AB=ƎA: Execution Equivalence as a New Type of Testing Oracle*

A. Elyasov, W. Prasetya, J. Hage
Utrecht University
Utrecht , The Netherlands
{a.elyasov,s.w.b.prasetya,j.hage}@uu.nl

U. Rueda, T. Vos, N. Condori-Fernández
Universitat Politècnica de València
Valencia, Spain
{urueda, tvos}@pros.upv.es,
n.condori-fernandez@vu.nl

ABSTRACT

This paper introduces a new type of automated testing oracle, called the *execution equivalence (EE) invariants*. These invariants can be mined from application logs that capture both application events and application states. The EE-invariants express an equivalence relation on the sequences of application events in terms of equality of respective initial and final states, which these sequences leave in the logs during the run-time. We claim that even equivalences up to a length of four events already provide useful testing oracle. We extended our tool LOPI (**LO**g-based **P**attern **I**nferencer) with the algorithm for mining EE-invariants, and evaluated the effectiveness of these invariants on a case-study — the web application *Flex Store*. The evaluation is carried out based on two parameters: the *false positive rate* and the *fault finding capability*. Moreover, we compared the strength of LOPI's execution equivalences with Daikon's data invariants. This comparison has shown that Daikon was slightly more effective than LOPI in testing Flex Store. However, we have found a suitable confidence level for LOPI which allows to outperform Daikon.

Categories and Subject Descriptors

F.3.1 [**LOGICS AND MEANINGS OF PROGRAMS**]: Specifying and Verifying and Reasoning about Programs — *Assertions, Invariants, Specification techniques*

General Terms

Experimentation, Measurement, Reliability

Keywords

regression testing, automated oracles, Daikon, inference, logs

*This work was financed by the FITTEST (Future Internet Testing) project, ICT-2009.1.2, no 257574.

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2695877

1. INTRODUCTION

> Testing shows the presence, not the absence of bugs.
> —Edsger W. Dijkstra, 1969

The laconic definition of testing given by Dijkstra more than 40 years ago still strikes by its profundity. But it does not prescribe a recipe to distinguish a bug from a normal behavior. Observing an application failure immediately indicates the presence of an error in the program. But not all errors result in evident failures. Some errors just *infect* the program state while the program still executes normally. The infected state can be identified by contrasting it with an *expected* valid state of the program. Usually, the description of the expected state is informal, and mostly relies on the user's understanding of the program behavior. Therefore, it practically becomes difficult to almost impossible to distinguish erroneous states from valid ones without user intervention. The problem of acquiring the expected states is known in the literature as the *oracle problem* [17]. The purpose of an oracle is to qualify tests as passing or failing.

Most approaches to the construction of automated oracles are based on learning generic properties from a set of trustable executions. Then, these properties are assumed to also hold for new executions of the same program. Thus, the properties can be used for identifying abnormal behaviors of the program. The behavior is considered *abnormal* if it violates one of the properties previously mined from the set of executions observed so far.

Orso and Rothermel in their recent travelogue on advances in software testing for the past 14 years have acknowledged the lack of success in the automated construction of oracles [12]. Despite significant progress in the automation of test generation, the human tester is still the best testing oracle available. Nguyen et al. [11] have evaluated the cost and effectiveness of three state-of-the-art automated oracle techniques: Data Invariants (Daikon) [7], Temporal Invariants (Synoptic) [2], and Finite State Automata (KLFA) [10]. The effectiveness of oracles was defined as a combination of two parameters: the rate of false positives (FP) and the fault finding (FF) capability. They found that despite the ability to reveal faults automatically, all three techniques are suffering from a high FP rate (on average 30%). Such an FP rate in addition to a limited FF capability strongly suggests the infeasibility of the existing automated oracle techniques to be successfully adopted in practice.

The problem of effectiveness of automated oracles motivates us to propose a new class of software oracles, that we

call *execution equivalence invariants (EE-invariants)*. Two executions are considered equivalent if they start in the same initial state, and result in the same final state. We propose to infer EE-invariants from application logs that are represented as alternations of application events and states logged in coordination. The main contributions of this work are the following:

1. We introduce EE-invariants — a new type of automated oracle that can be mined from application logs.
2. The inference algorithm is implemented in the tool called LOPI, which, as our evaluation has shown, can efficiently learn EE-invariants up to length four.
3. We perform a preliminary evaluation of the effectiveness of EE-invariants by measuring their false positives (FP) rate and fault finding (FF) capability. The evaluation has shown that the FP rate lies between 2-30%, whereas FF capability is around 70%. The comparison with Daikon revealed that the effectiveness of EE-invariants is competitive with the ones found by Daikon. We have also found an initial configuration that allows to maximize the effectiveness of EE-invariant and, finally, outperform Daikon.

The paper is structured as follows: The EE-invariants and their inference procedure are presented in Section 2. Section 3 describes the design of our experimental evaluation. Research questions are addressed in Section 4. Future work and threats to validity are discussed in Section 5. Related work is summarized in Section 6. Section 7 concludes the paper.

2. EXECUTION EQUIVALENCE

This section formally defines execution equivalence (EE), introduces EE-invariants, and also describes the inference procedure for the EE-invariants. But, first, we illustrate the concept of EE-invariants by an example.

2.1 Motivating Example

Let us consider a simplified version of the calculator application that can only divide two natural numbers. The application has two editable fields X and Y, where the operands of division are entered, another field to display the result, the button *div* to perform the division, and one extra button *clear* to erase all fields. Figure 1 shows an example log generated by the calculator during one user session. Each row in the log records an application event and the state of the application after the execution of the event. We recognize two event sequences as *equivalent* if they always end up in the same final state, whenever they start in the same initial state. Invariants on the right in Figure 1 are constructed according to the aforementioned definition of equivalence, based on the information in the log on the left. For instance, the first invariant states the commutativity of two events *editX* and *editY*, which can be established by observing the occurrences of these two events in the sample log. The following section formally treats all concepts illustrated here such as events, logs and invariants.

Log	X=?	Y=?
editX	5	?
clear	?	?
editX	6	?
editY	6	1
clear	?	?
clear	?	?
editY	?	1
editX	6	1
div	6	1

EE-invariants:
$[editX; editY] \equiv [editY; editX]$
$[editX; clear] \equiv [clear]$
$[div] \equiv []$

Figure 1: Log File and EE-invariants

2.2 Definitions

As is the case for many types of automated oracles, the EE-invariants are inferred from execution logs. Each log entry records an event that occurred in the application, and the application state after the event has been successfully executed. Events are derived from a finite alphabet $\mathcal{E}$, whereas states result from applying a projection over concrete states of the application. An *execution sequence* or *execution* is a sequence of application events $\tau = e_1, \ldots, e_n$ executed in sequence. We say that τ has length n ($|\tau| = n$), as it consists of n events. There is a special event $\epsilon \in \mathcal{E}$ that when executed never produces any affect on the application state; its length is zero, i.e. $|\epsilon| = 0$. In response to user actions, the application generates log files.

DEFINITION 1 (LOG). *Given an application A in a state s_0, the log file produced by A in response to the execution sequence $\tau = e_1, \ldots, e_n$ is the following sequence:*

$$L = [(\epsilon, s_0), (e_1, s_1), \ldots, (e_n, s_n)],$$

where $e_i \in \mathcal{E}$ is an application event, and $s_i \in \mathcal{S}$ is the application state sampled right after the event was executed.

To sample the initial state of an execution sequence, we assume that the ϵ event always precedes every actual execution. $\mathcal{L}(A)$ is the set of all possible logs produced by the application A in response to all possible execution sequences drawn from the event alphabet $\mathcal{E}$. We can define an equivalence relation on the executions produced by the application A. Two event sequences are equivalent, if starting from equal initial states they end up in equal final states, or more formally:

DEFINITION 2 (EXECUTION EQUIVALENCE). *Given an application A and the set of application logs $\mathcal{L}(A)$, two execution sequences $\tau_1 = e_1, \ldots, e_l$ and $\tau_2 = d_1, \ldots, d_k$ are equivalent, denoted as $\tau_1 \equiv \tau_2$, if $\forall L_1, L_2 \in \mathcal{L}(A)$:*

$$L_1 = [\ldots, (a_i, s_i^1), \overbrace{(e_1, s_{i+1}^1), \ldots, (e_n, s_{i+l}^1)}^{\tau_1}, \ldots]$$
$$L_2 = [\ldots, (b_j, s_j^2), \underbrace{(d_1, s_{j+1}^2), \ldots, (d_k, s_{j+k}^2)}_{\tau_2}, \ldots]$$

the following condition holds: $s_i^1 = s_j^2$ implies $s_{i+l}^1 = s_{j+k}^2$.

The length of an execution equivalence is equal to the sum of the lengths of both sides: $|\tau_1 \equiv \tau_2| = |\tau_1| + |\tau_2|$.

2.3 Inference Procedure

Given a set of application events $\mathcal{E}$ and some $n \in \mathbb{N}$, we can generate all possible equivalences between execution sequences up to length n.

$$\{(\tau_1, \tau_2) \mid \tau_1, \tau_2 \in \mathcal{E}^*,\ |\mathcal{E}| > 1,\ max(|\tau_1|, |\tau_2|) \leq n\},$$

where each pair in the set is an equivalence candidate $\tau_1 \equiv \tau_2$. The number of elements in this set grows exponentially in n. Moreover, with the increase of the length τ_1 or τ_2, the

Algorithm 1: `inferEE(L,n,W)`: inference procedure

```
   Input  : L is a set of application logs
            n is the maximal length of execution sequence
            W = [(1, i_1), ..., (2n, i_2n)] is a witness table
   Output : set of valid invariants I justified on L
1  I ⟵ ∅
2  E ⟵ CollectExecutions(L, n)              // E = [(τ, l, p, s^I, s^F)]
3  E_τ ⟵ GroupAndSort(E)                    // E_τ = [(τ, [(l, p, s^I, s^F)])]
4  for k ← 1 to |E_τ| do
5      for j ← k to |E_τ| do
6          (τ_k, w_k) ⟵ E_τ[k]
7          (τ_j, w_j) ⟵ E_τ[j]
8          l ⟵ |τ_k| + |τ_j|
9          c ⟵ CountWitness(w_k, w_j)
10         if c ≥ W(l) then I ⟵ I ∪ (τ_k ≡ τ_j)
11 return I
12 Function CountWitness(w_1, w_2)
13     c ⟵ 0
14     for i ← 1 to |w_1| do
15         for j ← 1 to |w_2| do
16             (l_1, p_1, s_1^I, s_1^F) ⟵ w_1[i]
17             (l_2, p_2, s_2^I, s_2^F) ⟵ w_2[j]
18             if s_1^I = s_2^I then
19                 if s_1^F = s_2^F then c ⟵ c + 1 else return -1
20     return c
```

likelihood of observing equivalence decreases. Therefore, it is practically desirable to keep the value of n small. For example, we suggest to focus on executions up to length two events.

The inference procedure is sketched in Algorithm 1. The procedure takes a set of application logs, an upper bound on execution length, and a witness table, and it returns a set of invariants justified by the logs. *The witness table* defines a mapping between the equivalence length and the number of times the equivalence is expected to be witnessed in the logs. That is, the equivalence of length k is accepted by the algorithm, if it is observed at least i times for some (k, i) in the witness table. This allows us to avoid accidental equivalences. Initially, the set of invariants is empty (line 1). We start by scanning the set of logs L and collecting all occurrences of executions up to length n (line 2). Each occurrence is represented by a 5-tuple (τ, l, p, s^I, s^F), where τ is an execution that occurred in the log l at a position p, s^I is an initial state preceding the execution τ, and s^F is a final state following τ. Then, we group occurrences based on equal executions and sort them in the shortlex order (line 3). Given the set of executions E_τ, the purpose of the two nested for-loops on lines 4–10 is to form all possible equivalence candidates and check them for validity on the set of logs L.

For an equivalence candidate, the function $CountWitness$ (line 9) returns the total number of witnesses of that equivalence. The code of $CountWitness$ is presented below in the same listing (lines 12–20). A pair of occurrences (lines 16, 17) with equal initial and final states increase the number of witnesses; an occurrences pair with equal initial states but unequal final states terminate the counting function for this pair with the exit code -1; and if no pairs of occurrences with equal initial states are found, zero is returned. Finally, if the number of observed witnesses is greater than or equal to the respective value for that equivalence in the witness table W, we accept the equivalence as a valid oracle and add it to the set O, line 10. Otherwise, the equivalence is rejected and the next equivalence candidate should be considered.

Table 1: User Logs

logs: #	1	2	3	4	5	6	7*	8	9	10	11	12
total events	5	5	40	43	48	79	106	109	149	153	158	230
unique events	4	4	21	15	19	27	68	52	78	51	58	57
event variability	3.2	3.2	11	5.2	7.5	9.2	43.6	24.8	40.8	17	21.3	14.1
							total length in events: 1125					

3. EXPERIMENTAL EVALUATION

Nguyen et al. [11] in their recent study (2013) on cost and effectiveness of automated oracles identified the following four key parameters that determine the practicality of the oracles obtained as result of dynamic log analysis: *false positive (FP) rate*, *fault finding (FF) capability*, *training cost, and checking cost*. Evaluation of our automated oracle, which is represented by EE-invariants, is mostly focused on the first two metrics, namely FP and FF. Though, the questions of inference and checking cost are also addressed. In addition those questions, we discuss the comparative effectiveness of the EE-invariants and Daikon [7] invariants used independently or together. The inference of EE-invariants is implemented in a tool called LOPI (**LO**g-based **P**attern **I**nferencer), which is available for download[1] together with all experimental data[2].

3.1 Experiment Design

For the experimental evaluation of the EE-invariants, we chose the web application *Flex Store*[3]. It is a prototype of a web shop for online purchasing of mobile phones. The application has been developed by Adobe to demonstrate the facilities of the Flex Framework, which provides an extensive collection of highly customizable GUI components. The application consists of 20 source files with a total of 2620 lines of code. We instrumented all GUI elements such that each user action emits the respective message into the log. The FITTEST Automation Framework[4] provides convenient means to enable GUI logging for Adobe Flex applications. It is supplied with an extensive collection of logging delegates for clickable GUI elements. We only had to manually specify the application state — a collection of variables sampled after a GUI event is triggered. The application state was characterized by seven variables representing different aspects of the application such as the number of phones in the shopping cart, and the number of currently visible phones in the catalog. As a result, all user activities such as browsing and filtering were traced in the log together with the respective application states.

Twelve different participants were requested to explore the functionality of Flex Store within a fixed period of time. In total we have obtained twelve different logs of lengths from 5 to 230 events, Table 1. The logs were stored in the FITTEST Logging Format [13] that captures information about both the application events and states. The FITTEST toolset provides a utility called (*haslog*) with the following functionalities: conversion of the "raw" FITTEST log (.log) to XML, "raw" log compression, conversion to Daikon's traces (.dtrace) etc. LOPI directly operates on the logs

[1] https://github.com/aelyasov/LOPI
[2] https://github.com/aelyasov/LopiOracleEvaluation
[3] http://www.adobe.com/devnet/flex/samples/flex_store_v2.html
[4] https://code.google.com/p/fittest/

produced in the FITTEST format, whereas in the case of Daikon, logs have to be converted to dtraces.

Conversion to Daikon The translation of FITTEST logs to the Daikon format is a complex procedure. Some relevant details of this procedure are exposed below. First, each variable in a dtrace should be declared in advance, together with its attributes such as its type. Conveniently, all dtrace logging points (ppt) have exactly the same set of declarations because the application states in the FITTEST logs consist of the same set of variables. Second, states and events in the FITTEST logs should be mapped to their counterparts in the dtraces. This way each variable in a FITTEST log is mapped to the corresponding variable of the dtrace. Every dtrace event consists of two components: ENTER and EXIT. During the conversion, the state preceding the event is associated with the ENTRY ppt, while the current state is coupled with the EXIT ppt.

4. RESEARCH QUESTIONS

The invariants (oracles) are valuable artifacts in software testing even though they neither sound nor complete [15]. But the level of "imperfection" should be precisely estimated, and possibly minimized. The practicality of invariants is defined by the rate of spurious warnings they give rise to and the number of real faults that the invariants are able to catch. Considered together these two parameters constitute *the accumulated effectiveness* of the invariants. Below is the list of research questions that should be addressed for EE-invariants in order to evaluate their potential for software testing.

RQ1 What is the rate of false positives?

RQ2 What is the fault finding capability?

RQ3 How to maximize the accumulated effectiveness?

RQ4 Which invariants are more effective: Daikon or LOPI? Can we combine them to achieve better effectiveness?

RQ5 What is the inference and checking cost?

In the rest of this section, we will try to answer all of these questions for EE-invariants.

It is evident that the quality of invariants depends on the set of logs used in the inference. Poor and non representative logs supplied for the inference can cause a high FP rate and low FF capability. To infer EE-invariants, we have exploited the complete set of logs collected in the experiment described in Section 3.1, which provides almost 100% line coverage of the Flex Store source code. We assume that satisfying this coverage criterion is a reasonable starting point to proceed with the evaluation.

4.1 FP Rate

Given an application and a set of invariants for this application, the *FP rate* is the percentage of false alarms that have been raised by the invariants on a set of logs produced by perfectly valid executions. Given a set of invariants I, the *absolute* false positive rate of I is the percentage of all possible valid logs rejected by I:

$$FP_{abs}(I) = \frac{|\text{all possible logs rejected by } I|}{|\text{all possible logs}|}.$$

Since the number of executions is infinite (or just huge), practical treatment of the FP rate requires to define a *relative* FP rate (with respect to some set of *validation logs* L):

$$FP_{rel}(I, L) = \frac{|\text{logs from } L \text{ rejected by } I|}{|L|}.$$

In place of L in the definition of FP_{rel}, we substitute the *test cases logs*, that is the logs that result from the execution of the Flex Store test cases generated by the FITTEST ITE [16]. The ITE supports several techniques for test case generation. We chose the state-based test case generation [9]. This technique was applied to the original user logs, and it produced 6317 executable test cases with an average length of 10 events. These test cases exhibited a lot of new executions not originally covered in the user logs.

Results: The level of the FP rate is presented by the columns **#FP** and **FP%** in Table 2. The first column says how many traces out of 6317 valid ones were rejected by the invariants, when the second is their respective percentage. We can see that LOPI (first row) has raised 30% FP rate. The invariants with such a high FP rate are often considered to be difficult to deal with in practice. In Section 4.3 we discuss how to handle this problem. The top part of Table 3 presents how the FP rate varies for the EE-invariants of different length ($n = i, i \in [1, 4]$). It is clear that the value of FP rate decreases with the length (n) of the invariant. For instance, the invariant of length one have contributed to the FP rate by 22%, whereas those of length four gave only 0.4% of FPs.

4.2 FF Rate

The capability to detect faults characterizes the potential of invariants to discover errors in the programs. But how can this potential be measured? Program mutations provide an effective way to compare the *power* of test suites. Given a program and a set of mutations of the program, one test suite is considered *more powerful* than the other, if it kills more mutations. The so-called *mutation coverage* is a useful measure for the quality of test suites.

In a similar way, mutations can be exploited for establishing of the *invariant's power*. Given a program, a set of mutations on the program, and a test suite, invariants that are able to kill a larger number of mutations for the given test suite are more suitable for testing that program than others. The usual way to estimate fault detection capability of invariants requires the following three steps:

1. inject a number of different faults into the application code, one at a time (the more faults the better);
2. drive the application to trigger the faults, consequently leaving some footprints in the corresponding log files;
3. check the generated logs for violation of any invariant.

It is clear that one set of invariants is more powerful than another, if it reveals more injected faults. In our case, implementation of the first two steps above meets some difficulties. First, we have to have access to the history of real faults in Flex Store, which is not the case for us. Moreover, such a history often is incomplete due to the limitations of testing. Second, each fault should have a test case exposing it to the outside world. And finally, this test should leave some

Table 2: Summary of Evaluation Results

Invariants	Mutation Operators														#FF	#FP	FF%	FP%	P	R	F_1	F_2	$F_{0.5}$
	ABS	DAR	DEC	EAR	EPS	INC	MIN	RAR	RLE	RVS	SEV	SLE	SST	ZER									
LOPI	56	186	2019	186	184	2019	1929	182	8	92	2014	4558	4925	1929	20287	1936	75.9	30.6	0.694	0.759	0.725	0.745	0.706
LOPI* (optimal)	56	180	1959	180	180	1959	1869	176	7	90	1587	4300	4897	1869	19309	139	72.3	2.2	0.978	0.723	0.831	0.763	0.914
Daikon	44	142	1525	143	206	1498	1897	143	16	70	4272	4655	4752	1840	21203	1718	79.4	27.2	0.728	0.794	0.76	0.78	0.74
LOPI ∩ Daikon	42	114	1345	117	179	1305	1685	122	4	60	1766	4181	4663	1639	17222	822	64.5	13	0.87	0.645	0.741	0.68	0.813
LOPI ∪ Daikon	58	214	2199	212	211	2212	2141	203	20	102	4520	5032	5014	2130	24268	2832	90.8	44.8	0.552	0.908	0.687	0.804	0.599
LOPI* ∩ Daikon	42	114	1345	117	179	1305	1685	122	4	60	1766	4181	4663	1639	17222	67	64.5	1.1	0.989	0.645	0.781	0.693	0.894
LOPI* ∪ Daikon	58	214	2199	212	211	2212	2141	203	20	102	4520	5032	5014	2130	24268	1790	90.8	28.3	0.717	0.908	0.801	0.862	0.748
Synoptic	0	0	0	0	0	0	0	0	67	0	5270	5270	0	0	10607	6286	39.7	99.5	0.005	0.397	0.01	0.024	0.006
KLFA	0	0	0	0	0	0	0	0	99	0	5493	5494	0	0	11086	6305	41.5	99.8	0.002	0.415	0.004	0.01	0.002
Total Mutants	58	221	2269	221	212	2269	2172	214	106	106	5565	5565	5565	2172	26715	6317							

Table 3: Evaluation of LOPI EE-invariants

Length	#W	FP%	FF%	P	R	F_1	F_2	$F_{0.5}$
$n = 1$	1	22.3	60.9	0.777	0.609	0.683	0.637	0.736
$n = 2$	1	6.3	57.6	0.937	0.576	0.713	0.624	0.833
$n = 3$	1	3.4	62.3	0.966	0.623	0.757	0.671	0.87
$n = 4$	1	0.4	14.1	0.996	0.141	0.247	0.17	0.45
$n = 1$	3	0.2	54.5	0.998	0.545	0.705	0.744	0.856
$n = 2$	2	1.6	53.3	0.984	0.533	0.691	0.613	0.842
$n = 3$	2	0.5	55.8	0.995	0.558	0.715	0.628	0.86
$n = 4$	1	0.4	50.9	0.996	0.509	0.674	0.564	0.836

Table 4: Log Mutation Operators

Oper.	Description	initial log	mutated log
RLE	remove a log entry	$l; m; n$	$l; n$
SLE	swap two log entries	$l; m; n$	$n; m; l$
SEV	swap two events	$e\langle p\rangle; f\langle q\rangle; d\langle r\rangle$	$d\langle p\rangle; f\langle q\rangle; e\langle r\rangle$
SST	swap two states	$e\langle p\rangle; f\langle q\rangle; d\langle r\rangle$	$e\langle r\rangle; f\langle q\rangle; d\langle p\rangle$
INC	increase by one an integer var	$e\langle v = 2\rangle; f$	$e\langle v = 3\rangle; f$
DEC	decrease by one an integer var	$e\langle v = 2\rangle; f$	$e\langle v = 1\rangle; f$
ABS	absolute value of an integer var	$e\langle v = -2\rangle; f$	$e\langle v = 2\rangle; f$
ZER	assign zero to an integer var	$e\langle v = 2\rangle; f$	$e\langle v = 0\rangle; f$
MIN	negate an ineteger var	$e\langle v = 2\rangle; f$	$e\langle v = -2\rangle; f$
EPS	assign empty to a string var	$e\langle v = \text{“}a\text{“}\rangle; f$	$e\langle v = \text{“”}\rangle; f$
RVS	reverse a string var	$e\langle v = \text{“}ab\text{“}\rangle; f$	$e\langle v = \text{“}ba\text{“}\rangle; f$
EAR	empty an array var	$e\langle v = [1, 2]\rangle; f$	$e\langle v = []\rangle; f$
RAR	reverse an array	$e\langle v = [1, 2]\rangle; f$	$e\langle v = [2, 1]\rangle; f$
DAR	drop the last elem. of an array	$e\langle v = [1, 2]\rangle; f$	$e\langle v = [1]\rangle; f$

evidence in the log file. Otherwise, the invariants would be blind to the introduced faults.

Since the faults are detected based exclusively on the information present in the logs, we propose to *inject faults explicitly in the logs* instead of the application code. This decision, of course, has its own pros and cons. On the one hand, it solves all issues with the mutation-based approach concerning real fault injection. On the other hand, some of the log mutations may have no associated faults, i.e. they will be infeasible program errors.

The log-based mutation approach requires some *ground string* to be used in place of a mutation object. The ground string could be chosen among twelve user logs. Applying mutations to all logs in turn is infeasible since each log produces thousands of mutants. Therefore, we decided to select one representative log file out of the 12 available and apply mutation operators to it. We have chosen log #7 from Table 1 to play the role of the *ground string* because it has the highest *event variability*, defined as $\frac{|\text{unique events}|^2}{|\text{total events}|}$. Analogous to the traditional program mutations [1], we introduced a set of mutation operators, *mutators*, on logs in Table 4. These mutators describe structural log transformations, which are driven by the abstract syntax of the logs [13]. A log is a list of event-state entries. There are two types of values stored in states: *primitive* and *object*. A primitive value is one of the primitive types such as *integer* and *string*. An object value is a collection of different values that together define the type of the object. An array is a special type of objects that consists of values of a uniform type.

To systematically describe log transformations, we grouped them into three categories in Table 4: 1) log entry mutators (RLS–SST); 2) primitive mutators (INC–RVS); and 3) array mutators (EAR–DAR). Each operator was exhaustively applied to the ground string (log #7). This process derived a total of 26715 mutants. The FF capability in this settings has been measured as *the number of mutants killed by the invariants*.

Results: In Table 2, the columns **#FF** and **FF%** indicate respectively the number of killed mutants (out of 26715) and their percentage rate. LOPI identified almost 76% of all mutations. In Table 2, we can also see the number of killed mutants per mutation category, where the bottom row shows the total number of mutation in each category. The killing rate of LOPI in all categories except two (RLE and SEV) is close to 90%. RLE and SEV are outliers because the ground string (initial log) has a high density of skip-like events. When an entry with such event is removed or two skip events are swapped, the effects of the corresponding mutations are unnoticeable for the most of the EE-invariants. Another consequence of having a large number of skip events is almost 60% FF capability of the invariant of lengths 1–3 in Table 3. For instance, if a and b are both skip events, i.e $a \equiv b \equiv \epsilon$, then the following invariants also hold: $ab \equiv \epsilon$, $a \equiv b$, $ab \equiv b$, etc.

4.3 Accumulated Effectiveness

There is a duality between FP rate and FF capability. To measure the former we assume that the invariants are complete, provide an additional set of valid traces, and count how many of them are rejected by the invariants. The smaller this number is, the lower the FP rate of the invariants. To measure the latter, we assume that the invariants are sound, provide an additional set of invalid traces, and count how many of them are spotted by the invariants. The greater this number is the higher the FF capability of the invariants. The cumulative effectiveness of invariants is composed of both the FP rate and the FF capability. Making inference more precise, we decrease the FP rate, but this often also reduces the FF capability. Inversely, increasing the FF capability of invariants typically increases the FP rate. This tight correlation requires to make compromises selecting the invariants for testing.

We use the F_β *score* — a measure of a test's accuracy taken from information retrieval [14] — to measure the cumulative effectiveness of invariants, which depends on both the FP rate and FF capability. It is defined in terms of the *precision* P and *recall* R according to the formula:

$$F_\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{(\beta^2 \cdot P) + R}$$

The value of the F_1-score is the *harmonic mean* of the precision and recall, i.e. they are both equally weighted. Op-

Table 5: Evaluation of Daikon Invariants

Invariants	FP%	FF%	P	R	F_1	F_2	$F_{0.5}$
EltLowerBound	0.8	10.2	0.992	0.102	0.185	0.124	0.361
EltOneOf	4.1	14	0.959	0.14	0.244	0.169	0.442
EltwiseIntGreaterThan	0	3.6	1	0.036	0.069	0.045	0.157
EltwiseIntLessThan	0.8	1.7	0.992	0.017	0.033	0.021	0.08
IntEqual	6.2	42.9	0.938	0.429	0.589	0.481	0.758
IntGreaterEqual	0.4	4.4	0.996	0.044	0.084	0.054	0.187
IntGreaterThan	2	2.9	0.98	0.029	0.056	0.036	0.13
IntLessEqual	3.9	3.5	0.961	0.035	0.068	0.043	0.153
IntLessThan	5.7	7.9	0.943	0.079	0.146	0.097	0.296
IntNonEqual	3	16.5	0.97	0.165	0.282	0.198	0.491
LinearBinary	0	0.8	1	0.008	0.016	0.01	0.039
LowerBound	1.2	2.6	0.988	0.026	0.051	0.032	0.118
NumericInt$Divides	1	4.1	0.99	0.041	0.079	0.051	0.176
OneOfScalar	17	44.6	0.83	0.446	0.58	0.491	0.708
OneOfSequence	18.3	50.6	0.817	0.506	0.625	0.548	0.728
OneOfString	6.3	27.4	0.937	0.274	0.424	0.319	0.631
SeqSeqIntEqual	3	51.7	0.97	0.517	0.674	0.57	0.825
SeqSeqIntGreaterEqual	0	4.5	1	0.045	0.086	0.056	0.191
SeqSeqIntGreaterThan	0	1.3	1	0.013	0.026	0.016	0.062
SeqSeqIntLessEqual	0.1	1	0.999	0.01	0.02	0.012	0.048
StringEqual	0	23.4	1	0.234	0.379	0.276	0.604
StringLessThan	0	0.7	1	0.007	0.014	0.009	0.034

positely, the F_2 and $F_{0.5}$ scores prioritize either higher recall or higher precision respectively.

Results: We calculated the values of P, R, F_1, F_2 and $F_{0.5}$ for LOPI; they are presented in Table 2. The recall (R) corresponds to the FF rate, whereas the precision (P) is the value complimentary to the FP rate (1 − FP rate). The value of F-score should lie between 0 and 1; a higher value corresponds to more effective invariants.

One of the input parameters for the inference algorithm `inferEE` is *the invariant's witness table W*, which specifies the number of witnesses required for the invariant of a certain length in order to be accepted by the algorithm. The second row in Table 2 (LOPI*) corresponds to the witness table that *maximizes* the value of F_1-score (0.831). The number of required witnesses per invariant is shown in the column **#W** of Table 3. The bottom part of the table shows the number of witnesses maximizing F_1. For instance, by increasing the limit from 1 to 3 witnesses, we reduce the FP-rate by 22% for the EE-invariants of length one. As result, instead of the initial 30% FP rate of LOPI, we got only 2.2% of FPs. But, it comes at the price of decreased FF capability (72.3% instead of the original 75.9%).

4.4 Comparison with Daikon

Daikon [7] is one of the most representative tools in the field of automated oracle inference. Daikon expresses *data invariants* over program variables. During the inference, the values of program variables sampled at various program points are consolidated together and substituted for the variable placeholders into the invariant templates from the Daikon's catalog. Daikon is shipped with an extensive collection of invariant templates which result in a large set of concrete invariants. The inference process consists of the cross checking of the invariant templates against the consolidated values. Those invariants that can stay valid and pass the confidence level check are reported to the user. In the experiment we used Daikon 5.1.0[5] with default configurations to infer invariants from user logs translated to dtraces. The log translation process is described is Section 3.1.

Results: All measurements related to Daikon are presented in the third row of Table 2. The rate of FPs for Daikon is almost as high as for LOPI (27% vs. 30%). Nevertheless, Daikon discovered 79% of mutants, which is higher than the corresponding value of LOPI and LOPI*. Comparing the values of F_1-score of LOPI and Daikon, we can deduce that generally *Daikon invariants are slightly more effective for testing than LOPI*: 0.76 vs. 0.725. Nevertheless, Daikon was beaten by LOPI*: 0.76 vs 0.831.

[5] http://plse.cs.washington.edu/daikon/

Table 6: Inference and Checking Cost of the Invariants

Invariants	Inference		Checking	
	Time (sec.)	Space (MB)	Time (sec.)	Space (MB)
LOPI	223	61	1.5	128
Daikon	7	398	3	184

Table 5 presents the effectiveness of the particular Daikon invariants. In total there were 22 types of Daikon invariants involved in the evaluation. The invariants of the types `OneOfSequence` and `OneOfScalar` triggered the highest FP rate: 18.3% and 17% respectively. They describe the property stating that an array or a scalar variable of type *long* "takes on only a few distinct values" [6]. But at the same time, these invariants have a relatively high FF capability, 50% and 45% respectively. The invariants `OneOfString` and `IntEqual` with the FP (FF) rates of 6.3% (27%) and 6.2% (43%) respectively are in the second place. The `OneOfString` invariant is similar to those two examples of `OneOf`-invariants seen above, except that the variable's type should be *String*. The `IntEqual` invariant express "an equality between two *long* scalars" [6]. All other invariants have an FP rate lower than 6% and will not be discussed separately.

The second reasonable question to ask when comparing two types of invariants is: can the invariants be combined together to improve the value of the F_1-score? We investigated this question by calculating the F-scores for the intersection and union of Daikon and LOPI (LOPI*) invariants respectively. In theory, unification of invariants should potentially increase the mutation killing rate, because the mutants killed by any invariant in the union are reported together. At the same time, this process also increases the rate of FPs since both invariants produce false positives. The situation with the invariant's intersection is inverse: less mutants can be killed but FP rate is also lower. The summary of results is presented in Table 2 (last four rows). Among all available combinations LOPI* still has the highest values of F_1 and $F_{0.5}$ (0.831 and 0.914 respectively). Its extremely high precision (0.978) together with a reasonable recall (0.723) provided the success. Whereas, if the recall is more important, then the union of LOPI* and Daikon outstrips the other combinations with a value of $F_2 = 0.862$.

4.5 Inference and Checking Cost

We have not yet discussed the inference and checking cost of the EE-invariants. The cost of invariant inference is the time (*wall-clock time*) and space (*Resident Set Size*) that is required for the inference procedure to complete. Analogously, we define the cost of invariant checking, where each inferred invariant is examined on a newly provided data set. Our tool LOPI supports both of those features. The inference cost was measured on 12 user logs, which were passed to LOPI as one input argument. The resulted invariants were used for measuring checking cost. Since all generated mutants have similar checking cost we have just randomly selected one to be used in the current experiment. All experiments presented in this section were carried out on an Intel i5 (2.4 GHz) machine with 6GB of RAM under control of Ubuntu 14.04 OS.

Even though we only considered executions up to maxi-

mum two events, the inference reported 6878 oracles out of about $2.5 * 10^5$ equivalence candidates. The inference algorithm is implemented in Haskell, which allows us to lazily deal with hundreds of thousands of candidates consuming little space. Table 6 summarizes the results of all experiments for both LOPI and Daikon. The inference time of Daikon is almost 30 times faster than LOPI (7 sec vs. 3.41 min). However, LOPI consumes much less memory. Improving LOPI's run-time is future work that we discuss in Section 5. Invariant checking time for LOPI and Daikon is equal to 1.5 and 3 seconds respectively.

5. DISCUSSION AND FUTURE WORK

The new type of oracle introduced in this paper — EE-invariants — as well as many other oracles dynamically inferred from the application logs suffer from a lack of precision. To alleviate this problem, the inference procedure should be augmented with some threshold values for invariants of different types. Our evaluation has shown that the threshold values of the EE-invariants are inversely proportional to the lengths of the invariants. In the algorithm `inferEE`, the threshold values correspond to the rows of the witness table W. The experiment for maximization of the F_1-score in Section 4.3 suggests that the witness table corresponding to LOPI* could be used as the default value for the respective parameter (W) in `inferEE`.

Our tool LOPI allows to pass the witness table as a command line parameter. Similarly Daikon lets users alter the confidence level. We found that changing this parameter does not decrease the FP rate for Daikon. The problem is that the confidence is not defined in terms of the number of witnesses of a given invariant. However, there exists an option for redefining the confidence by modifying the respective method in the source code. We saw in Table 5 that Daikon invariants have different strengths. Therefore, we believe it should be possible to find a combination of invariant types that maximizes the value of F_1-score. For example, in our case study the `OneOfSequence` invariant can be completely replaced by `SeqSeqIntEqual` since the latter has lower FP rate but still identifies a comparable number of mutants.

To improve the run-time of LOPI we need to shrink the number of potential candidates to be the invariant. The following heuristics could significantly help there: 1) only skip-like invariants are inferred, and then 2) corresponding skip events are used to simplify all other equivalence candidates. For applications such as Flex Store, which have a large number of skip-like events, these heuristics should be especially effective. Moreover, by making the inference concurrent we can also speed up the run-time of our algorithm.

In our study of the EE-invariants we have limited ourselves to only executions up to length two. In the future, we would like to go beyond that limit.

Threats to Validity: Our comparative study of invariant effectiveness has several threats to validity. The main threat is *external validity* since there was only one subject application involved. Thus, our findings about the FP rate and FF capability of LOPI should be generalized with caution. At the same time, the findings about the FP rate of Daikon are trustworthy since they do not contradict with the previously reported results [11]. An *internal* threat to validity is the set of logs used in the evaluation. There are two sources for this threat. First, we had only one set of logs consisting of 12 executions. Second, there was only one state abstraction in use.

6. RELATED WORK

In 2013 Harman et al. completed a survey on software testing oracles [8]. Despite the number of reported techniques, the problem of how to generate effective automated oracles with a low level of false positives is still not completely addressed. This conclusion is confirmed by the empirical validation of automated oracles carried by Nguyen et al. [11]. They compared the effectiveness of the three state-of-the-art oracles represented by Daikon [7] (data invariants), Synoptic [2] (temporal invariants) and KLFA [10] (finite state automata). For both Daikon and Synoptic the FP rate was on average 30%, whereas KLFA showed a rate of 90%. The evaluation of Daikon on the Flex Store logs in our study has also confirmed the expected 30% rate of FPs. We believe our approach of measuring FF capability based on log mutations provides a statistically more confident result in contrast with [11], which was limited to only seven faults seeded into the subject application.

We have also evaluated Synoptic and KLFA on the Flex Store logs; the results are shown in Table 2. Both types of oracles were inferred only based on the underlying sequences of events, i.e. states were thrown away. As result, the oracles were not able to detect mutations in the states, and the FP rates were higher than 90%.

Elyasov et al. [5] introduced three types of *equivalence patterns* that correspond to a subset of the EE-invariants of length four. The EE-invariants are reminiscent of the cross-checking oracles introduced by Carzanga et al. [3]. These oracles represent pairs of the method calls sequences that are observationally equivalent. To obtain the equivalences, it was suggested to adapt the previous work of Carzanga et al. [4] on automatic workarounds, which essentially are equivalent executions. They proposed three generic types of workarounds, but the approach still relies on manual specification of concrete workarounds. Whereas, the EE-invariants can be automatically inferred from logs.

7. CONCLUSION

In this paper, we introduced a new type of oracle, namely EE-invariants, which describe the equivalences of executions. These invariants can be used for in-house testing or runtime monitoring. The preliminary evaluation has shown that the EE-invariants give rise to approximately 30% of false positives, but this rate can be pushed down to 2% by choosing suitable initial parameters for the inference procedure.

Applying extensive mutation analysis, we have experimentally justified that the EE-invariants are competitive to Daikon invariants for detecting faults. But at the same time, we have discovered several hindrances of our invariants. First, as do most of automated oracles, EE-invariants trigger a high rate of false positives. However, we found an optimal witness table that reduces the FP rate to just few percent. Second, the inference time indicates a potential issue with the scalability of LOPI to handle inference of the EE-invariants of length greater than two, as well as to sustain the increase in the number of events. Nevertheless, current inference time seems reasonable for applications of scale similar to Flex Store.

8. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[2] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*, pages 267–277, 2011.

[3] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of ICSE*, pages 931–942, 2014.

[4] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of FSE*, pages 237–246, 2010.

[5] A. Elyasov, I. W. B. Prasetya, and J. Hage. Guided algebraic specification mining for failure simplification. In *Testing Software and Systems*, pages 223–238. 2013.

[6] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The daikon invariant detector user manual, 2007.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[8] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical report, Tech. Rep. CS-13-01, 2013.

[9] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Procedding of ICST*, pages 121–130, 2008.

[10] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of ISSRE*, pages 117–126, 2008.

[11] C. D. Nguyen, A. Marchetto, and P. Tonella. Automated oracles: an empirical study on cost and effectiveness. In *Proceedings of FSE*, pages 136–146, 2013.

[12] A. Orso and G. Rothermel. Software testing: a research travelogue (2000–2014). In *Proceedings of ICSE, FOSE*, 2014.

[13] I. Prasetya, A. Elyasov, A. Middelkoop, and J. Hage. Fittest log format (version 1.1). *Tech. Rep. UU-CS-2012-014*, 2012.

[14] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[15] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *ICSE*, pages 391–400, 2011.

[16] T. Vos, P. Tonella, J. Wegener, M. Harman, W. Prasetya, E. Puoskari, and Y. Nir-Buchbinder. Future internet testing with fittest. In *Proceedings of CSMR*, pages 355–358, 2011.

[17] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.