Shadow Symbolic Execution with Java PathFinder

Yannic Noller
Humboldt University of Berlin
yannic.noller@informatik.hu-berlin.de

Hoang Lam Nguyen
Humboldt University of Berlin
nguyenhx@informatik.hu-berlin.de

Minxing Tang
Humboldt University of Berlin
tangminx@informatik.hu-berlin.de

Timo Kehrer
Humboldt University of Berlin
timo.kehrer@informatik.hu-berlin.de

ABSTRACT

Regression testing ensures that a software system when it evolves still performs correctly and that the changes introduce no unintended side-effects. However, the creation of regression test cases that show divergent behavior needs a lot of effort. A solution is the idea of *shadow symbolic execution*, originally implemented based on KLEE for programs written in C, which takes a unified version of the old and the new program and performs symbolic execution guided by concrete values to explore the changed behavior. In this work, we apply the idea of shadow symbolic execution to Java programs and, hence, provide an extension of the Java PathFinder (JPF) project to perform shadow symbolic execution on Java bytecode. The extension has been applied on several subjects from the JPF test classes where it successfully generated test inputs that expose divergences relevant for regression testing.

Keywords

Java PathFinder; Symbolic PathFinder; Symbolic Execution; Regression Testcase Generation; Software Engineering

1. INTRODUCTION

One of the distinctive properties of real-world software is that it *evolves*, since it has to be adapted to its continuously changing environment. Software changes, usually referred to as *patches*, typically fix incorrect behavior or introduce new functionality. However, it is also known that these patches are prone to introduce new errors [3, 10], which is why users are often hesitant to update to the latest version.

To prevent this problem, *regression testing* is performed on the modified program version in order to provide confidence that the newly introduced software changes behave as expected and have no unintended side-effects. Since this is an expensive process, it is important to select the appropriate test cases. For instance, several regression testing techniques [4, 2] select and run a subset of the test cases from the program's existing test suite or automatically generate test cases with high coverage of the changed code [6]. However, even if the selected test cases achieve full statement or full branch coverage of the patch code, they do not necessarily exercise all new behaviors introduced by the patch.

To give an illustration, consider a patch that only changes the conditional statement `if(x > 5)` to `if(x > 10)`. The two test cases `x=0` and `x=15` cover both sides of the branch, but the execution of these inputs is completely unaffected by the patch since they result in the same branching behavior in both program versions. On the other hand, if `x` is between 6 and 10 (inclusive), the two program versions exhibit divergent behavior as they take different sides of the branch.

Recently, Palikareva et al. [7] have introduced a dynamic symbolic execution-based technique, which they refer to as *shadow symbolic execution*. Their technique is designed to generate test inputs that cover new program behaviors introduced by a patch. Shadow symbolic execution works by executing both the old (buggy) and new (patched) version in the same symbolic execution instance, with the old version *shadowing* the new one. Therefore, it is necessary to manually *merge* both programs into a change-annotated, unified version. Based on such a unified version, the technique detects divergences along the execution path of an input that exercises the patch. Their tool SHADOW, which we refer to as $Shadow_{KLEE}$, is implemented on top of the KLEE symbolic execution engine [1].

Our novel implementation $Shadow_{JPF}$, as an extension of the Java PathFinder (JPF) [9], applies the idea of shadow symbolic execution to Java bytecode and, hence, allows to detect divergences in Java programs that expose new program behavior. The application of our extension on various subjects from the JPF test classes evaluate its test case generation capabilities.

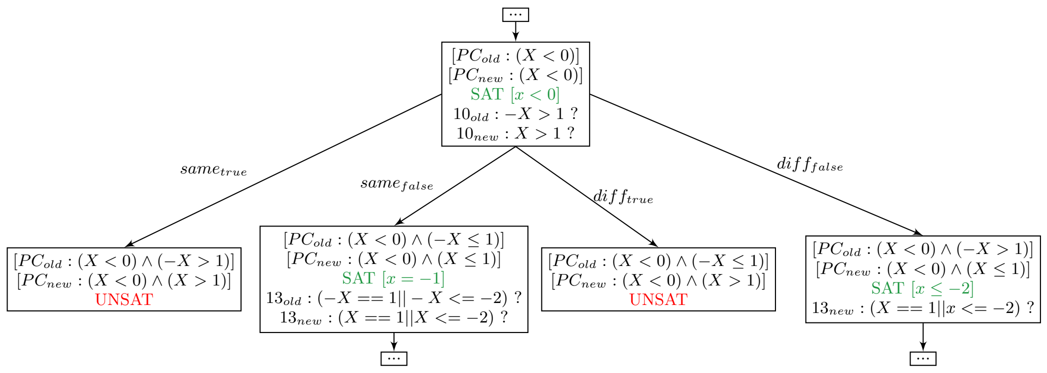
2. SHADOW SYMBOLIC EXECUTION

Shadow symbolic execution [7] aims at generating test inputs that cover the new program behaviors introduced by a patch. Their approach takes as input the buggy and the patched version (say *old* and *new*, respectively) and assumes an existing test suite.

```
 1  int foo(int x){
 2    int y;
 3    if(x < 0){
 4       y = -x;
 5    }
 6    else{
 7       y = 2 * x;
 8    }
 9+   y = -y;
10    if(y > 1){
11        return 0;
12    } else {
13        if(y == 1 || y <= -2){
14            assert(false);
15        }
16    }
17    return 1;
18  }
```

Listing 1: Toy example to show the approach of shadow symbolic execution.

To give an illustration, consider the patch for the method `foo()` in Listing 1. There is an additional assignment in line 9 for the variable y that negates it to $-y$. This patch fixes the assertion error (line 14) for $x = -1$, but it introduces a new assertion error for, e.g., $x = -2$. Since the approach aims at generating test cases for the different execution paths of the buggy and the



$\cdots$
$[PC_{old} : (X < 0)]$
$[PC_{new} : (X < 0)]$
SAT $[x < 0]$
$10_{old} : -X > 1$?
$10_{new} : X > 1$?
$same_{true}$
$same_{false}$
$diff_{true}$
$diff_{false}$
$[PC_{old} : (X < 0) \wedge (-X > 1)]$
$[PC_{new} : (X < 0) \wedge (X > 1)]$
UNSAT
$[PC_{old} : (X < 0) \wedge (-X \leq 1)]$
$[PC_{new} : (X < 0) \wedge (X \leq 1)]$
SAT $[x = -1]$
$13_{old} : (-X == 1 || -X <= -2)$?
$13_{new} : (X == 1 || X <= -2)$?
$[PC_{old} : (X < 0) \wedge (-X \leq 1)]$
$[PC_{new} : (X < 0) \wedge (X > 1)]$
UNSAT
$[PC_{old} : (X < 0) \wedge (-X > 1)]$
$[PC_{new} : (X < 0) \wedge (X \leq 1)]$
SAT $[x \leq -2]$
$13_{new} : (X == 1 || x <= -2)$?
$\cdots$
$\cdots$

Figure 1: Partial four-way forking symbolic execution tree for the combined execution of the old and the new version of the program in Listing 1 for the test input $x = -1$. Each node represents a state in the symbolic search space, where each state holds the combined information of the old and the new symbolic execution.

patched version, the optimal result would be two test inputs: (i) one for the fixed path, and (ii) one for the path with the newly introduced assertion error.

In order to execute both program versions in a single symbolic execution instance, Palikareva et al. [7] follow an approach that unifies both program versions with `change()` annotations. The annotations resemble a function call with two arguments, where the first argument represents the code expression from the old version and the second argument the expression from the new version. The unifying process is performed manually. In our example, the change of line 9 is annotated as `y=change(y,-y)`.

Afterwards, the dynamic symbolic execution is performed in two steps: (i) the concolic phase and (ii) the bounded symbolic execution (BSE) phase. The concolic phase is initialized with the test cases that touch at least one patch statement. It collects the divergence points that are later used as starting points for the BSE phase. All types of instructions are basically handled in the same way as in traditional symbolic execution, except for conditional statements, for which shadow symbolic execution forks execution into four different paths, where each path is specified by the side of the branch taken by the different program versions (cf. Figure 1). A *same* path is the execution path where the new and the old program version take the same side of the conditional statement, i.e., if both versions follow the *true* branch (aka $same_{true}$ path) or vice versa both versions follow the $false$ branch (aka $same_{false}$ path). A *diff* path is an execution path where the new program version takes an execution path which is different from that of the old version, i.e., if the new version follows the *true* branch of a conditional statement while the old version follows the $false$ branch (aka $diff_{true}$ path) or vice versa (aka $diff_{false}$ path). If the concrete executions of the two versions diverge at a conditional statement, the concolic phase will be stopped and the divergence point for the $diff_x$ path will be added to the BSE phase. If both program versions behave identically for the concrete input, shadow symbolic execution follows the $same_x$ path. However, if divergences are possible, for each feasible *diff* path an input that exercises the divergent behavior will be generated and added to the BSE phase.

This strategy includes that for adding and removing straightline code, encapsulated by the change annotations `if(change(false, true))` and `if(change(true,false))`, shadow symbolic execution directly triggers a divergence point. This is a conservative approach, hence, an over-approximation of the *diff* paths, since the added/deleted code may not propagate a change at a branching point and thus not lead to a *diff* path. As long as the concolic executions do not diverge, shadow symbolic execution continues executing both program versions until the end of the program in order to explore any additional divergences along the way. The BSE phase runs only on the new version starting from the collected conditional statements, i.e., symbolic execution on a fixed resource budget with a breadth-first exploration of the execution tree.

At the end, the generated test inputs are used to execute both versions and the results are manually compared to classify them as expected divergences (such as an intended bug fix) or as regression bugs. A solution is the idea of shadow symbolic execution, originally implemented based on KLEE for programs written in C.

As an illustration, Figure 1 shows a partial four-way forking symbolic tree for the combined execution of the *old* and the *new* version of the program in Listing 1. Depending on the concrete input, shadow symbolic execution only explores a subset of the possible symbolic states. This has the benefit to limit the search space. Suppose that a developer has written the test case $x = -1$, as this input caused an assertion error in the old program version. Note that this input also fully covers the changed statements. At the first conditional statement in line 3, shadow symbolic execution simply follows the concrete execution since there was no change yet and, hence, the *diff* paths are unsatisfiable at this point. After executing line 9, the variable y is mapped to the symbolic expression $-X$ in the old program version and to X in the new version with the concrete values now being 1 and -1, respectively. As a result, both concrete executions take the $false$ branch at the conditional statement in line 10, i.e. they follow the $same_{false}$ path. Shadow symbolic execution identifies additionally, that the $diff_{false}$ path is satisfiable (cf. Figure 1) and stores the divergence point for the second execution phase. Continuing with the concrete executions this leads to the path that is only followed by the input $x = -1$. This execution will be stopped at line 13, where the concrete executions diverge: the old version takes the *true* branch that leads to an assertion error and the new version takes the $false$ branch that returns 1 and represents the bug fix. Shadow symbolic execution will report a *diff* path for $x = -1$, which can be classified as an expected change. Eventually, bounded symbolic execution will be started from the stored divergence point from line 10. With the given path condition only one path is left as feasible, which is $x \leq -2$. This *diff* path leads to an assertion error and can be classified as a regression bug.

3. IMPLEMENTATION

$Shadow_{JPF}$ is implemented as an extension of the symbolic execution project of JPF, namely Symbolic PathFinder (SPF) [8], and leverages its symbolic execution functionality in order to enable shadow symbolic execution of Java bytecode. Similar to SPF, the tool makes use of various extension mechanisms of JPF, such as attribute objects, choice generators and listeners. In fact, $Shadow_{JPF}$ overrides the core extensions of SPF in order to specifically support shadow symbolic execution. The tool is available at: `https://github.com/hub-se/jpf-shadow`

3.1 Efficiently sharing Symbolic States using Diff-Expressions

Since shadow symbolic execution runs both program versions (as a single unified program) in the same symbolic execution instance, it is important to maximize sharing between the symbolic states in order to keep memory consumption low. Similar to the approach of Palikareva et al. [7], instead of maintaining two separate symbolic stores, $Shadow_{JPF}$ constructs a `DiffExpression` whenever the tool encounters a `change()` annotation. A `DiffExpression` basically stores the symbolic and shadow expression of a variable and is associated with it as a data attribute object in the same way as a regular symbolic expression in SPF. Note that storing a `DiffExpression` is only necessary if the symbolic expression of a variable diverges between the two program versions. As long as the symbolic expression of a variable is equal in both program versions, storing a single symbolic expression object (as provided by SPF) is sufficient. Algorithm 1 illustrates how a `DiffExpression` is constructed whenever a `change()` method is invoked.

Algorithm 1: Execute change(old,new) method

```
1  attr_old ← attribute object of old;
2  attr_new ← attribute object of new;
3  if attr_old instanceof DiffExpression then
4      result_shadow ← attr_old.getShadow();
5  else
6      result_shadow ← attr_old
7  if attr_new instanceof DiffExpression then
8      result_symbc ← attr_new.getSymbc();
9  else
10     result_symbc ← attr_new
11 return new DiffExpression(result_shadow,result_symbc)
```

3.2 Extended Bytecode Implementation

Arithmetic bytecode. Since a symbolic variable can be associated to either a symbolic expression or a `DiffExpression`, the arithmetic as well as the branching bytecode has to be extended in order to support both types of expressions. As an example, consider Algorithm 2 that describes how shadow symbolic execution of the `IADD` instruction is performed. The highlighted lines show the differences between the implementation of SPF and $Shadow_{JPF}$. Similar to the implementation in SPF, it is first checked whether both operands are concrete, in which case the execution is delegated to the concrete super class. Otherwise, if at least one operand is symbolic, the result also becomes symbolic. The key idea is to determine the shadow and symbolic expression for both operands (line 9 to 14) and simply add the respective expressions to obtain the symbolic and shadow expression of the result (line 15 and 16). Note that the shadow and symbolic expression of a variable are equal if they have not diverged yet (line 13 and 14). For this reason, only if at least one of the operand attributes is a `DiffExpression`, the resulting attribute object also becomes a `DiffExpression` (line 17 to 20).

Choice	Path	PC
1	$same_{true}$	$pc \wedge (sym\_v1 = sym\_v2) \wedge (shadow\_v1 = shadow\_v2)$
2	$same_{false}$	$pc \wedge (sym\_v1 \neq sym\_v2) \wedge (shadow\_v1 \neq shadow\_v2)$
3	$diff_{true}$	$pc \wedge (sym\_v1 = sym\_v2) \wedge (shadow\_v1 \neq shadow\_v2)$
4	$diff_{false}$	$pc \wedge (sym\_v1 \neq sym\_v2) \wedge (shadow\_v1 = shadow\_v2)$
5	concrete	depends on the concrete input

Table 1: The five possible choices for each execution path and the corresponding path conditions for the `IF_ICMPEQ` instruction. pc denotes the current path condition while sym_v1/v2 and shadow_v1/v2 represent the symbolic and shadow expressions of the operands to be compared, respectively.

Branching bytecode. In SPF, the symbolic execution of a conditional statement involves setting a `ChoiceGenerator` with two choices, which represent the *true* and *false* sides of the branch. Each choice is associated with the respective path condition, which is checked for satisfiability by a constraint solver. Recall that shadow symbolic execution forks execution into four different paths, where each path is specified by the side of the branch taken by the two program versions (cf. Figure 1). Therefore, it is necessary to create four choices, one for each possible path. Figure 1 gives an overview of the choices and the resulting path conditions for the `IF_ICMPEQ` instruction that compares to variables for equality. Originally, shadow symbolic execution operates in two phases: (i) the concolic phase and (ii) the bounded symbolic execution (BSE) phase. Instead of running a concolic phase, we added a fifth choice that determines the next execution path based on the concrete inputs. To give an illustration, consider the modified branching statement `if(change(x>1, x<=5))` with the concrete input $x = 3$. In order to determine the outcome in both program versions, we check the satisfiability of the constraints $(x > 1 \wedge x = 3)$ and $(x <= 5 \wedge x = 3)$ for the old and new program version, respectively. Since both constraints are satisfiable, both versions take the *true* path. Note that in this case the concrete execution replaced the exploration of the $same_{true}$ path (choice 1). As a result, if we only consider the choices 5, 4 and 3 (cf. Figure 1) at each conditional statement, $Shadow_{JPF}$ can follow the concrete execution of both program versions until they diverge, while checking for possible *diff* paths along the concrete execution path. As soon as a *diff* path is explored, only the choices 1 and 2 are considered (while ignoring the shadow expressions), effectively replacing the bounded symbolic execution phase.

4. EVALUATION

We evaluated our implementation w.r.t (i) its correctness and (ii) its effectiveness of generating regression tests for Java programs. Therefore, we answer the following research questions:

RQ.1: Is our implementation consistent with the original implementation that was implemented for C programs?
RQ.2: Compared to pure SPF, can $Shadow_{JPF}$ generate more test cases that are relevant for regression testing?

4.1 Experimental Setup

For the evaluation we used publicly available artifacts that JPF/SPF can handle and made them usable for regression testing by generating multiple versions of them with the MAJOR mutation framework [5]. We used the full generation setup as provided by the authors, without the operators that cannot be handled by SPF or which produced errors in our SPF experiments. As a first evaluation step, we selected the following software artifacts as our experimental subjects from the official SPF repository[1]: BankAccount.deposit(), BankAccount.withdraw(), BankAccount.main()

[1]https://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc

Algorithm 2: Shadow symbolic execution of `IADD`

```
op_v1 ← attribute object of first operand ;
op_v2 ← attribute object of second operand ;
if operands concrete then
    return super.execute() ;
else
    stack.pop();
    stack.pop();
    stack.push(0);
    if op_v$_i$ ($i \in 1, 2$) instanceof DiffExpression then
        sym_v$_i$ ← op_v$_i$.getSymbc() ;
        shadow_v$_i$ ← op_v$_i$.getShadow() ;
    else
        sym_v$_i$ ← op_v$_i$;
        shadow_v$_i$ ← op_v$_i$;
    sym_r ← sym_v1 + sym_v2;
    shadow_r ← shadow_v1 + shadow_v2;
    if op_v1 or op_v2 instanceof DiffExpression then
        result ← new DiffExpression(shadow_r,sym_r);
    else
        result ← sym_r;
    setAttributeObject(result);
    return nextInstruction();
```

and generated in total 51 mutants. For all mutants we manually added the change annotations to generate the unified version. Since only the executable binaries of the original implementation Shadow are available, but not the actual source code, we decided to manually transform our Java subjects into C programs, so that we can check the consistency between our results and the results by Shadow.

The experiments were conducted on a machine running macOS 10.12.6 featuring an 2.9GHz Intel Core i5 and 16 GB of memory. We configured the symbolic executions with an unbounded depth limit and a timeout of one hour.

4.2 Results and Analysis

Table 2 shows the detailed results of the performed experiments. The first column names the corresponding class and method that were tested, together with an id which specifies each mutant. Column *type* contains the mutation operation adapted by [5]: Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR) and Statement Deletion (STD). Since Major only generated single mutants per class, we also combined them manually to get multiple changes per class. In such cases the numbers at the end of each subject name denote the combined mutants (we named the type "MUL" for multiple changes). The following columns describe the execution time in seconds, the number of visited states during the symbolic exploration, the maximum used memory in MB and the number of resulting path conditions (the number in the brackets for SPF represent the number of paths that were *diff* paths) for the normal symbolic execution (SPF), for the shadow symbolic execution with our SPF extension $Shadow_{JPF}$ (SWPF) and for the original tool $Shadow_{KLEE}$ (SW). The first row of the table shows the detailed execution results for the method `foo()` from Listing 1.

Subject	**Type**	**Time [s]**		**# States**		**Memory [MB]**		**# Paths (diff)**		
		SPF	$SWPF$	SPF	$SWPF$	SPF	$SWPF$	SPF	$SWPF$	SW
Foo	-	2	2	16	17	434	690	4 (2)	2	2
BankAccount.deposit_1	ROR	< 1	< 1	4	19	245	245	2 (0)	1	1
BankAccount.deposit_2	ROR	< 1	< 1	4	16	245	245	2 (0)	1	1
BankAccount.deposit_3	ROR	< 1	< 1	2	10	245	245	1 (0)	1	1
BankAccount.deposit_4	STD	< 1	< 1	4	7	245	245	2 (0)	1	1
BankAccount.deposit_5	AOR	< 1	< 1	4	5	245	245	2 (0)	0	0
BankAccount.deposit_6	AOR	< 1	< 1	4	5	245	245	2 (0)	0	0
BankAccount.deposit_7	AOR	< 1	< 1	8	9	245	245	3 (1)	1	1
BankAccount.deposit_8	STD	< 1	< 1	4	5	245	245	2 (0)	0	0
BankAccount.withdraw_1	ROR	1	1	6	19	245	245	3 (0)	1	1
BankAccount.withdraw_2	ROR	<1	<1	6	19	245	245	3 (0)	1	1
BankAccount.withdraw_3	ROR	<1	<1	4	9	245	245	2 (0)	2	2
BankAccount.withdraw_4	STD	<1	<1	8	9	245	245	4 (2)	2	2
BankAccount.withdraw_5	ROR	<1	<1	6	22	245	245	3 (0)	1	1
BankAccount.withdraw_6	ROR	<1	<1	6	22	245	245	3 (0)	1	1
BankAccount.withdraw_7	ROR	<1	<1	4	10	245	245	2 (0)	1	1
BankAccount.withdraw_8	STD	<1	<1	6	8	245	245	3 (0)	0	0
BankAccount.withdraw_9	AOR	<1	<1	6	8	245	245	3 (0)	0	0
BankAccount.withdraw_10	AOR	<1	<1	6	8	245	245	3 (0)	0	0
BankAccount.withdraw_11	AOR	<1	<1	8	12	245	245	4 (1)	1	0
BankAccount.withdraw_12	STD	<1	<1	6	8	245	245	3 (0)	0	0
BankAccount.main_1	ROR	2	< 1	24	52	433	245	4 (0)	1	1
BankAccount.main_2	ROR	2	< 1	24	52	690	245	4 (0)	1	1
BankAccount.main_3	ROR	2	< 1	16	26	434	245	3 (0)	1	1
BankAccount.main_4	STD	3	< 1	24	17	690	245	4 (1)	1	1
BankAccount.main_5	AOR	3	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_6	AOR	3	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_7	AOR	3	< 1	32	22	690	245	5 (0)	0	0
BankAccount.main_8	STD	3	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_9	ROR	2	< 1	24	58	434	245	4 (0)	1	1
BankAccount.main_10	ROR	3	< 1	24	58	690	245	4 (0)	1	1
BankAccount.main_11	ROR	1	< 1	16	26	433	245	3 (0)	1	1
BankAccount.main_12	STD	3	< 1	24	17	690	245	4 (1)	1	1
BankAccount.main_13	ROR	3	< 1	24	20	690	245	4 (0)	0	0
BankAccount.main_14	ROR	3	< 1	24	20	690	245	4 (0)	0	0
BankAccount.main_15	ROR	3	< 1	20	11	434	245	4 (1)	1	1
BankAccount.main_16	STD	3	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_17	AOR	2	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_18	AOR	2	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_19	AOR	3	< 1	28	22	690	245	5 (0)	1	0
BankAccount.main_20	STD	2	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_21	STD	3	< 1	24	17	690	245	4 (0)	0	0
BankAccount.main_22	ROR	1	< 1	8	15	309	245	2 (2)	2	2
BankAccount.main_23	ROR	1	< 1	8	15	309	245	2 (2)	2	2
BankAccount.main_1_13	MUL	2	< 1	24	52	433	245	4 (0)	1	1
BankAccount.main_2_22	MUL	1	< 1	8	41	309	245	2 (0)	3	3
BankAccount.main_15_23	MUL	1	< 1	6	13	245	245	2(2)	2	2
BankAccount.main_5_18	MUL	2	< 1	24	14	690	245	4 (0)	0	0
BankAccount.main_3_23	MUL	1	1	8	15	309	245	2 (2)	2	2
BankAccount.main_17_22	MUL	1	1	8	15	309	245	2 (2)	2	2
BankAccount.main_3_10_22	MUL	< 1	1	2	5	245	245	1 (0)	1	1
BankAccount.main_5_18_23	MUL	1	< 1	8	15	309	245	2 (2)	2	2

Table 2: Experimental results for the comparison of Symbolic PathFinder (SPF), $Shadow_{JPF}$ (SWPF) and the original tool $Shadow_{KLEE}$ (SW).

RQ.1 Consistence with original $Shadow_{KLEE}$

In order to answer RQ.1 we compared the number of test cases generated by $Shadow_{JPF}$ and $Shadow_{KLEE}$ (cf. Table 2). In almost all cases $Shadow_{KLEE}$ generated the same number of *diff* paths. The only differences can be observed for BankAccount.main_11 and BankAccount.main_19, since in both cases the *new* version introduces a potential zero-division error and $Shadow_{KLEE}$ does not find this error (at least not with the configuration we used it). Additionally, we also manually compared the generated path conditions, which matched for all considered subjects.

RQ.2 Generating Regression Test Cases with $Shadow_{JPF}$ that SPF missed

In order to answer RQ.2 we compared the number of test cases generated by pure SPF and $Shadow_{JPF}$ (cf. Table 2). The numbers show that $Shadow_{JPF}$ can reduce the number of generated test cases enormously. All generated test cases of $Shadow_{JPF}$ are real regression test cases because all of them show a divergence between the two versions. In contrast SPF generates a lot of irrelevant paths for regression testing. This is based on the fact that SPF only can consider the information of the *old* or the *new* program version exclusively. As we used the new version to run SPF, the generated path constraints are often too imprecise to trigger a real regression input, i.e., the constraint subsumes the real regression path constraint because SPF misses the crucial information from the *old* version. Then it depends on the value generation of the constraint solver whether the concrete test inputs hit the *diff* path or not. All in all, our results show that $Shadow_{JPF}$ can generate the regression test cases that were missed by SPF.

5. CONCLUSION AND FUTURE WORK

In this work we presented our tool $Shadow_{JPF}$ as an extension of the Java PathFinder project. Our tool applies the idea of shadow symbolic execution [7] on Java bytecode and, hence, makes shadow symbolic execution available to a large range of Java programs. We performed preliminary experiments on 51 generated mutants and compared the results with Symbolic PathFinder and the original implementation $Shadow_{KLEE}$. They show that $Shadow_{JPF}$ can significantly reduce the number of test cases compared to SPF, and that it behaves like the original implementation in $Shadow_{KLEE}$. Although these results are very promising, the analyzed subjects are relatively small, and hence, we cannot generalize the results to larger and real-world examples.

In future we plan to extend our implementation and build on top of $Shadow_{JPF}$ a more powerful tool for the generation of regression test cases. This includes the full automation of the change annotation, which currently is done manually. Additionally, we want to extend our evaluation by adding more JPF compatible classes and real-world regression bugs to our analysis.

6. REFERENCES

[1] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[2] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[3] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[4] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[5] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[6] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[7] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1181–1192, New York, NY, USA, 2016. ACM.

[8] 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*, 20(3):391–425, 2013.

[9] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003.

[10] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.