Documentation for schemaFuzz

Ulrich "Feideus" Erwan

August 24, 2018

Documentation For SchemaFuzz

1 Summary?

This document actually needs a front page.

2 Introduction

SchemaFuzz is a free software command line tool incorporated inside the Gnu Taler package which is a free software electronic payment system providing anonymity for customers. The main goal of this project is to provide an effecient debbuging tool that uses a "fuzzing" strategy oriented on databases. Where a traditionnal fuzzer would send malformed input to a program, SchemaFuzz modifies the content of a database to test that program's behavior when stumbling on such unexpected data.
Obviously, this tool is meant to be used as a mean of debugging as the goal is to pop buggs or put into light the security breaches that the code may contain regarding the retrieving, usage and saving of a database's content. As this tool is being developped as a master's thesis project, its current state is far from being finished and there are many options and optimisations that deserve to be implemented that are not yet available. These future/missing features will be detailed and discussed in a dedicated section.

3 Context and Perimeter

SchemaFuzz's developpement enrolls in the global dynamic of the past decades regarding internet that sustain great efforts to make it a more fluid, pleasant but more importantly a safer space.
It uses the principle of "fuzz testing" or "fuzzing" to help find out which are the weak code paths of one's project.

> Traditionnal fuzzing is defined as "testing an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program".

This illustation is very well illustated by the following example :

> Lets's consider an integer in a program, which stores the result of a user's choice between 3 questions. When the user picks one, the choice will be 0, 1 or 2. Which makes three practical cases. But what if we transmit 3, or 255 ? We can, because integers are stored a static size variable. If the default switch case hasn't been implemented securely, the program may crash and lead to "classical" security issues: (un)exploitable buffer overflows, DoS, ...

It is declined in severals categories that each focus on a specific type of input. UI fuzzing focuses on button sequences and more genericly any kind of user input during the execution of a program. The above exemple falls into this category. This principle had already successfully been used in existing fuzzing tool such as the well known "American fuzzy loop". File format fuzzing generates multiple malformed samples, and opens them sequentially. However, SchemaFuzz is a database oriented fuzzer. This means that it focuses on triggering unexpected behavior related to the usage of a external database content
This tool is meant to help developpers, mainteners and more genericly anyone that makes use of database comming from a database under his influence in their task. A good way to summerise the effect of this tool is to compare it with an "cyber attack simulator". This means that the idea behind it is to emulate the damage that an attacker may cause subtly or not to a database he unlegitly gained privileges on. This might in theory go from a simple boolean flip (subtle modifications) to removing/adding content to purely and simply destroying or erasing all the content of the database. SchemaFuzz focuses on the first part : modification of the content of the database by single small modification that may or may not overlap. These modifications may be very aggressive of very subtle. It is intresting to point out that this last point also qualifies SchemaFuzz as a good "database structural flaw detector". That is to say that errors typically triggered by a poor management of a database (wrong data type usage, incoherence beetween database structure and use of the content etc ...) might also appear clearly during the execution.

3.1 Perimeter

This tool is based on some of the SchemaSpy tool's source code. More precisely, it uses the portion of the code that detect and stores the target database's structure. The main goal of this project is to build on top of this piece of existing code the functionnalities required to test the usage of the database content by any kind of utility. The resulting software will generate a group of human readable reports on each modification that was performed.

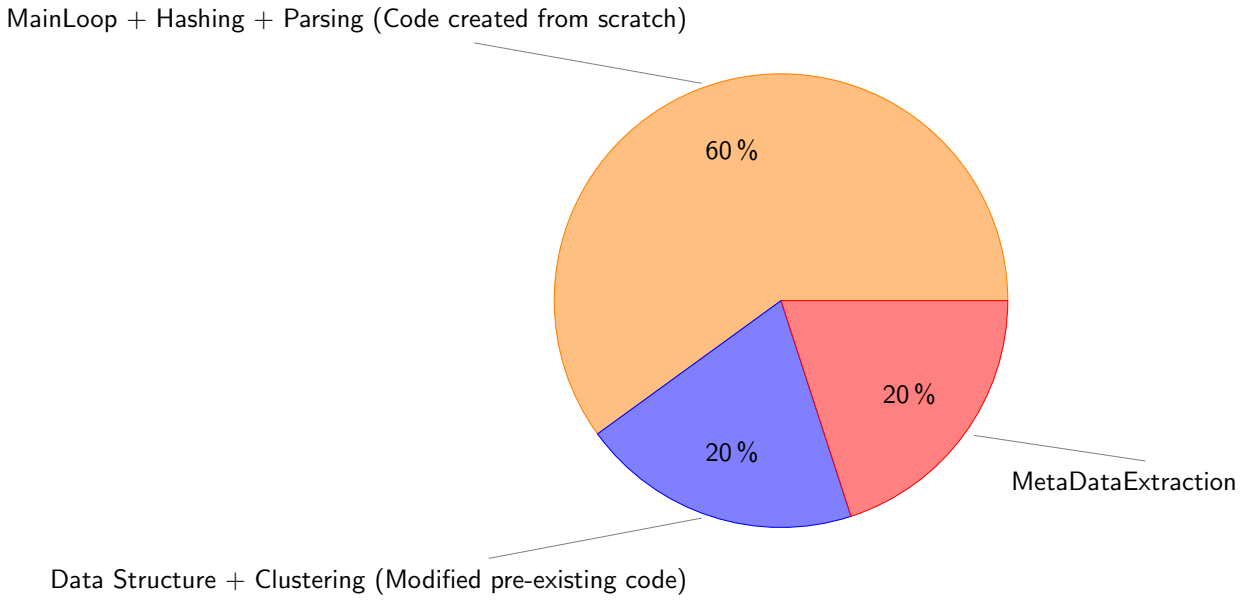


Figure 1: Shows the nature of the code for every distinct component. The slice size is a rough estimation.

3.2 When to use it

SchemaFuzz is a very usefull tool for anyone trying secure a piece of software that uses database ressources. The target software should be GDB compatible and the DBMS has to grant access to the target database through credentials passed as argument to this tool.
—It is very strongly advice to use a copy of the target databas erather than on the production material. Doing so will very likely result in the database being corrupted and not usable for any usefull mean.

4 Design

4.1 Generic explanation

SchemaFuzz implementation is based on some bits of the SchemaSpy project source code. The majority of this project is built on top of this already existing code and is organised as follows :

- mutation/data-set used as a way to store the imputs,outputs and other intresting data from the modification that was performed on the target database
- the mutation Tree, used to store the mutations coherently
- an analyser that scores the mutations to influence the paths that will be explored afterwards

This organisation will be detailled and discussed in the following sections.

4.2 SchemaSpy legacy/metadata extraction

SchemaSpy source code has provided the metadata extraction routine. The only job of this routine is to initialise the connection to the database and retrieve its metadata at the very beginning of the execution (before any actual SchemaFuzz code is run). These metadata include data types, table and table column names, views and foreign/primary key constraints. Having this pool of metadata under the shape of java objects allows the main program to properly frame what the possibilities are in terms of modifications as well as dealing with the possible constraints on the different tables. The modifications that will be created during the main loop will be designed to respect the frame built for the result of this extraction.
Exemple of typical metadata Set
In order to do that, the user shall provide this set of mandatory database related arguments

- The driver to the corresponding database RDBMS (only support PostGres at the moment)

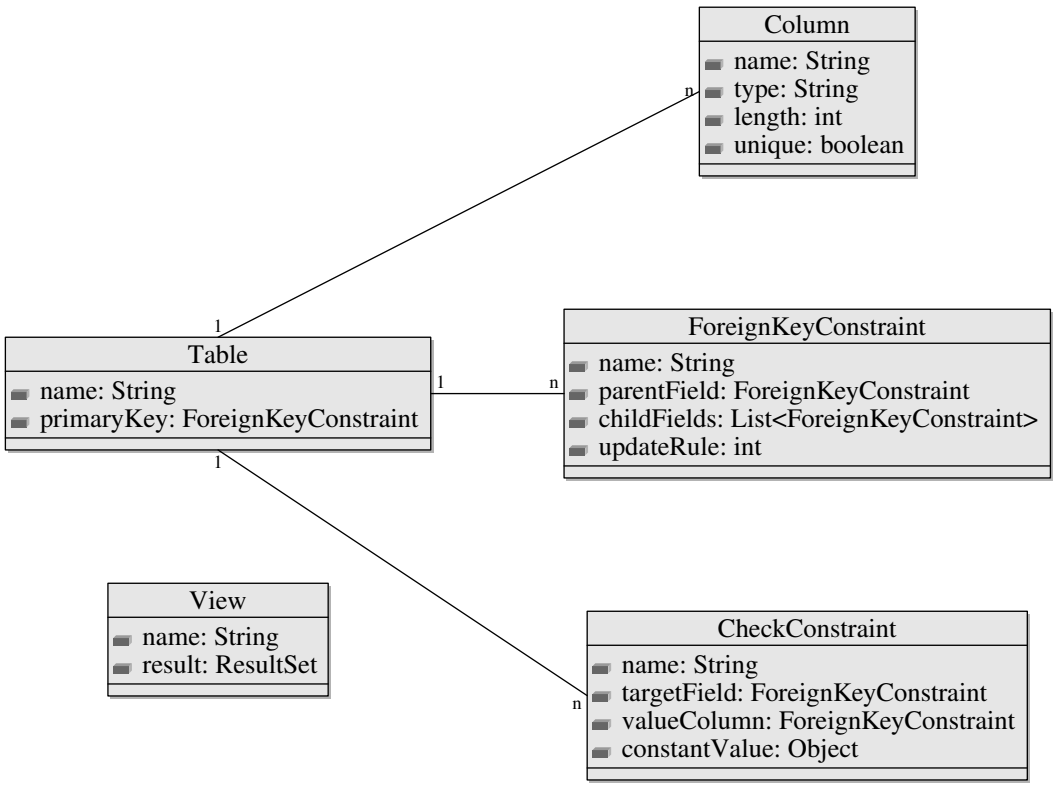


Figure 2: Objects returned by the metadata extraction routine.

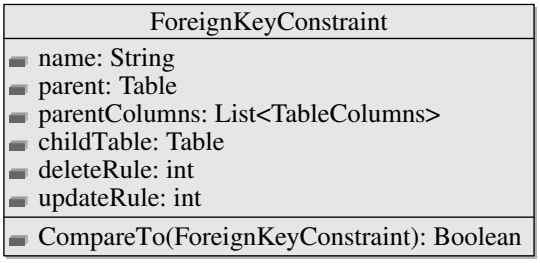


Figure 3: Objects returned by the metadata extraction routine.

- The credentials to be used to access the database.
- The name of the database (duh)

4.3 SchemaFuzz Core

4.3.1 Constrains

The target database often contains contraints on one or several tables. These constraints have to be taken into account in the process of fabricating mutations as most of the time they restrict the possible values that the pointed field can take. This restriction can take the shape of a Not Null constraint, Check constraint, Foreign key constraint (value has to exist in some other table's field) or Primary key constraint (no doublets of value allowed). These constraints are stored as Java objects instanciated from the corresponding class.
The last two ones are the problematic ones. They imply specific work before applying any mutations to make sure that the value respect all the restrictions. before doing anything else after the metadata extraction is done, SchemaFuzz performs an update of all the existing constraints on the database to add the CASCADE clause. This allows the values bonded by a foreign key constraints to take effect. This update reverts to take the constraints back to their initial state before the program exits.

Primary key contraints (PKC) : The primary key constraints require an extra DB query that checks the existence of the value in the column. If the value already exists (the query's result is not empty), the mutation will be dropped before being executed.

Foreign key contraints (FKC) : The foreignKey constraint is the trickiest one. Its inherent nature bonds two values of different table column where the value being referenced is called the father, and the referecing field, the child. To be precise, in order to change one of the two values, the other has to be changed accordingly in the same statement.SchemaFuzz uses the power of the CASCADE clause to make the change possible. This clause allows the

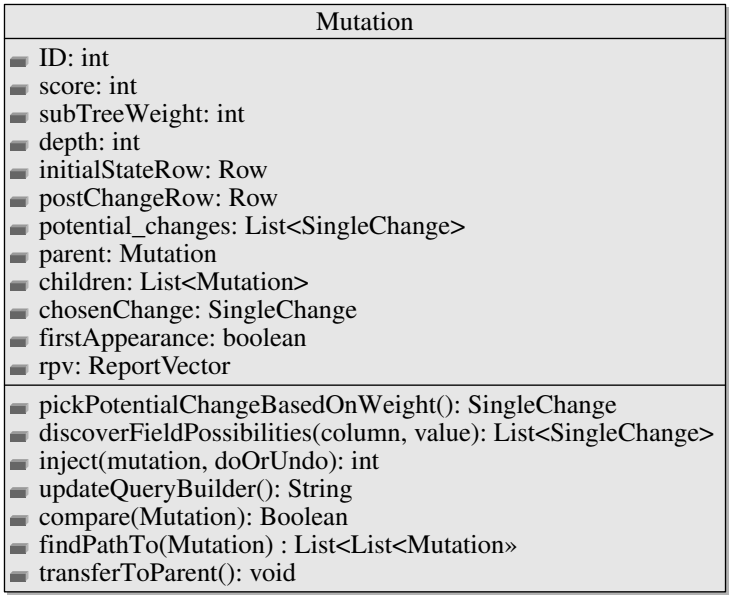


Figure 4: Structure of a Mutation

DRBMS to automaticly change the value of the child if the father has been changed. This mechanic allows to change any of the bounded values by changing the father's value. To do so, the software has a way to tranfert the mutation from a child to its parent (called the mutationTransfert).

4.3.2 Mutations

What is a Mutation A mutation is a Java object that bundles all the informations that are used to perform a modification in the database. Every is linked to its parent and inherits some of his parent's data. In the case of a follow up mutation the child inherits the the database row that was his parent's target.Therefore the initial state (state before the injection of the modification) of its target is exactly the final state (state after injection of the modification) of his parent's target. A mutation is created for each iteration of the main loop and represents a single step in the progress of the fuzzing. It also holds the informations concerning the result of the injection in the shape of a data vector. This data vector is then used to perform a clustering calculus to determine the "uniqueness" of the mutation. This value is also stored inside the mutation object and is used as the weight of this mutation in the tree.

Choosing patern For each iteration of the main loop, a modification has to be picked up as the next step in the fuzzing proccess. This is done by concidering the current state of the tree. Three parallel code paths can be triggered from this point.

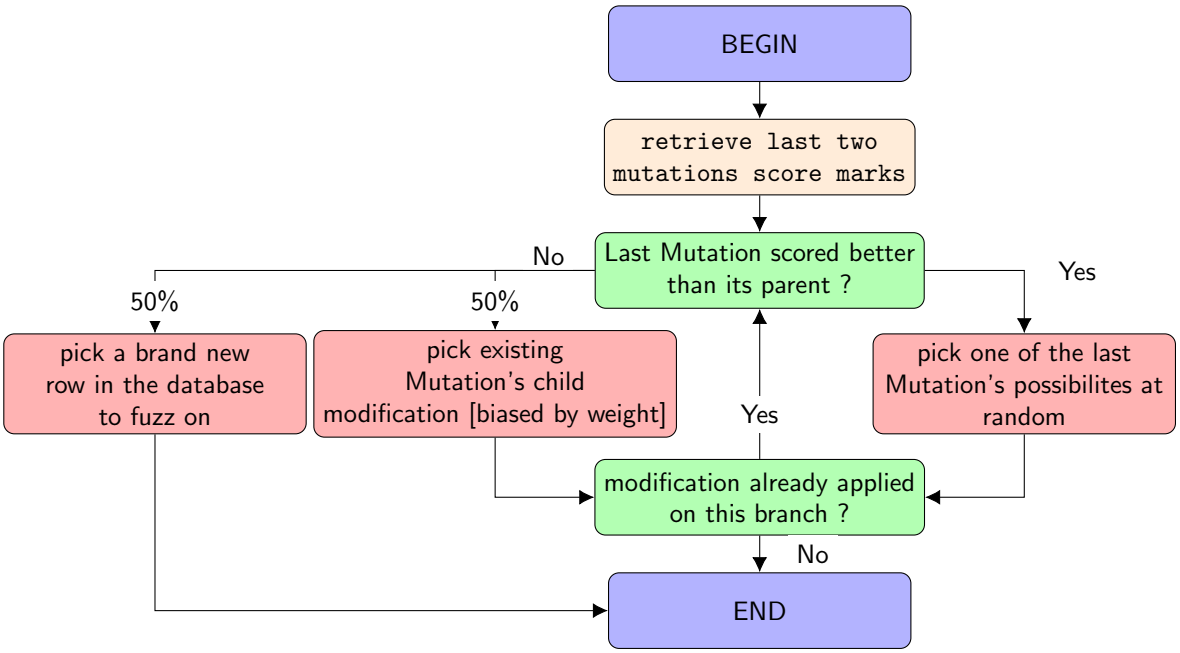


Figure 5: picking Patern schema

- Continue on the current branch of the tree (triggered if the last mutation scored better than its parent)
- Pick an existing branch in the tree and grow it (triggered if the last mutation scored worse than its parent on a 50/50 chance with the next bullet)
- Start a new branch (triggered if the last mutation scored worse than its parent on a 50/50 chance with the previous bullet)

A branch is a succession of mutation that share the same database row as their modification target. The heuristics determining the next mutation's modification are still very primitive and will be thinly ajusted in futures versions.

Creating malformed data As the goal of running this tool is to submit unexpected or invalid data to the target software it is necessary to understand what t Fuzzing a complex type such a timestamp variable has nothing to do with fuzzing a trivial boolean. In practice, A significant part o and this matter could absolutly be the subject of a more abstract work. We focused here on a very simple approach (as a first step). After retrieving the current row

being fuzzed (may it be a new row or a previously fuzzed row), the algorithm explores the different The algorithm then builds the possible modification for each of the fields for the current row. At the moment, the supported types are : More primitives types will be added in the future. The possible modifications that this tool can produce at the moment are : Int Types:

- Extreme values (0-32676 (int) etc...)
- Random value (0¡value¡32676 (int) etc...)
- Increment/Decrement the existing value (332 -¿ 333 OR 332 -¿ 331)

String Types:

- Change string to "aaa" ("Mount Everest" -¿ "aaa")
- Increment/Decrement ASCII character at a random position in the string ("Mount Everest" -¿ "Mount Fverest") Boolean
- Swaping the existing value (F -¿ T OR T -¿ F)

Date Types : (! IMPLEMENTED BUT NOT FULLY FUNCTIONNAL)

- Increment/Decrement date by 1 day/minutes depending on the precision of the date
- Set date to 00/00/0000

Obviously, these "abnormal" values might in fact be totally legit in some cases. in that case the analyzer will rank the mutation rather poorly, which will lead to this tree path not being very likely to be developped further more.

Sql handling All the SQL statements are generated within the code. This means that the data concerning the current and future state of the mutations have to be very precise. Otherwise, the SQL statement is very likely to fail. Sadly, since SchemaFuzz only supports postgreSQL, the implemented synthax follow the one of postgres DBMS. This is already a very big axis for future improvements and will be detailled in the dedicated section. The statement is built to target the row as precisely as possible, meaning that it uses all of the non fuzzed values from the row to avoid updating other row accidently. Only the types that can possibly be fuzzed will be used in the building of the SQL statement. Since this part of the code is very delicate in the sense that it highly depends on an arbitrary large pool of variables from various types it is a good bugg provider.

Injecting : The injection process sends the built statement to the DBMS so that the modification can be operated. After the execution of the query, depending of the output of the injection (one modification, several modifications, tranfer) informations are updated so that they can match the database state after the modification. If the modification failed, no trace of this mutation is kept, it is erased and running goes on like nothing happenned.

Special Case(MutationTransfert) : The mutation tranfert is a special case of a modification being applied to the database. It is triggered when the value that was supposed to be fuzzed is under the influence of a FKC as the child. In the case a FKC (In CASCADE mode), only the father can be changed, which also triggers the same modification on all of his children. The algorithm then "transfers" the modification from the original mutation to its father. After injecting the transfered mutation, the children mutation is indeed modified but the modification "splashed" on some parts of the database that was not meant to be changed. Hopefully, this does not impact the life of the algorithm until this mutation is reverted (see next paragraph).

Do/Undo routine : The Do/Undo mechanism is at the center of this software. Its behavior is crucial for the execution and will have a strong impact on the coherence of the data nested in the code or inside the target database throughout the runtime. This mechanism allows the algorithm to revert a previous mutation or, if necessary inject it one more time. Undoing a mutation applies the exact opposite modification that was originally applied to the database ending up in recovering the same database state as before the mutation was injected. Reverting mutations is the key to flawlessly shifting the current position in the mutation tree. The case of the transfered mutation is no exception to this. In this case, the mutation applied changes on an unknown number of fields in the database. But, the FKC still bounds all the children to their father at this point (this is always the case unless this software is not used as intended). Changing the father's field value back to its original state will splash the original values back on all the children. This mechanism might trigger failing mutations in some cases (usually mutations following a tranfer). This issue will be addressed in the known issues section.

4.3.3 Tree Based data structure

All the mutations that are injected at least once in the course of the execution of this software will be stored properly in a tree data structure. Having such a data structure makes parent-children relations between mutations possible. The tree follows the traditionnal definition of the a n-ary algorithmic tree. It is made of nodes (mutations) including a root (first mutation to be processed on a field selected randomly in the database) Each node has a number of children that depends on the ranking its mutation and the number of potential modifications that it can perform.

Weight : Weighting the nodes is an important part of the runtime. Each mutation has a weight that is equal to the analyzer's output. This value reflects the mutation's value. If it had an intresting impact on the target program behavior (if it triggered new buggs or uncommon code paths) than this value is high and vice-versa. The weight is then used as a mean of determining the upcomming modification. The chance that a mutation gets a child is

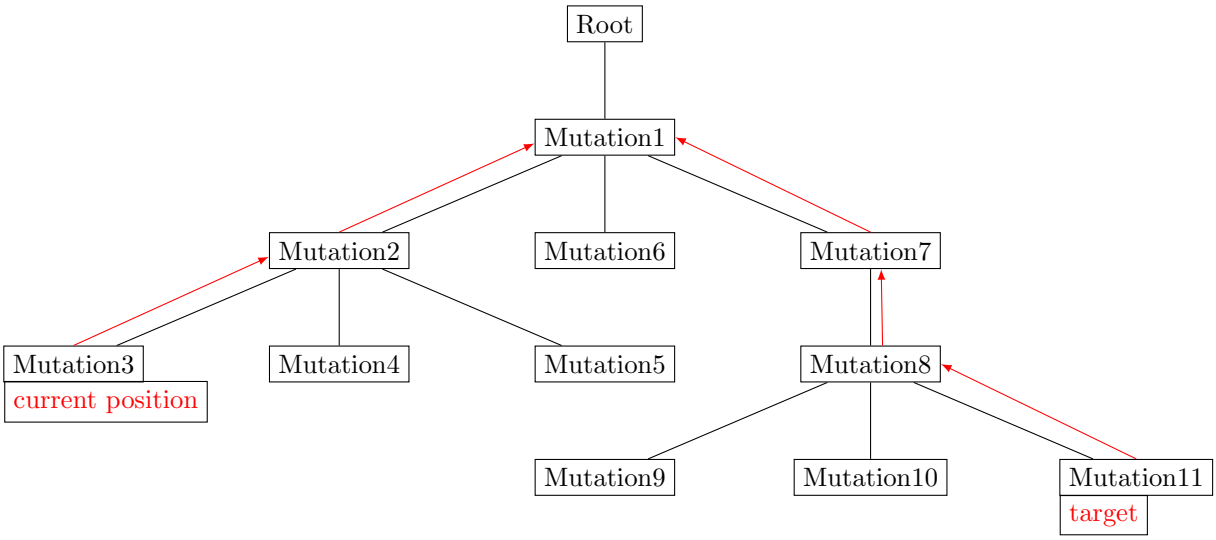


Figure 6: Objects returned by the metadata extraction routine.

directly proportionnal to its weight. This value currently isn't biased by any other parameter, but this might change in the future.

Path Since the weighting of the mutation allows to go back to previously more intresting mutations, there is a need for a path finder mechanism. Concretly, this routines resolves the nodes that separate nodes A and B in the tree. A and B might be children and parent but can also belong to complitely different branches. This path is then given to the do/undo routine that processes back the modifications to set the database up in the required state for the upcomming mutation.

4.3.4 The analyzer

Analyzing the output of the target programm is another critical part of Schema-Fuzz. The analyzer parses in the stack trace of the target software's execution to try measuring its interest. The main criteria that defines a mutation intrest is its proximity to previously parsed stack traces. The more distance between the new mutation and the old ones, the better the ranking.

Stack Trace Parser The stack trace parser is a separate Bash script that processes stack traces generated by the GDB C language debugger and stores all the relevent informations (function's name, line number, file name) into a Java object. The parser also generates as a secondary job a human readable

E	X	E	M	P	L	E	
✓	✓	✘	✓	✓	✓	✓	✘
E	X	A	M	P	L	E	S

Figure 7: Exemple of the levenshtein distance concept.

file for each mutation that synthetises the stack trace values as well as additionnal intresting information usefull for other mechanisms (that also require parsing). These additionnal informations include the path from root to mutation (usefull for rolling back the database to a specific state).

Hashing In order to be used in the clustering algorithm, the stack trace of a mutation has to be hashed. Hashing is usually defined as follows :

> "A hash value (or simply hash), also called a message digest, is a number generated from a string of text. The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is extremely unlikely that some other text will produce the same hash value."

In the present case, we used a different approach. Since proximity beetween two stack traces is the key to a relevant ranking, it is mandatory to have a hashing function that preserves the proximity of the two strings. In that regards, we implemented a version of the Levenshtein Distance algorithm. This algorithm can roughly be explain by the following :

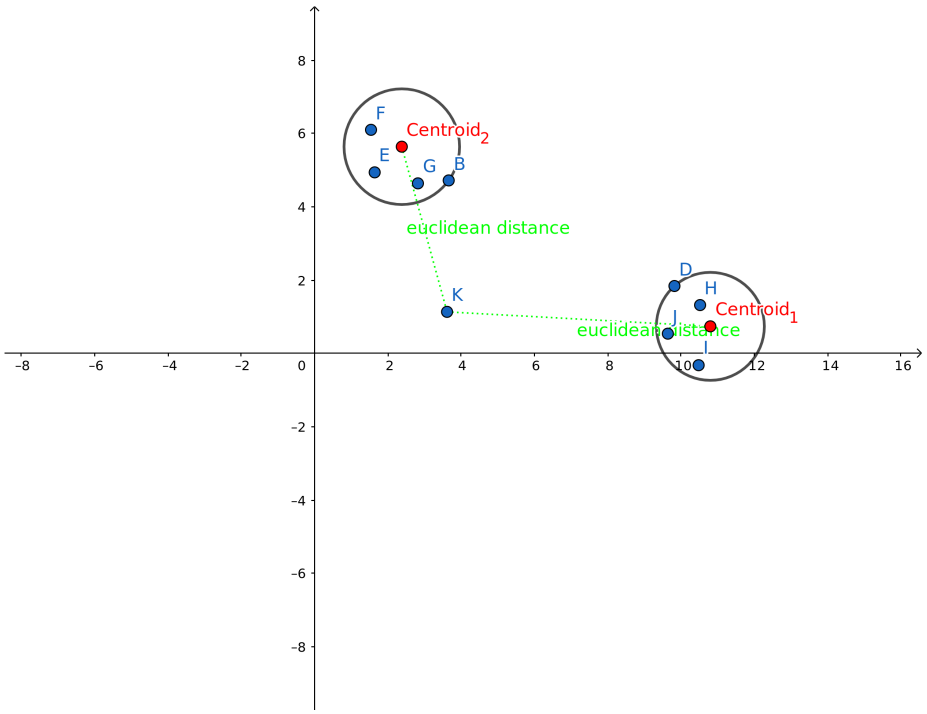
> "The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other."

After hashing the file name and the function name into numerical values trough Levenshtein distance, we are creating a triplet the fully (but not fully accuratly yet) represents the stack trace that is being parsed. This triplet will be used in the clustering method.
The Distance for this exemple is 2/8x100

The Scoring mechanism The "score" (or rank) of a mutation is a numerical value that reflects its intrest. This value is calculated through a modified version of a clustering method that computes an n-uplet into a integer depending on the sum of the euclidian distances from the n-uplet to the existing centroids (groups of mutation's n-uplets that were already processed). This value is then set as the mutation's rank and used as a mean of chosing the upcomming mutation.

4.4 Known issues

About one mutation out of 15 will fail for unvalid reaseons.



4.4.1 Context Cohorence

A significant amount of the failing mutations do so because of the transfer mechanism. As said in the dedicated section, this mechanism applies more than one change to the database (Potentially the whole database). In specific case, this property can become problematic. More specificaly, when the main loop identifies a mutation's child as the upcomming mutation and its parent row has been splashed with the effect of a transfer. In this case, the data embedded in the schemaFuzz data structure may not match the data that are actually in the database, this delta will likely induce a wrongly designed SQL statement that will result in a SQL failure (meaning that 0 row were updated by the statement).

4.4.2 Foreign Key constraints

For a reason that is not yet clear, some of the implied FKC of the target database can't be properly set to CASCADE mode. This result in a FKC error (mutation fails but the program can carry on)

4.4.3 Tests

Besides the test suit written by the SchemaSpy team for their own tool (still implemented in SchemaFuzz for the meta data extraction), the tests for this

project are very poor. Their are only very few of them and their utility is debatable. This is due to the lack of experience in that regard of the main developper. Obviously, we are very well aware of this being a really severe flaw in this project and will be one of the main future improvements. This big lack of good maintenance equipment might also explain some of the silent and non silent buggs that still remain in the code to this day.

4.4.4 Code Quality

We are well aware that this tool's source code is of debatable quality. This fact induces the buggs and unexpected behaviors discussed earlier on some components of this program. The following points constitute the main flaw of the source code:

- Hard to maintain. The code is not optimised either in term of size or efficency. Bad coding habits tend to make it rather weak and unstable to context changes.
- Structure is not intuitive. The main loop of the program lacks a good structure.

5 Results and exemples

In the proccess of being written.

6 Upcomming features and changes

This section will provide more insights on the future features that might/may/will be implemented as well as the changes in the existing code. Any sugestion will be greatly appriciated as long as it is relevent and well argumented. All the relevent information regarding the contributions are detailled in the so called section.

6.1 General Report

In its future state, SchemaFuzz will generate a synthesized report concerning the overall execution of the tool (which it does not do right now). This general report will primarely contain the most "intresting" mutations (meaning the mutations with the highest score mark) for the whole run. A more advanced version of this report would also take into account the code coverage rate for each mutation and execute a last clustering round at the end of the execution to generate a "global" score that would represent the global value of each mutations.

6.2 Code coverage

We are considering changing or simply adding code coverage in the clustering method as a parameters.Not only would this increase the accuracy of the scoring but also increase the accuracy of the "type" of each mutation. To this day, this tool does not make a concrete difference in terms of scoring or information generating (reports) beetween a mutation with a new stack trace in a very common code path and a very common stack trace in a very rarely triggered code path.

6.3 Data type Pre-analyzing

This idea for this feature to be is to implement some kind of "auto learning" mechanism. To be more precise, this routine is meant to performed a statistical analysis on a representative portion database's content. This analysis would provide the rest of the program the commun values encountered for each field. More genericly, this would allow the software to have a global view over the format of the data that the database holds. Such global understanding of the content format is very intresting to make the modifications possibilites nore relevent. Indeed, one of the major limitation of SchemaFuzz is its "blindness". That is to say that some of the modifications performed in the course the execution of the program are irrelevent due to the lack of information on what is supposed to be stored in this precise field. For instance, a field that only holds numerical values that go from 1 to 1000 even if it has enough bits to encode from -32767 to 32767 would have a very low chance of triggering a crash if this software modifies its value from 10 to 55. on the other end, if the software modifies this very same field from 10 to -12000, then a crash is much more likely to pop up. Same principle applies to strings. Suppose a field can encode 10 characters. the pre analysis, detected that, for this field, most of the value were surnames beginning with the letter "a". Changing this field from "Sylvain" to "Sylvaim" will probably not be very effective. However, changing this same field from "Sylvain" to "NULL" might indeed triggered an unexpected behavior.
This pre-analysis routine would only be executed once at the start of the execution, right after the meta data exctraction. The result of this analysis will be held by a specific object. this object's lifespan is equal to the duration of the main loop's execution (so that every mutation can benefits from the analysis data.)

6.4 Centralised anonymous user data

SchemaFuzz's efficiency in thightly linked to the quality of its heuristics. this term includes the following points

- Quality of the possible modifications for a single field
- Quality of the possible modifications for each data type

- Quantity of possible modifications for a single field
- Quantity of supported data types

Knowing this, we are also concidering for futures enhancements an anonymous data collection for each execution of this tool that will be statisticly computed to determine the best modification in average. This would improve the choosing mechanism by balancing the weights depending on the modifcation's average quality. Modifications with higher average quality would see their weight increased (meaning they would get picked more frequently) and vice versa.

1 Internship organisation

1.1 Introduction

This section is meant to be added to the University version of this documentation. It will be written as Erwan Ulrich and will focus on the different aspects of the organisation of the project. The folllowing text will also be written with a more personnal and more critical point of view as a mean of self analysement.

1.2 Calendars

The SchemaFuzz project has had since its genesis a quiet clear view of how the development should evolve. The desired features have been discussed and the big picture had been designed to fit the time that the main developer had for his work at this position. The project had to pass trough different phases of development that are detailed in the following timeline diagram.

Some of the tasks of the above timeline were completed on time, some others were delivered late, and some were delayed in the timeline because of the previous point. In the end, the project was lead in a way that is best described by the following timeline diagram.

Those two diagrams differ on some points. This is one of the major failures for the development of this project throughout the course of these 6 months. There are several reasons that explain why this project could have been lead in a better way. they will be detailled and discussed in the next section.

1.3 Organisationnal failures

This section has a particular value in this report, it is on the first hand a description of why the SchemaFuzz did not meet all of its defined goals. Other the other hand, it is a personnal reminder of what should be improved in my work habbits and general organisation when leading a project of such a large size.

- Defining tasks/features as daily/weekly sub goals
- Improving multitasking
- Setting up more fluid communication

1.4 Positive outcomes

7 Contributing

You can send your ideas at
`erwan.ulrich@gmail.com` Or directly create a pull request on the official repository to edit this document and/or the code itself

8 Conclusion