in a vector to analyze its contents in order to compute similarity between bug reports. Line 1 in Figure 6.1 and Figure 6.2 represents top stack frame. Initially we place both bug reports in same cluster irrespective of similarity between stack traces. We then do divisive hierarchical clustering using our similarity metric. At first level we compute similarity based on function names using `Levenshtein Distance`. It is clear from line 1 that function name in top stack frame is similar, i.e. nsCRT::strtok. Moreover, their distance to the occurrence of error is same. i.e., both are last function calls. So we put both bug reports in same cluster and perform second level of clustering. Here we use file name as similarity metric and compute similarity as described above. After top stack frame analysis these bug reports turn out to be similar.

6.3 Technique

Flow of execution of our technique is given in Figure 6.3. We collect bug reports from on line repository using Web API of Bugzilla. Then we extract stack traces from these bug reports and store information in a defined feature vector for each stack frame of stack trace of a bug. After preprocessing data we perform hierarchical clustering on our relational dataset to make sets of correlated bugs on the basis of stack trace similarity. Initially all bug reports belong to same set, i.e. cluster. We then split each set into subsets where each subset contains bug reports that are more similar to each other than reports in other subset. We analyze top stack frame by repeating

same procedure for file names and number and type of parameters in order to place similar bug reports in the same set. Bug reports in same set are analyzed to tune distance threshold. We also applied same technique given in Algorithm 6.1 on the top five stack frames.

6.4 Implementation details

Our method of data collection is described in Section 6.4.1, data cleaning in Section 6.4.2, similarity computation in Section 6.4.3, clustering of bug reports in Section 6.4.4, tuning of distance threshold in Section 6.4.5 and details of evaluation metrics in Section 6.4.6.

6.4.1 Data Collection

We extracted bug reports from Bugzilla, which is 'Bug-tracking System'. It allows developers to keep track of bugs in their products efficiently without any cost. We consider 132 products for bug report extraction. These includes 27 Mozilla products, 11 Firefox, Thunderbird, SeaMonkey, Rhino etc. Most of the applications are written mainly in C/C++. These products are used in the open-source community as well as in industry. Bug reports of 132 products are collected through web service and in XML format. It took a lot of time to collect 877000 bug reports comprising of 6818394 comments altogether.

6.4.2 Data Preprocessing

We processed collected bug reports to extract stack traces. Stack trace is a list of function calls made by the application when error is encountered. We randomly selected few bugs reports that contain keyword 'stack trace' or 'trace' and analyzed these reports. We observed that stack traces in the bug report generally do not have the same format. Format of stack traces varies from application to application. In the examined bug reports, we found several different formats of stack traces. We classified them into three different categories.

Type I - Event Reference Pair: These traces are in form of a series of event and reference pairs. Their format is like `<Class::FunctionName> <FileName>`. Regular expression for

```
Stack trace = Extracted stack trace from comment of Bug Id
BugId = Unique Bug Identifier
Set = All objects in one cluster
BugInfo= get file, func name, parameters of top frame form set
makeSetOfSimilarReports(Set, StoppingCriteria,Metric )
for each set
  if(!stoppingCriteria)
    while (all objects not visited in Set)
       if(first object)
  make subset
      else
         min dist = min. of distances form all subsets in cluster
         if(min dist less than distance threshold)
   add bug report to this cluster
        else
  make new subset with this bug report
     end while
     makeSetOfSimilarReports(Subsets, StoppingCriteria,Metric )
   else
      Return
end for
```

Algorithm 6.1: Clustering Algorithm for Bug reports

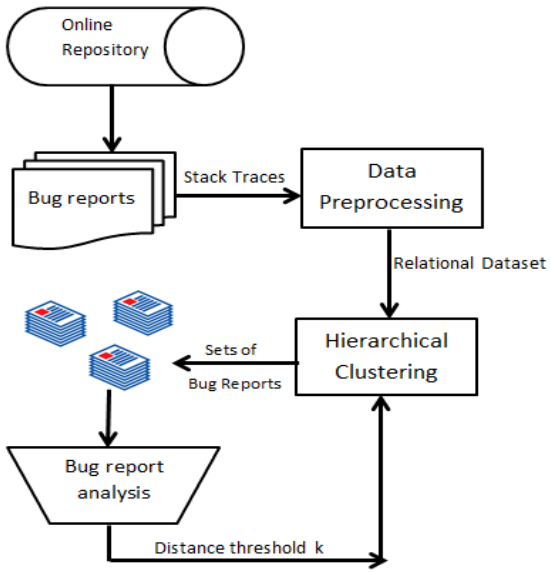


Figure 6.3: Bug Similarity Computation

these stack traces is divided into two types, one that includes keyword 'stack trace' and one that does not. Format of the stack traces that does not have keywork 'stack trace' is `<FileName> <Class::FunctionName> <Parameters> <LineNumber> <MemReference>`.

Type II- Event Function File Triplets: This type of trace takes the form of a series of events, functions and file triples. Their format is given as, `<MemReference> <FunctionName> <Parameters> <File Name> <Line Number>`. In this category we have some traces which are generated by GDB, while others are not, but have the same pattern. Regular expression for both types of patterns is calculated to get information from the stack traces.

Type III-Detailed Report: The type of traces which are arranged as some sort of detailed bug reports.

We extracted stack trace data from comments tags of XML files by parsing them using XPath in JAVA to compute correlation among bugs. Above mentioned regular expressions are used to separate three types of stack traces from the extracted information about bugs and stored in form of relational data. Table 6.1 lists the number of stack traces found for each type.

We define feature vector to store information and all stack traces are mapped to it. Details of features of data set along with data types are listed in Table 6.2. For each stack trace in a bug

Table 6.1: Stack Traces

Total Bug Reports	877000
Total Comment Retrievals	6818394
Total Stack Traces Found	17219
Type I Traces	6146
Type II Traces	2705
Type III Traces	8368

Table 6.2: Feature Vector

Data set Feature	Data type	Description
Product	String	Product name
BugId	Integer	Unique identifier for each bug
Status	String	New/Resolved/Assigned/reopened
commentId	Integer	Unique identifier for every comment in a bug
FilePath	String	Complete path of file
FileName	String	Name of file in execution
LineNo	Integer	Line number of executing line in a file
ClassName	String	Class name to which executing function belongs to
FuncName	String	Function name in execution
MemAddress	String (hexadecimal value)	Address of memory location
Offset	String(hexadecimal value)	Memory offset
funcParameter	List of strings	List of function arguments

report, we maintain a list of called functions along with their parameters in a sequence in which they appear in a stack trace.

6.4.3 Similarity Computation between Stack Traces

In order to compute similarity between stack traces, we define similarity metrics for stack frames. We have used function name, file name and number of parameters for computation of similarity to find correlation. We used *Levenshtein Distance*, which is a string metric for measuring similarity. We computed similarity firstly based on top stack frame and then on top five stack frames. For both, we computed similarity between name of the function, number of parameters and name of the file containing that function. For each function *i* in top stack frame or top five stack frames, we define similarity as function *i* is similar to the functions in cluster C_j and belongs to cluster C_j if the distance of *i* from C_j is minimum and less than distance threshold k. It can be given as,

$$\begin{cases} i \in C_j & \text{if } \underset{\forall \, \mathrm{j}}{Dist_{min}}(i, C_j) < k \\ i \in C_i & \text{otherwise} \end{cases}$$

Two bug reports are deemed similar if their stack frames has less than k Levenshtein distance where k is configurable constant and based on its value we cluster our bug reports. We tune k to place similar bug reports in same set.

6.4.4 Clustering of Bug Reports

We adopted divisive Hierarchical Clustering technique, which is essentially a top-down technique. Overview of this type of clustering is shown in Figure 6.4. Initially all bug reports are part of a single cluster. Then we divide each cluster into sub cluster by grouping bug reports using different similarity metric on each level. To sub divide our first cluster, we use function name of the top stack frame. Then for second level we used file name and for third level we used number of parameters to group similar bug reports in one cluster that we called a set. This kind of clustering helps to find true similarities between bug reports. For now, our stopping criteria is number of similarity metrics. We are using three similarity metrics, function name, file name and number of parameters.

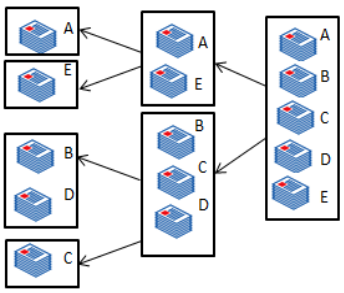


Figure 6.4: Overview of Hierarchical Clustering

6.4.5 Distance Threshold

For each of the similarity metric, we perform clustering with distance threshold, i.e., k. Initially we start with k set to 8, then gradually we decrease it. We analyzed bug reports in one cluster by keeping duplicate bug reports statistics to find best value for$k$. We found that with k =2 for function name and k =4 for file name similarity leads to grouping of similar bug reports in one cluster.

6.4.6 Evaluation Metric

For evaluating our clusters, we used *silhouette coefficient*, which measures how well object lies within its cluster. Silhouette coefficient uses cohesion and separation to measure effectiveness of clustering. Cohesion is how similar an object is to its own cluster whereas separation measures how dissimilar an object is to other clusters. Let $a(i)$ be the average distance or dissimilarity of a bug report with all other reports in its own cluster and $b(i)$ be the average dissimilarity of a bug report from bug reports of other than its own cluster then, silhouette coefficient $s(i)$ has value between 0 and 1. Closer the value to 1, better the clustering is. It is defined as,

$$s(i) = \begin{cases} 1 - \frac{a(i)}{b(i)}, & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ \frac{b(i)}{a(i)} - 1, & \text{if } a(i) > b(i) \end{cases}$$

.

6.5 Evaluation

6.5.1 Datasets

We collected data from online bug reports majorly reported on Bugzilla [122] and related to LLVM[3], Mozilla Products[4] etc. We transform information in these bug reports to feature vectors in a relational database.

6.5.2 Discussion

We applied our algorithm on the stack traces extracted from bug reports. In order to find correlated bugs, we analyze top stack frame and top five stack frames. Number of sets or clusters obtained for each with similar bug reports is shown in Figure 6.5. It is clear that if we take only top stack frame in account, then number of clusters are less but number of bug reports belonging to each cluster increase indicating presence of false positive. By applying two layer similarity function with top five stack frames, we observed that number of clusters increase whereas number of bug reports within each cluster decrease. This shows that if top five stack traces are similar there is high probability that bug reports belonging to same cluster are correlated. We inspected ten such sets. Manually inspection of bug report revealed that bugs grouped together were actually correlated, i.e., reason for error was the same. Even we found that few bugs were marked as duplicate of other bug in the same cluster.

We run our method with different values of k. For larger values of k, we get less number of clusters with very large amount of bug reports in each. Even for k as large as 8 we found there were few cluster that contain more than 242 similar bugs. But as we compared stack traces of those bugs manually, we saw there were false positives. Thus we decreased k. We executed our method for different values of k and for k =2, we found best results. Although total number of clusters increase with less bug reports within one cluster. Median of similar bug reports in a cluster is 3. There some sets with large amount of bug reports in them as shown in Table 6.3.

[3] `https://llvm.org/bugs/`

[4] `https://bugzilla.mozilla.org/`

Table 6.3: Effect of Distance Threshold on Number of Clusters and Bug Reports in each Cluster

Distance Threshold (k)	No. of Clusters	Max Reports in a Cluster	Avg. Reports per Cluster
8	1635	43104	242
6	2485	31054	34
4	5137	1612	3
2	5262	1137	3

Table 6.4: Tuning Distance Threshold k

Distance Threshold (k)	Cohesion	Separation	Silhouette Coefficient
2	2	22	0.909091
4	4	16	0.75
6	6	10	0.4
8	8	10	0.2

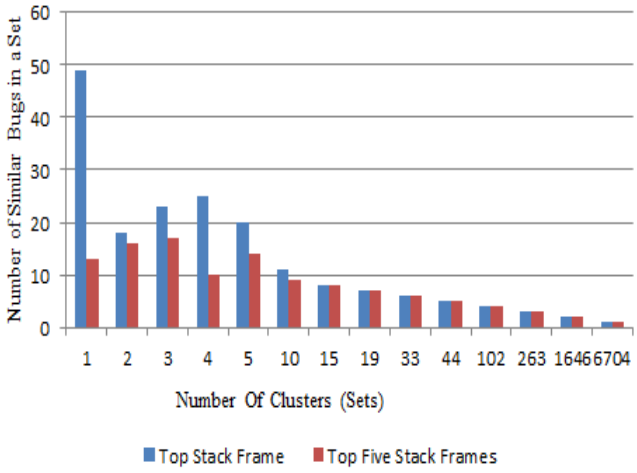


Figure 6.5: Similar Sets using Function Names

In order to evaluate our clusters, we used silhouette coefficient as shown in Table 6.4. It is clear from results that with k less than 4 yields better results. Silhouette coefficient for k = 2 is close to 1, which means that best clustering is achieved at this point. We even verified our clusters by manually analyzing bug reports.

6.6 Related Work

There are tools for bug reporting, such as Windows Error Reporting [54] , Apple Crash Reporter [53], Mozilla Crash Reporter [52] etc. These tools focus on collecting bug reports and then grouping similar reports together. Much work has also been done in the field of bug localization and bucketing of the bugs based on crash report. Bug localization [4] techniques are used to recover potentially buggy files from bug reports. Bucketing of bugs[5], targets to group similar bugs in a bucket which helps developer to prioritize fixing of the bugs, depending on the size of the bucket. However all of these techniques work on bugs related to one product at a time. In a closely related work on improving bug localization using correlations in crash reports [56], authors exploit significance of crash signatures, stack frames of traces and order of frequent subsets to create crash correlation groups (CCG).Crash Correlation Groups offer a diversity of crashing scenarios that help developers identify the cause of bugs more efficiently. Our work is different

from theirs in a way that they applied their technique on single application at a time, whereas we find correlation among stack traces of different applications.

In their paper, Yingnong Dang Rongxin and Hongyu proposed a method ReBucket [55], to cluster crash reports based on the similarity of stack traces. It firstly calculates similarity among stack traces and then based on similarity; it assigns the crash reports to appropriate buckets. They evaluated their technique using crash data of five Microsoft products Microsoft Publisher, Microsoft OneNote, Microsoft PowerPoint, Microsoft Project and Microsoft Access.

In another work for locating crashing faults, authors proposed a novel technique CrashLocator [57] for automatically locating crashing faults using stack traces. CrashLocator computes the suspiciousness scores of all functions by generating approximate crash traces through iterative stack expansions, using function call graph generated from source code.

Another approach [58] for correlated bugs revolve around call graph of program. This is altogether different from our work as according to their approach, two bugs are correlated if occurrence of one bug triggers other. Whereas focus of our work is to find two programs using same buggy function.

6.7 Future Work

We find correlated bugs belonging to different products that arise due to same fault. Finding correlated bugs will help to automatically fix the bug by using fix of one bug for other. Focus of this work was to find structural matching of function calls in stack traces of bug reports. We plan to add more metrics to compute similarity. Moreover this work can be extended to find semantic similarities among function calls present in stack trace.

Chapter 7

Applications of Program Analysis

7.1 Program Analysis for Embedded Devices

Monitoring CPU usage can provide useful insight to help checkpointing the system. We evaluate performance for C source code through static analysis of respective assembly code. Our target mote architecture for this purpose is MSP430 Von-Neumann architecture. We use instruction set for MSP430 architecture that is comprised of 27 core and 24 emulated instructions to compute CPU cycles. Emulated instructions are translated to core instructions without any overhead while execution. CPU cycles depend on the format of instruction as well as the addressing mode. There are three different instruction formats, with seven addressing modes for the source operand and four addressing modes for the destination operand to address the complete address space. Format I, Format II and Format III Instructions involves both source and destination operand, either source or destination operand and no operand respectively. All Jxx instructions take two CPU cycles.

7.1.1 Technique

We use options of GCC compiler to get the assembly translation of the C code along with source code and some additional information. We calculate CPU cycles required by each assembly instruction. Each source code line is mapped onto multiple assembly instructions. We use source code to assembly mapping to find CPU cycles required by each line of source code to evaluate cu-

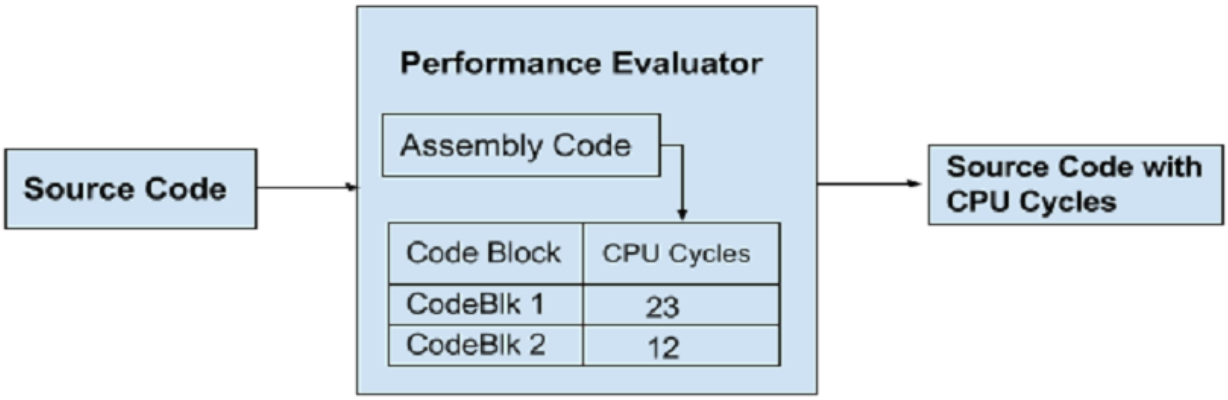
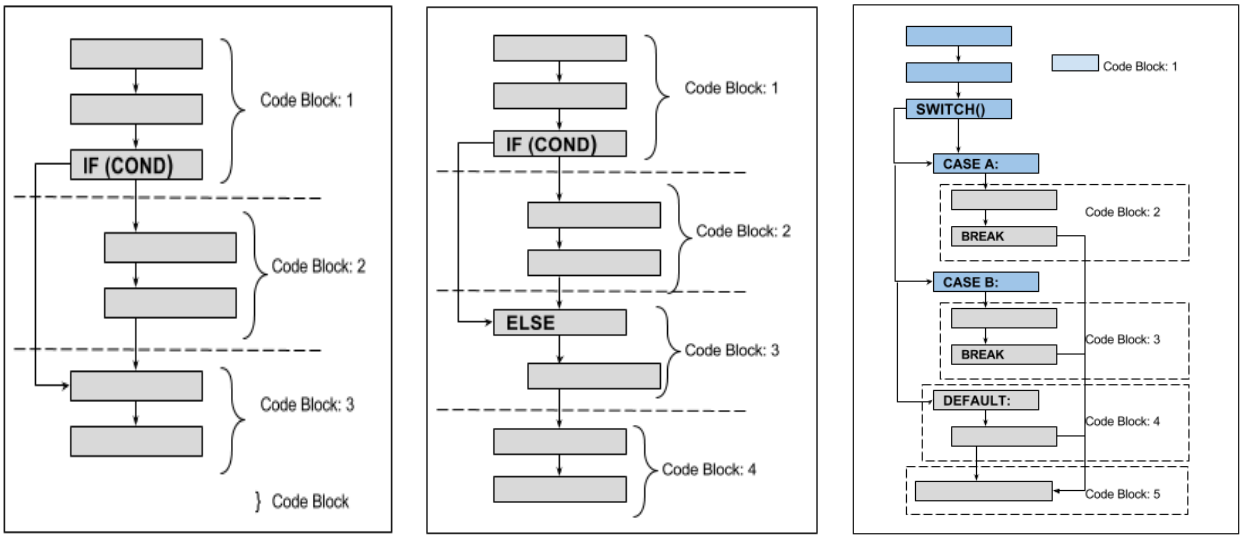


Figure 7.1: Flow of Performance Evaluation



(a) IF Condition (b) IF-ELSE Condition (c) Switch Statement

Figure 7.2: Branch Conditions

mulative CPU cycles for each block of code. We calculate performance for worst-case scenario by computing maximum number of cycles for each code block. Lastly, we append number of cycles for each code block in our source code.

7.1.2 Implementation Details

Code block: Overall program is divided into different code blocks. We define code block as the sequence of instructions until conditional statement encounters or program ends. These conditional statements can be branch conditions and/or loops. Statements within each branch forms a separate block. Similarly, we consider end of conditional block as code block boundary and statements after the conditional block are part of new code block.

Block of statements Sequence of instructions in a program until multiple execution paths reached form a block

Branch statements

If-statement: In case of if-statement as shown in Figure 1a, conditional statement is part of first block as based on this condition program execution will lead to conditional block or to statements after the conditional block. Conditional block makes another code block and instructions after the conditional block are part of third code block.

If-else-statement: For if-else-statements, we have separate code block for else part.

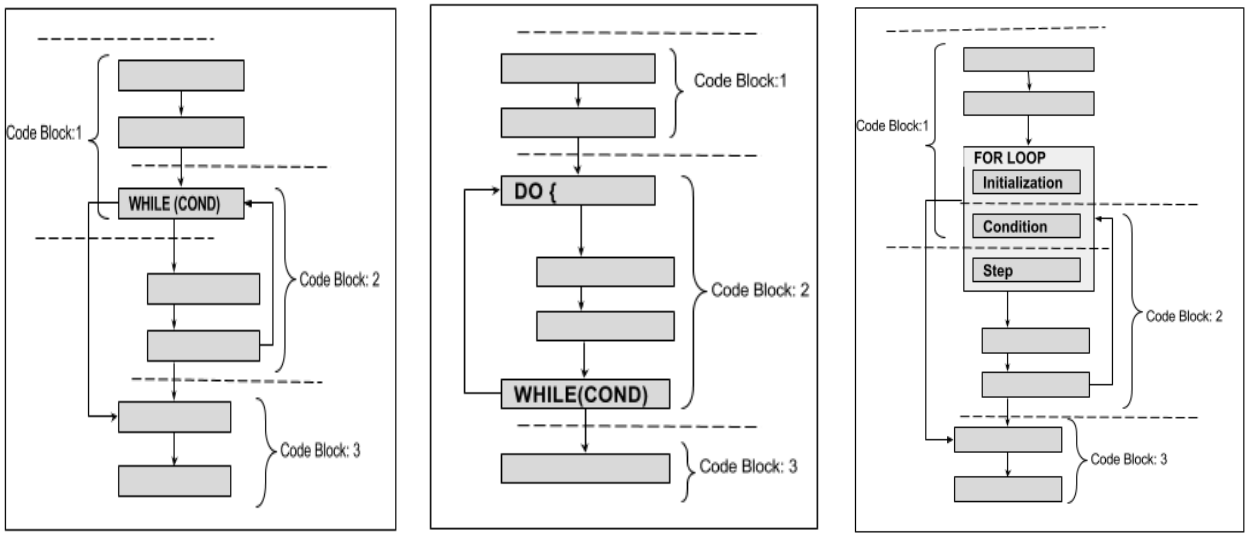
Switch statement: Switch statements are special case where all cases are part of the code block before switch statement due to the fact that before entering switch block, program will evaluate expression with given possibilities and then define respective targets. Statements within each case constitute separate code block as shown in Figure 1c.

Loops

While-loop: In case of while-loop, conditional statement is part of the previous block as well as the conditional statement block as conditional expression is evaluated before entering and while exiting the loop block as shown in Figure 2a.

Do-while loop: Unlike while loops where conditional expression appears at the beginning of the loop, do-while loop have conditional statement at the end of the loop. If condition is true, flow of control jumps to do-statement. Hence conditional statement is part of conditional block only as shown in Figure 2b.

For loop: For-loops are different in a way that they are comprised of three separate statements, i.e., initialization, condition and step statement. Here initialization is done before evaluation of



(a) WHILE Loop (b) DO-WHILE Loop (c) FOR Loop

Figure 7.3: Loops

conditional statement. Thus initialization and condition statement are part of previous block as shown in figure 2c. Since condition is checked again at the end of the loop thus it is part of loop block as well.

7.1.3 Identification of Code Blocks in Assembly Code

Conditional statements are translated to jump statements (i.e. jxx statements of MSP430). Evaluation of these conditional statements lead execution to different jump targets, i.e. labels in assembly code. It is difficult to identify code blocks based on jump instruction only because if the condition in conditional statement is true then there is no way to tell when conditional-statement code block ends. In this case statements within conditional-block and after the conditional-block will make one code block whereas in fact these are two different code blocks. In order to circumvent this situation we can safely assume jumps as well as their target labels as boundaries of code blocks. Thus, in assembly code we identify code blocks by keeping track of jump statements and their target labels. Whenever label is encountered in assembly code, we check if the label is in the list of jump targets then we mark occurrence of label as code block boundary.

Table 7.1: Assembly translation of C source code

....	
Line 8 : if(i > j)	8:ifonly.c **** if(i > j)
Line 9 : p= i+j;	cmp -8(r4), -6(r4)
Line 10: p =5+j;	jge .L2
.	9:ifonly.c **** p= i+ j;
	mov -8(r4), r15
	add -6(r4), r15
	mov r15, -4(r4)
	.L2:
	10:ifonly.c **** p =5+j;
	mov -6(r4), r15
	add #5, r15
	mov r15, -4(r4)

Branch statements: If-statements without else part will have only one jump instruction. Based on the criteria for code blocks given code in following figure will have three code blocks. Code Block 1: Line no 1 - 8 ; Code Block 1: Line no 9; Code Block 1: Line no 10 - end of program Labels that are jump targets are considered as code block boundary. We compute number of CPU cycles for these code blocks. In similar manner CPU cycles for if-else and if-elseif-else and switch statements are computed.

Loops

CPU cycles for loops are calculated in the same way as given above only with one exception that cycles for conditional statement are considered part of both the block before conditional statement as well as the conditional block itself. In case of while-loop and for-loop, conditional statement is part of the previous block as well as the conditional statement block. So CPU cycles for conditional statement are added to both blocks. In such cases, there is jump statement in assembly code corresponding to conditional statement in source code. Jump target then evaluates

conditional statement expression. Thus, jump target is stored in list of labels and whenever this label appear in the assembly code we compute CPU cycles for the label and add to both blocks. For do-while loop, as at the start of loop there is no jump instruction, hence we parsed source code line within assembly to mark start of loop as code block boundary.

We compute CPU cycles for each code block and store in a sorted hash table.

7.1.4 Appending CPU Cycles in Source Code

The hash table containing the clock cycles for each basic block is written out to a file on disk. To add the cycle counts to the source code, we read in this file, parse the data and store it in a sorted Map. Then we read in the source code file, and at the end of each basic block we append the number of cycles for that basic block, which includes the line numbers in that basic block, as a comment. For example:

root2 = (-b-sqrt(determinant))/(2*a); //Lines: 16 - 18; Cycles = 7

A new file is then generated which contains the newly appended information.

7.1.5 Results:

We verified number of cycles taken by each assembly line by using MSPSim utility. Our number of cycles computed are over estimation of CPU cycles required to run the program. In case when conditional statement is conjunction of multiple statements then there is quite a possibility that only one condition executes on run time. Sometimes execution of one conditional statement is sufficient to decide whether to execute conditional block or not. For example in code given below, when i = 1, j= 3; conditional expression will return FALSE because (i ¿ 2) is FALSE. On run time execution of program will jump to instructions after the conditional block without evaluating second expression.Here in this example, CPU cycles computed by our program is 15 where as with above scenario CPU cycles may reduce to 8.

7.1.6 Future Work

Our performance evaluator evaluates number of cycles required to execute the C code on a telosb mote. We calculate number of cycles for MSP430 architecture. We will extend our technique to find the energy consumed by the program so that we can checkpoint the system before energy of the system runs out.

7.2 Program Analysis for Tracking the Trackers

Malwares are spreading widely and becoming a threat for security. Apart from traditional techniques such as heap spray, modern malware also tracks user for launching exploit. Apart from malwares there are several other motives behind tracking users. One of the major purposes is behavioral advertising, used by Ad Network to present user with ads related to his interest. Another interesting dimension involves price discrimination in order to present different priced items to different users. Even in the presence of privacy add-ons and disabled cookies, these websites are able to track users. Browser fingerprinting is well known technique for tracking user uniquely. These tracking websites exploit fingerprints to identify user. We present a technique that will detect whether these websites are spoofing user's computer configurations, details about system internals and what information is accessed by these websites. This will not only indicate what information is required by such sites to fingerprint user but also help to dig out the sites that are tracking user without user's consent; User is sure that he is not tracked. This will eventually assist to classify trackers in different categories based on the type of fingerprinting information they are extracting. Moreover, it will give better understanding of the trackers and problems that they can give arise to. Tracking information can be used by oraganisations to protect against malicious activities. We used our technique to generate logs of information accesses by Alexa top 10,000 websites. Analysis of these log files will lead to interesting results.

7.2.1 Introduction

Malware is widely accredited as a growing threat now a days with many samples reported each day with many new malwares different from the existing ones. There is incredible diversity in characteristics and behavior of malwares. Each new and unknown malware is dissimilar from the previous in a sense that new and unknown malware are more precise towards their target. These ever evolving malwares uncover loopholes in security that needs to be addressed and are becoming threat on a global scale.Modern malwares do not rely on well-known techniques such as heap spray, embedded binary code (for execution purpose) or code obfuscation. There are many counter measures already defined for identification and circumvention of infected codes by such techniques. To make malicious code undetectable, malware authors apply these techniques at different layers using multiple granularities.

Uniquely identifying user on the Internet is closely related to user personalization. Ad Networks usually track users to personalize user in order to present user with ads that are of his interest or based on user's past activity. Datta et al [123] throws lights on Google's Ad settings to show how tracking information can be used to present Ads to users. Lecuyer et al presented a tool [124] which can be used to infer what are the most useful attributes which can be used to personalize user. Another interesting dimension which involves personalization is price discrimination. It is used to present different prices to differ users or to present different products within specific price range to users. In this area much work has been done to expose sites which present users with different prices of same products [125] [126]. Sometimes malware authors work more intelligently by extracting information related to user in order to come up with unique identifier that represents user on the world wide web as unique entity through tracking user. These malware authors exploit the fact that information about computer's configuation can be easily extracted while user is browsing through a website.There are many sites who want to steal user information. They track user on Internet. Many sites compromise on user anonymity. Users are anonymised on the Internet either by tracking their computers configuration or by keeping record of browsing sites through use of cookies. Users are often tracked to present with inaccurate information or to block access to different sites.

Since long, web analytics services collect basic web browser configuration information in order to analyze web traffic and identify variants of click frauds. Client side scripting languages such as java script, or other programs such as Flash can be used to collect much more esoteric parameters. These parameters are multifarious in nature, e.g., information about installed fonts, language, add-ons, screen resolution, environment variables etc. This information is integrated to make a profile of user also known as fingerprint tied with pattern of characteristic rather than specific tracking cookie.

Whenever web page is loaded, certain information about device and browser is broad casted to not only the visiting website but also to any trackers that are embedded within the webpage, e.g., advertisements. Online service such as VirusTotal[1] uses information form other anti virus engines to detect malicious content in files and URLs.

7.2.2 Background

Million of Internet users are using privacy tools to block trackers. Some of such tools are Ghostery, AdBlock and Disconnect. Despite the use of these tools, malware authors are tracking user to infect, redirect or to use their machines to participate in launching exploit. Panopticlick analyzed how well users are protected against online tracking and concluded that even in the presence of privacy add-ons, security is vulnerable if browser fingerprint is unique [127].

Browser fingerprint is information collected through the configuration and device setting information visible to the websites for the purpose of identification to track web browser. In the absence of cookies these fingerprints can be used to identify individual users or devices on the Internet. It is different from traditional tracking that make use of IP addresses and unique cookies. Another approach was to link browser sessions [128] together to get information. Some techniques generate fingerprint by analyzing the pixels produced in canvas element [129]. These canvas elements are produced by rendering text and WebGL scenes. Another class of fingerprinting is browser independent [130] as it relay mostly on installed fonts, screen resolution and timezone. However, it is evident from experimental evaluation that reliable browser identification is crucial for online

[1] `https://www.virustotal.com/`

Listing 7.1: Javascript Code for Browser Engine Name

```
<p id="demo"></p>
< script >
document. getElementById ("demo").innerHTML = navigator . product ;
</ script >
```

Listing 7.2: Javascript Code for Browser Engine Name accessed through variable

```
<p id="demo"></p>
< script >
var x = 'd' + 'o' + 'c' + 'u ' + 'm' + 'e' + 'n' + 't';
var y = eval(x);
y. getElementById ("demo").innerHTML = navigator . product ;
</ script >
```

security and privacy [131], e.g., regarding drive-by downloads and user tracking, and can be used to enhance the users security.

Much work has been done from different perspectives to track the users. Recent work in form of tool OpenWPM [132] is done by Steven and Arvind, which measured tracking on about one million web sites taken from Alexa. In their work, they analyzed accesses to fingerprinting information. They have collected tracking information based on both stateless and stateful fingerprinting. They have also recorded accesses to WebRTC, HTML Canvas font, Audio and Battery Fingerprinting information to measure and analyse tracking information gathered by different sites. Our work is different from theirs in a way that we have also tracked events as well as number of instructions between these accesses.

7.2.3 Design Details

We present automated way to check whether website is trying to fingerprint browser or not. Whenever visiting site try to access information about device or browser, our technique not only detect request for unnecessary information but also will tell what information is requested by the web site. We have used Firefox browser to keep track of the access made to the internal objects. Firefox is an open source browser [2] and has SpiderMonkey as an interpreter for javascript. In following sections, we discussed in detail instrumentation of browser and collection of information.

Extraction of information

Fingerprinting attributes can either be directly requested using their name fields or some times extracted indirectly in order to mislead user. For example, to extract the version of the browser, web-page can use JavaScript with command directly as shown in listing 7.1, or indirectly by first storing the variable in form of string of characters and then accessing through some other variable after evaluating it, as given in listing 7.2. In order to intercept these polymorphic calls, we need to extract information from lowest level, at which browser is returning requested data. We will collect attributes used for fingerprinting and form one big set comprised of all possible fingerprinting information collected above.

Instrumentation

We instrumented Firefox source files to record accesses to three different elements required to track the trackers.

1. **Stateful and Stateless Fingerprinting**

   There are generally two types of fingerprinting techniques stateful and stateless. Stateful is cookie based tracking whereas stateless is system attribute based tracking; Attributes are gathered through JavaScript accesses. Other than keeping track of the cookies, mainly

[2] `https://archive.mozilla.org/pub/PROJECT/releases/RELEASE/source/`

we tracked window.navigator, window.screen, HTMLCanvasElement and CanvasRenderingContext2D objects. We recorded all accesses to these objects and store respective key value pairs. For Example, for browser it stores ('Browser', 'Mozilla'). We collected large variety of attributes that are required in order to make a fingerprint. We gather information from various sites about fingerprinting information and attributes that contribute to fingerprint generation. Fingerprint information includes user agent string, HTTP ACCEPT headers, Screen resolution and color depth, timezone of computer, browser plugins, versions, installed fonts, system platform, timestamps, isjavaenabled, support for flash objects and cookies.

2. **HTTP Request Object**

   We also tracked all requests sent from the loaded page by keeping track of all HTTPRequest object. Whenever request for some other resource is generated; it is recorded in our trace log. For this purpose we tracked change in state of AJEX objects. Changes to REQUEST_SEND and REQUEST_SENT objects are recorded.

3. **Events**

   In order to record to and from communication of the webpage with the outer world, we also recorded events. Tracking of events in conjunction with other two elements described above will help to classify the trackers, i.e. whether their intension is to spread malware or for improved behavioral analysis for Ad Network. Combination of incoming and outgoing events with fingerprinting information can lead to interesting findings.

4. **Number of Instructions**

   We kept check on number of instruction between different accesses mentioned above. Further investigation of number of instructions between these accesses can lead to interesting results. Specific access patterns may help in classifying the tracker.

Collection of Trace Log

We accessed Alexa top 10,000 websites automatically using instrumented browser and recorded accesses to instrumented objects. Trace log for all these sites is collected for further analysis.

Tools

We used Selenium[3] for automation of websites. **Selenium** is an open source tool that is used for automation of web browsers. It supports use of multiple browsers ranging form light weight limited functionality browsers such as PhantomJS to full fledge modern browsers such as Chrome, Internet Explorer and Firefox as well. We used **JSsoup**[4], an open source project of MIT, to parse pages to extract links of top websites. It is a Java library which is designed to work with DOM components within HTML. It is well suited for applications that require scarping, parsing and manipulating of HTML components. It provides easy access to data between different elements of web page using simple methods. Data of document can be navigated using DOM like methods of JSoup library.

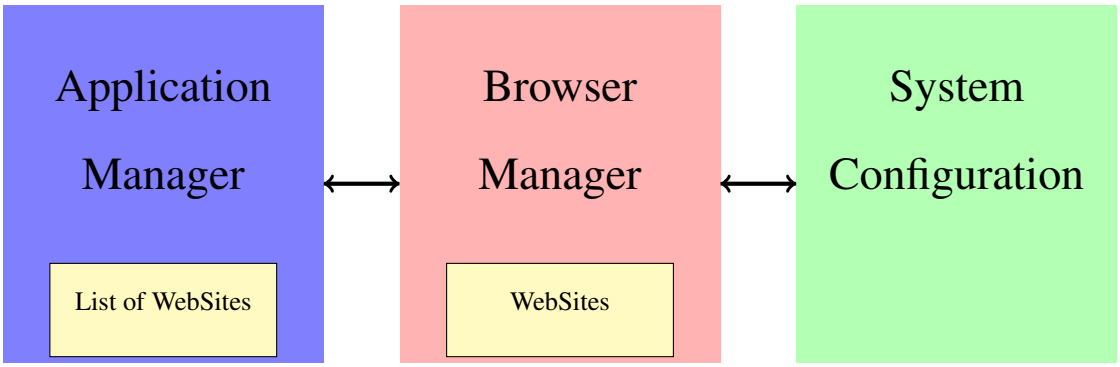


Figure 7.4: Overview of System Architecture

7.2.4 Technique

We know that information is accessible only through browser. Our browser on one hand has access to information about our computer such as operating system, even environment variables. On other hand our browser is interfacing with website as shown in Figure 7.4. Whenever we visit some website, we perform detailed analysis of the web page that is running in the browser. Our technique will effectively analyze various component present in the webpage and their interaction with external environment through use of browser. Our technique will generate log of all such

[3] http://www.seleniumhq.org/
[4] https://jsoup.org/

interactions. For now we dont have any gold standards about what sites are using what kind of fingerprinting information in order to track user. There are two main components of our application. Application Manager and Browser Manager. Firstly, Application Manager starts by visiting Alexa to extract top 10,000 sites. We use Java library to parse Alexa to extract links to home page of these websites and then to store in a list. Then in a serial fashion, Application Manager takes out these links from list and delegates to Browser Manager. Browser Manager loads webpage into instrumented browser and all accesses to instrumented browser are recorded in a file.

Against each website, respective file is created in the directory contain log entries . Each webpage has a trace log against it and it is stored in a file with a name same as that of the link of website. Once done with one site, Browser Manager gives control back to Application for further exploration. Application Manager repeats this until tracelog for all links is generated.

Logs file contains accesses to internal information ; It includes fingerprinting information such as UserAgent, HTTPRequest object, events along with their types and number of instruction executed between these accesses. Right now, Browser Manager is not running like the real world program in a sense that it is not interacting with users through mouse clicks, keyboard interactions, scrolling etc. But in order to take full advantage of instrumentation, we plan to interact with these sites to capture detailed information especially HTTP Requests and Responses.

Then we will analyze extracted log to see what attributes are requested by the particular website, in order to report user that he has been tracked.

7.2.5 Evaluation

Experimental Setup All experiments are performed on Intel Core i5 Machine with 8GB RAM. We instrumented Firefox 48 browser source code. In order to automate whole procedure we have used Selenium 3 and to make it work with Firefox, we used separate driver, gecko that interacts with Firefox browser. In order to parse HTML documents, we used JSoup which is HTML parser in Java. We have tested our technique on top 10,000 web sites from Alexa. These sites were opened in browser and their log files are stored in a directory for later analysis.

Discussion Collection of trace logs for all 10,000 sites took almost 4 days. Initially, collection

of data was much slow due to waiting for response on the pop up windows and many a times crash of selenium driver. This problem was later on circumvented by storing list of all 10000 sites on hard disk and then loading files in a batch of 500 files. This technique reduced a lot of time but again time taken by the pop up windows was large enough. For that we set timeout for loading websites that enables loading of websites relatively fast. In future, we plan to address this issue in some other way because firstly interaction with site may lead to interesting logs and secondly, placing timeout may prevent recording of useful information. Although time taken for collection of trace log was long enough but since we will collect this data occasionally, we can spend this much of time. Moreover this data is required only once to check whether site is tracking the user or not. We have also observed that sites that contains large number of links took long to load into the browser and have larger log files in the end then that of those which have less number of links within them.

7.2.6 Future Work

Measuring tracking on the Internet has been a topic of interest for a decade. In recent years much progress has been made in this area. We present a method to measure tracking information by instrumenting Firefox browser source code for fingerprinting information. We ran our application on Alexa top 10k sites. Whenever these sites access fingerprinting information for navigator, screen, HTMLCanvasRendering2D, events, HTTPRequest objects, we record key value pair in our tracelogs. In future, we plan to classify malicious sites based on the pattern in which theses sites are extracting user information. By using information in trace logs, we will classify trackers in different categories such as malware authors, Ad networks, user personalizations, behavioral advertising etc.

Chapter 8

Conclusion

database-driven applications are a necessary requirement in today's computing environment. It is essential that they work correctly, hence their testing is an important part of their implementation. Manual testing of such applications is time consuming and error prone, making automated testing a much more suitable option. Typically, automated test case generation aims at high code coverage. Testing database-driven applications involves larger set of possible inputs. Due to presence of data within the databases beside usual program inputs, full path coverage is a difficult task. Tools that provide full path coverage eventually deal with many redundant test cases. In this thesis, we adapt symbolic execution for automated test case generation of database-driven applications.

Firstly, we propose a technique for testing of stored procedures using symbolic execution. Stored procedures are modular programs including database accesses and reside within the database server. Stored procedures are based on classical relational database models that interact with database using standard query language commands. For this, we instrument PostgreSQL to extract constraints directly from within the database, hence generating more precise test cases efficiently. Our symbolic executor generates database states and program input as test case. Our evaluation on open-source projects show that our technique is able to generate test cases for complete procedures by modeling tables and associated constraints in a reasonable time. Results of our comparative analysis depict that our approach is efficient in comparison to existing approaches. However, it can give rise to scalability issues for large number of queries within procedure.

Database applications, when used in highly distributed environments and accessed by multiple clients impose challenges on their testing due to possible inconsistent database state caused by different order of execution of requests by the clients. We address this challenge by introducing effective partial order reduction in model checking database applications. Our technique increases average speed of execution by a factor of 2.8 for independent accesses. It also reduces the to-be-explored state space of the program.

database-driven applications are also extensively used with NoSQL databases. Testing NoSQL based database applications is arduous due to complex data structure, flexible and variable nature of NoSQL databases. For these applications, we propose a hybrid approach of using symbolic execution with model checking. We systematically explore control flow graph of the application using a model checker and gather constraints along a path to form path condition. We formulate path condition as a symbolic expression for each path. Later on we solve these symbolic expressions to generate test cases. This ensures full path coverage whist generating precise test cases for the database-driven applications based on NoSQL applications.

In future, we plan to extend our coverage of SQL grammar and functional testing of stored procedures for end-to-end automated testing of PostgreSQL. We further intend to provide support for complex nested queries. We also intend to adapt symbolic execution to generate test for embedded documents for NoSQL based databases. Other research direction which we aim to proceed is to localize fault in such procedures due to database constraints and to suggest its repair.

There are many test generation techniques for database-driven applications that adapt symbolic execution for test input as well as test database generation. It is evident from the evaluation that our symbolic executor provides full path coverage and able to generate test cases efficiently. However, in the case of unbounded loop iterations, symbolic execution might not be able to cover all scenarios. Also, we are exploring each database constraint and its effect on the execution path, which may lead to a large number of test cases for the same path.

Bibliography

[1] K. Pan, X. Wu, and T. Xie, "Guided test generation for database applications via synthesized database interactions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, p. 12, 2014.

[2] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "Relational symbolic execution of sql code for unit testing of database programs," *Science of Computer Programming*, vol. 105, pp. 44–72, 2015.

[3] M. A. Mohamed, O. G. Altrafi, and M. O. Ismail, "Relational vs. nosql databases: A survey," *International Journal of Computer and Information Technology*, vol. 3, no. 03, pp. 598–601, 2014.

[4] N. Matthew and R. Stones, *Beginning Databases with PostgreSQL.* Apress, 2005.

[5] K. Pan, X. Wu, and T. Xie, "Database state generation via dynamic symbolic execution for coverage criteria," in *Proceedings of the Fourth International Workshop on Testing Database Systems*, p. 4, ACM, 2011.

[6] S. A. Khalek and S. Khurshid, "Systematic testing of database engines using a relational constraint solver," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp. 50–59, 2011.

[7] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pp. 73–82, 2011.

[8] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pp. 249–260, 2008.

[9] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pp. 10–10, 2010.

[10] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 151–162, ACM, 2007.

[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[12] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[13] C. Baier and J.-P. Katoen, *Principles of model checking*. The MIT Press, 2008.

[14] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," pp. 337–340, 2008.

[15] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *European Conference on Object-Oriented Programming*, pp. 504–527, Springer, 2005.

[16] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM systems journal*, vol. 22, no. 3, pp. 229–245, 1983.

[17] J. Zhang, C. Xu, and S.-C. Cheung, "Automatic generation of database instances for white-box testing," in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pp. 161–165, IEEE, 2001.

[18] C. Binnig, D. Kossmann, and E. Lo, "Reverse query processing," in *IEEE 23rd International Conference on Data Engineering (ICDE)*, pp. 506–515, 2007.

[19] M. Veanes, P. Grigorenko, P. De Halleux, and N. Tillmann, "Symbolic query exploration," in *Formal Methods and Software Engineering*, pp. 49–68, Springer, 2009.

[20] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing SQL database queries," *Journal of Software Testing, Verification and Reliability*, vol. 20, no. 3, pp. 237–288, 2010.

[21] C. De La Riva, M. J. Suárez-Cabal, and J. Tuya, "Constraint-based test database generation for SQL queries," in *Proceedings of the 5th Workshop on Automation of Software Testing*, pp. 67–74, 2010.

[22] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[23] C. Li and C. Csallner, "Dynamic symbolic database application testing.," in *DBTest*, 2010.

[24] D. Chays, J. Shahid, and P. G. Frankl, "Query-based test generation for database applications," in *Proceedings of the 1st international workshop on Testing database systems*, p. 6, ACM, 2008.

[25] Y. Deng, P. Frankl, and D. Chays, "Testing database transactions with AGENDA," in *Proceedings of the 27th international conference on Software engineering*, pp. 78–87, 2005.

[26] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "A relational symbolic execution algorithm for constraint-based testing of database programs," in *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 179–188, 2013.

[27] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "Towards testing of full-scale SQL applications using relational symbolic execution," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pp. 12–17, 2014.

[28] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *Software*

Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 31–40, IEEE, 2013.

[29] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*, pp. 134–153, Springer, 2008.

[30] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, pp. 213–223, ACM, 2005.

[31] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 263–272, ACM, 2005.

[32] A. Neufeld, G. Moerkotte, and P. C. Lockemann, "Generating consistent test data for a variable set of general consistency constraints," *VLDB J.*, vol. 2, no. 2, pp. 173–213, 1993.

[33] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 289–292, ACM, 2010.

[34] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[35] J. Tuya, M. J. Suarez-Cabal, and C. De La Riva, "Sqlmutation: A tool to generate mutants of sql database queries," in *Mutation Analysis, 2006. Second Workshop on*, pp. 1–1, IEEE, 2006.

[36] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *Proceedings of the 8th International Workshop on Automation of Software Test*, pp. 111–117, IEEE Press, 2013.

[37] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing sql injection attacks in stored procedures," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pp. 191–198, 2006.

[38] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker, "A framework for testing database applications," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 147–157, 2000.

[39] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "Agenda: A test generator for relational database applications," *Department of Computer and Information Sciences, Polytechnic University, Brooklyn, NY, Tech. Rep*, 2002.

[40] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, 1997.

[41] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pp. 251–262, 2012.

[42] M. Martin and M. S. Lam, "Automatic generation of xss and sql injection attacks with goal-directed model checking," in *Proceedings of the 17th Conference on Security Symposium*, SS'08, pp. 31–43, 2008.

[43] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 474–494, July 2010.

[44] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, 2000.

[45] W. Visser, C. S. Psreanu, and S. Khurshid, "Test input generation with java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[46] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *ACM Sigplan Notices*, vol. 40, pp. 110–121, 2005.

[47] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032. 1996.

[48] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[49] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, "On race vulnerabilities in web applications," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pp. 126–142, 2008.

[50] Y. Zheng and X. Zhang, "Static detection of resource contention problems in server-side scripts," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 584–594, 2012.

[51] M. Gligoric and R. Majumdar, "Model checking database applications," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 549–564, 2013.

[52] "Mozila Crash Reports, 2012." "`http://crashstats.mozilla.com`".

[53] "Technical Note TN2123: CrashReporter, 2010." "`http://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html`".

[54] A. Ganapathi, V. Ganapathi, and D. A. Patterson, "Windows xp kernel crash analysis.," in *LISA*, vol. 6, pp. 49–159, 2006.

[55] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: a method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1084–1093, IEEE Press, 2012.

[56] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 286–295, ACM, 2005.

[57] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 204–214, ACM, 2014.

[58] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pp. 247–256, IEEE, 2013.

[59] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[60] Y. Agarwal, B. Balaji, R. Gupta, J. Lyles, M. Wei, and T. Weng, "Occupancy-driven energy management for smart building automation," in *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, pp. 1–6, ACM, 2010.

[61] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 8–pp, IEEE, 2006.

[62] Y.-J. Tu, W. Zhou, and S. Piramuthu, "Identifying rfid-embedded objects in pervasive healthcare applications," *Decision Support Systems*, vol. 46, no. 2, pp. 586–593, 2009.

[63] O. Chipara, C. Lu, T. C. Bailey, and G.-C. Roman, "Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit," in *Proceedings of the 8th ACM conference on embedded networked sensor systems*, pp. 155–168, ACM, 2010.

[64] J. Jun, Y. Gu, L. Cheng, B. Lu, J. Sun, T. Zhu, and J. Niu, "Social-loc: Improving indoor localization with social sensing," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, p. 14, ACM, 2013.

[65] R. Huuck, "Iot: The internet of threats and static program analysis defense,"

[66] L. Invernizzi, S. Miskovic, R. Torres, C. Kruegel, S. Saha, G. Vigna, S.-J. Lee, and M. Mellia, "Nazca: Detecting malware distribution in large-scale networks.," in *NDSS*, vol. 14, pp. 23–26, 2014.

[67] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks.," in *USENIX Security Symposium*, pp. 169–186, 2009.

[68] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection.," in *USENIX Security Symposium*, pp. 33–48, 2011.

[69] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 443–457, IEEE, 2012.

[70] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 203–222, Springer, 2004.

[71] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "Hercule: attack story reconstruction via community discovery on correlated log graph," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 583–595, ACM, 2016.

[72] S. Arzt and E. Bodden, "Stubdroid: Automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 725–735, ACM, 2016.

[73] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 106–117, ACM, 2015.

[74] P. Ilia, I. Polakis, E. Athanasopoulos, F. Maggi, and S. Ioannidis, "Face/off: Preventing privacy leakage from photos in social networks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 781–792, ACM, 2015.

[75] J. M. Such and N. Criado, "Resolving multi-party privacy conflicts in social media," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 7, pp. 1851–1863, 2016.

[76] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 169–180, ACM, 2016.

[77] M. Fredrikson and B. Livshits, "Repriv: Re-imagining content personalization and in-browser privacy," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 131–146, IEEE, 2011.

[78] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE Computer Society, 2013.

[79] S. Ji, S. Yang, T. Wang, C. Liu, W.-H. Lee, and R. Beyah, "Pars: A uniform and open-source password analysis and research system," in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 321–330, ACM, 2015.

[80] A. Silberschatz, H. F. Korth, S. Sudarshan, *et al.*, *Database system concepts*, vol. 4. McGraw-Hill New York, 1997.

[81] M. A. Ghafoor, M. S. Mahmood, and J. H. Siddiqui, "Effective partial order reduction in model checking database applications," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pp. 146–156, IEEE, 2016.

[82] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," vol. 2, pp. 215–222, May 1976.

[83] J. C. King, "Symbolic Execution and Program Testing," vol. 19, pp. 385–394, July 1976.

[84] C. Barrett and C. Tinelli, "CVC3," in *Proc. 19^{th} International Conference on Computer Aided Verification (CAV)*, pp. 298–302, 2007.

[85] N. Sörensson and N. Een, "An Extensible SAT-solver," in *Proc. 6^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 502–518, 2003.

[86] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Proc. 9^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 553–568, 2003.

[87] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[88] D. A. Ramos and D. R. Engler, "Practical, Low-Effort Equivalence Verification of Real Code," in *Proc. 23^{rd} International Conference on Computer Aided Verification (CAV)*, pp. 669–685, 2011.

[89] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based Repair of Complex Data Structures," in *Proc. 22^{nd} International Conference on Automated Software Engineering (ASE)*, pp. 64–73, 2007.

[90] C. Seo, S. Malek, and N. Medvidovic, "Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems," in *Proc. 11^{th} International Symposium on Component-Based Software Engineering*, pp. 97–113, 2008.

[91] M.-Y. Chan and S.-C. Cheung, "Testing database applications with sql semantics.," in *CODAS*, vol. 99, pp. 363–374, 1999.

[92] M. S. Mahmood, M. A. Ghafoor, and J. H. Siddiqui, "Symbolic execution of stored procedures in database management systems," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pp. 519–530, IEEE, 2016.

[93] L. A. Clarke, *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[94] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[95] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013.

[96] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice Experience*, vol. 30, pp. 775–802, June 2000.

[97] P. Godefroid, "Compositional Dynamic Test Generation," in *Proc. 34^{th} Symposium on Principles of Programming Languages (POPL)*, pp. 47–54, 2007.

[98] C. Cadar and D. Engler, "Execution Generated Test Cases: How to make systems code crash itself," in *Proc. International SPIN Workshop on Model Checking of Software*, pp. 2–23, 2005.

[99] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. 13^{th} Conference on Computer and Communications Security (CCS)*, pp. 322–335, 2006.

[100] J. H. Siddiqui and S. Khurshid, "ParSym: Parallel Symbolic Execution," in *Proc. 2^{nd} International Conference on Software Technology and Engineering (ICSTE)*, pp. V1: 405–409, 2010.

[101] M. Staats and C. Păsăreanu, "Parallel Symbolic Execution for Structural Test Generation," in *Proc. 19^{th} International Symposium on Software Testing and Analysis (ISSTA)*, pp. 183–194, 2010.

[102] J. H. Siddiqui and S. Khurshid, "Staged Symbolic Execution," in *Proc. 27^{th} Symposium on Applied Computing (SAC): Software Verification and Testing Track (SVT)*, 2012.

[103] J. H. Siddiqui and S. Khurshid, "Scaling Symbolic Execution using Ranged Analysis," in *Proc. 27^{th} Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2012.

[104] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution," in *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*, pp. 504–515, 2011.

[105] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized Symbolic Execution," in *Proc. 2012 International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2012, pp. 144–154, 2012.

[106] K. Pan, X. Wu, and T. Xie, "Program-input generation for testing database applications using existing database states," *Automated Software Engineering*, vol. 22, no. 4, pp. 439–473, 2015.

[107] K. Tsumura, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Pairwise coverage-based testing with selected elements in a query for database applications," in *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*, pp. 92–101, IEEE, 2016.

[108] P. McMinn, C. J. Wright, and G. M. Kapfhammer, "An analysis of the effectiveness of different coverage criteria for testing relational database schema integrity constraints," *Department of Computer Science, University of Sheffield, Tech. Rep*, 2015.

[109] M. J. Suárez-Cabal, C. de la Riva, J. Tuya, and R. Blanco, "Incremental test data generation for database queries," *Automated Software Engineering*, vol. 24, no. 4, pp. 719–755, 2017.

[110] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for sql queries," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 1230–1230, ACM, 2018.

[111] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[112] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pp. 15–19, IEEE, 2013.

[113] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[114] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 134–138, Springer, 2007.

[115] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[116] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 179–180, ACM, 2010.

[117] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 27–30, ACM, 2008.

[118] O. Baysal, M. W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pp. 297–298, IEEE, 2009.

[119] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 131–140, IEEE, 2009.

[120] M. M. Rahman, G. Ruhe, and T. Zimmermann, "Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 439–442, IEEE Computer Society, 2009.

[121] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 333–342, IEEE, 2011.

[122] "Bugzilla Bug Repository." `http://www.bugzilla.org`.

[123] A. Datta, M. C. Tschantz, and A. Datta, "Automated experiments on ad privacy settings," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 1, pp. 92–112, 2015.

[124] M. Lécuyer, G. Ducoffe, F. Lan, A. Papancea, T. Petsios, R. Spahn, A. Chaintreau, and R. Geambasu, "Xray: Enhancing the webs transparency with differential correlation," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 49–64, 2014.

[125] A. Hannak, G. Soeller, D. Lazer, A. Mislove, and C. Wilson, "Measuring price discrimination and steering on e-commerce web sites," in *Proceedings of the 2014 conference on internet measurement conference*, pp. 305–318, ACM, 2014.

[126] T. Vissers, N. Nikiforakis, N. Bielova, and W. Joosen, "Crying wolf? on the price discrimination of online airline tickets," in *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, 2014.

[127] P. Eckersley, "How unique is your web browser?," in *International Symposium on Privacy Enhancing Technologies Symposium*, pp. 1–18, Springer, 2010.

[128] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," *Proceedings of W2SP*, vol. 2, pp. 180–193, 2011.

[129] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," *Proceedings of W2SP*, 2012.

[130] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *Nordic Conference on Secure IT Systems*, pp. 31–46, Springer, 2011.

[131] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien, "Fast and reliable browser identification with javascript engine fingerprinting," in *Web 2.0 Workshop on Security and Privacy (W2SP)*, vol. 5, 2013.

[132] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1388–1401, ACM, 2016.