To find the effectiveness of ADFD and ADFD+ techniques

Mian Asbat Ahmad and Manuel Oriol

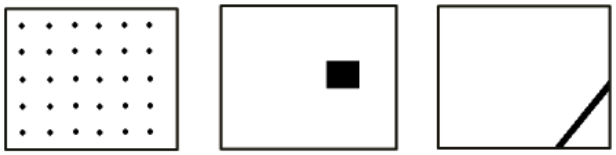
***Abstract*—The achievement of up-to 50% better results by Adaptive Random Testing verses Random Testing ensures that the pass and fail domains across the input domain are highly useful and need due consideration during selection of test inputs. The Automated Discovery of Failure Domain (ADFD) and its successor Automated Discovery of Failure Domain+ (ADFD+) techniques, automatically find failures and their domains in a specified range and provides their visualisation. They can precisely detect the failure-domain of the identified failure in an effective way. Performing exhaustive testing in a limited region around the failure is the key to the success of ADFD and ADFD+ techniques.**

We performed an extensive experimental analysis of Java projects contained in Qualitas Corpus for finding the effectiveness of automated techniques (ADFD and ADFD+). The results obtained were analysed and cross-checked using manual testing. Furthermore the impact of nature, location, size, type and complexity of failure-domains on the testing techniques were also studied. The results provide insights into the effectiveness of automated techniques and a number of lessons for testing researchers and practitioners.

***Index Terms*—software testing, automated random testing, manual testing, ADFD, Daikon.**

I. Introduction

The input-domain of a given SUT can be divided into two sub-domains. The pass-domain comprises of the values for which the software behaves correctly and the failure-domain comprises of the values for which the software behaves incorrectly. Chan et al. [1] observed that input inducing failures are contiguous and form certain geometrical shapes. They divided these into point, block and strip failure-domains as shown in Figure 1. Adaptive Random Testing (ART) achieved up to 50% better performance than random testing by taking into consideration the presence of failure-domains while selecting the test input [2].



(a) Point domain (b) Block domain (c) Strip domain

Fig. 1. Failure domains across input domain [1]

The cost of software testing alone constitute about half of the total cost of software development. Software testing is an essential but expensive process which becomes particularly tedious, laborious and error-prone when performed manually [3]. Automated testing is an alternative approach to manual testing. The case study reveals that the 150 hours of automated testing found more faults in complex .NET code than a test engineer finds in one year by manual testing [4].

We have developed two fully automated techniques ADFD [5] and ADFD+ [6], which effectively find failures and their domains in a specified range and provides visualisation of the pass and fail domain as either point, block or strip failure-domain. This is achieved in two steps: first, random testing is used to find the failure and secondly, exhaustive testing in a limited region around the detected failure is done to identify the failure domain. The ADFD strategy searches in one-dimensional and covers longer range than ADFD+ which is more effective in multi-dimensional and covers shorter range.

Both ADFD and ADFD+ techniques are formed by the combination of three separate tools i.e. YETI, Daikon and JFreeChart. York Extensible Testing Infrastructure [7] is used to test the program automatically using random strategy (ADFD and ADFD+). Daikon [8] observes the all the test execution and automatically generate invariants to show the failure-domains. Finally, JFreeChart [9] generates the graph from the test output to show the pass and failure-domains.

Software testing can be performed either automatically or manually. Both the techniques have their own advantages and limitations. The main advantage of automated testing is execution of large number of tests in little time, whereas manual testing utilizes the tester experience to concentrate on error-prone part of the SUT and generate target oriented test cases [10].

The rest of the paper is organized as follows: Section II presents an overview of ADFD+ technique. Section III evaluates and compares ADFD+ technique with Randoop. Section IV reveals results of the experiments. Section V discusses the results. Section VI presents the threats to validity. Section VII presents related work. Finally, Section VIII concludes the study.

II. Automated Techniques

The two automated techniques used in our experiments are ADFD and ADFD+. A short overview of both the techniques and the enhancements that have been made to both the techniques along with a motivating example are given below:

add the upgraded workflow with identification of the difference

A. Overview of ADFD technique

Automated Discovery of Failure Domain is an automated technique for finding and drawing the failure domain of detected failure in the input domain. ADFD searches for failure domain between the specified lower and upper bound in uni-direction. It test and note only the ranges of pass and fail values and uses the scatter graph of the JFreeChart to plot them on the screen. For more details please see [].

B. Overview of ADFD+ technique

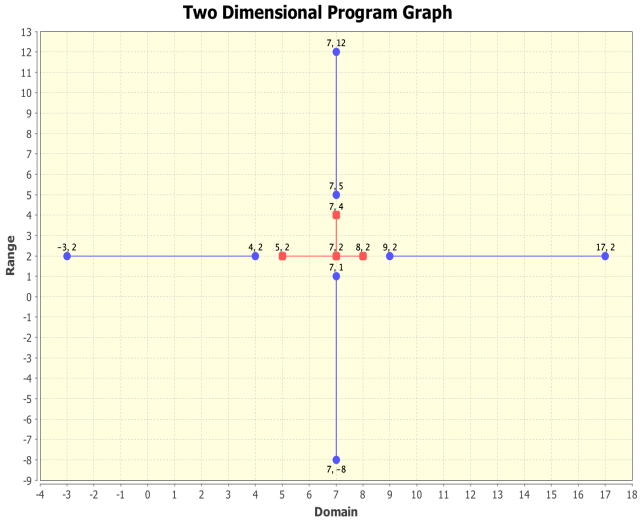
Automated Discovery of Failure Domain+ is an automated technique for finding and drawing the failure domain of detected failure in the input domain. ADFD+ searches for failure domain around the failure in specified range in multi-idirection. It test and note individual value as either pass or fail. The values are drawn using the vector graph of the JFreeChart. For more details please see []

C. Enhancement to ADFD and ADFD+ technique

Integration of Daikon Extension to support primitive types other than int type. Display of the test case causing an error in the sut.

D. Motivating Example to explain ADFD and ADFD+

```
public class BlockErrorPlotTwoShort {

    public static void blockErrorPlot (int x,
        int z;

        if ((x >= 5) && (x <= 8) && (y==2))
            { z = 50/0;}


        if ((x >= 5) && (x <= 8) && (y == 3))
            { z = 50/0;}


        if ((x >= 5) && (x <= 8) && (y == 4))
            { z = 50/0;}

    }
}
```




```
=============Test LOGS=============

 Candidate invariant:
i != 0
i <= 8
i >= 5
j one of { 2, 3, 4 }
j != 0
i > j

=============Test Case=============

[   /** Test case automatically generated by YETI **/
  @Test public void test_0() throws Exception {
    BlockErrorPlotTwoShort.blockErrorPlot((int)7,(int)4);
    /**BUG FOUND: RUNTIME EXCEPTION**/
    /** java.lang.ArithmeticException: / by zero
                at BlockErrorPlotTwoShort.blockErrorPlot(BlockErrorPlotTwoShort.java:15)**/
  }
]
```

Fig. 2. Graph and Invariants generated by ADFD

add picture of graph generated due to ADFD+
Also show the invariants and the log
you can also show the full interface of the program.

III. Automated Discovery of Failure Domain+

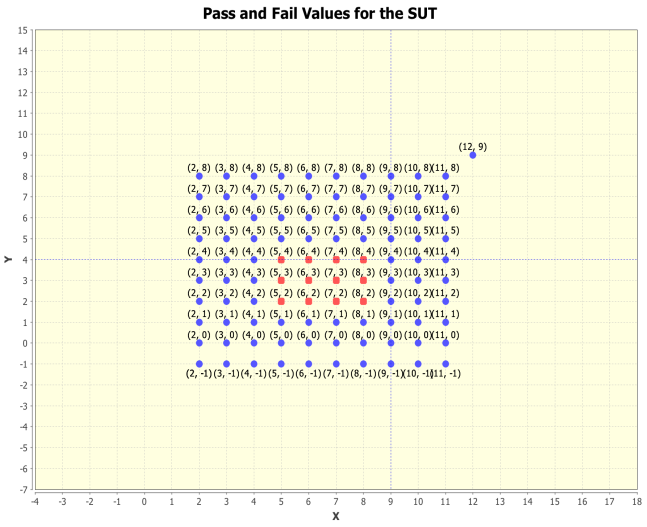
IV. Evaluation

To evaluate the presence, nature and type of failure-domains in production software we tested the main jar files of all the 106 projects in Qualitas Corpus []. The source code of the programs containing failure-domains were also evaluated manually to verify the conformance of automated results. Only one and two dimensional numerical programs were selected for evaluation. Every program was tested independently by ADFD, ADFD+ and manual testing. All the programs in which failure-domains were identified are presented in Table **??**. Due to the absence of contracts and assertions in the code under test, undeclared exceptions were taken as failures in accordance with the previous studies [5], [11].

A. Research questions

The following research questions have been addressed in the study:

1) *Is ADFD and ADFD+ techniques capable of correctly identifying failure-domains in production software?* The experimental results claiming the correct identification of ADFD and ADFD+ were based on the purpose build error-seeded programs []. To answer the question, we applied the two techniques to all the projects of Qualitas Corpus and examined the results.
2) *What is the frequency of existence of point, block and strip failure-domains in production software?*
3) *What are the types of identified failure-domains?* There




```
=============Test LOGS=============

 Candidate invariant:
i != 0
i <= 8
i >= 5
j one of { 2, 3, 4 }
j != 0
i > j

=============Test Case=============

[   /** Test case automatically generated by YETI **/
   @Test public void test_00 throws Exception {
      BlockErrorPlotTwoShort.blockErrorPlot((int)7,(int)2);
      /**BUG FOUND: RUNTIME EXCEPTION**/
      /** java.lang.ArithmeticException: / by zero
                  at BlockErrorPlotTwoShort.blockErrorPlot(BlockErrorPlotTwoShort.java:7)**/

   }
]
```

Fig. 3. Graph and Invariants generated by ADFD+

are strategies []. that exploit the presence of block and strip failure-domain to get better results. Therefore identifying the presence of underlying failure-domains in production software can help in high quality of software testing. To answer the questions, we reviewed all the classes containing failure-domains manually, automatically and graphically.

4) *If the nature of identified failure-domains simple or complex?* An interesting point is to know what failure is responsible for a failure-domain and how difficult it is to identify that failure by manual testing. To answer this question, we studied the test logs and test output of the automated testing and the source code of the program manually to identify the cause and complexity of failures of failure-domains.
5) *If the invariants generated by Daikon correctly represent the failure domains?* Invariants generated by Daikon can identify the start and stop of the failure domain. To answer this question we compared the generated invariants with the source code and the failure-domain presented in graphical form.
6) *If the graph generated by ADFD correctly represent the pass and fail domains?* Both the ADFD and ADFD+ techniques generate graphs to represent failure-domains for simplicity. To answer the question we compared the generated graphs with the source code and the invariants generated by Daikon.
7) *If obtained results consistent with previous theoretical and practical results presented?* As per our knowledge, till now no specific study has been conducted to automatically identify the pass and fail domains however it has been claimed by some researchers [] that there exist more block and strip patterns then the point patterns.
8) *If the presence of a particular failure-domain can make it easy or hard to find using automated and manual techniques?* Failure-domain can reside in the form of point, block or strip shape in the input domain. To answer this question we analysed the source code of all the programs in which failure-domains were detected.

V. Experimental Setup

We tested all the 4500 classes included in the main jar files of each of the 106 open-source packages contained in Qualitas Corpus [12] using ADFD and ADFD+ technique. The Qualitas Corpus is selected for two reasons: (1) It is a database of open-source Java programs that spans across the whole set of Java applications. (2) It is specially built for empirical research which takes into account a large number of developmental models and programming styles. Each test on average took 40 seconds to complete. The YETI ran for 5 seconds while the remaining time is used for finding failure-domains, generating invariants and drawing graph. The machine took approximately 100 hours to process the experiments. Only the failing one and two dimensional methods with arguments (int, long, float, double and short) were taken in to consideration. This is because at the moment ADFD and ADFD+ are not capable of drawing/handling more than two dimensions. All experiments were conducted with a 64-bit Mac OS X Mountain lion version 10.8.5 running on 2.7 GHz Intel Core i7 with 16 GB (1600 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.7.0_45]. The ADFD and ADFD+ executable files are available at https://code.google.com/p/yeti-test/downloads/list/.

VI. Results

We found 57 faulty classes spread across 25 different packages. Results of the experiments are given in Table 1, 2, 3 and 4. All those failure-domains were declared as strip failure-domains in which 100 or more contagious failures were discovered. Accordingly, in 48 out of 57 classes strip failure-domain is detected as shown in Table I. The failure-domains in which 10 or more contagious failures were discovered. The failure-domains in which 5 or less contagious failures were discovered are termed as point failure-domains.There are 4 classes which contain point failure-domain as shown in

Table II. The 2 classes contain block failure domain as shown in Table III. The 2 classes contain two types of failure-domains i.e. AnnotationValue with both point and block failure-domain and Token with point and Strip failure-domain as shown in Table IV.

The source code of all the 57 classes were analysed manually.

VII. Experimental results

VIII. Threats to validity

As in the case of any experimental study, the results are considered conclusive only if the programs tested represent general behaviour. We have tried to minimize this threat by choosing all the classes from all the projects included in the purpose built repository of Qualitas Corpus.

YETI with ADFD and ADFD+ strategies selected is executed only for 5 seconds to find a failure in the SUT. Since both the test strategies are based on random+ strategies which have high precedence for boundary values therefore most of the failures detected are boundary failures. Despite the fact that the failure-domains remained the same for a specific failure detected, It is likely possible that increasing the test duration may lead to the discovery of new failures exhibiting different behaviour.

Another threat to validity may be related to hardware and software resources. For example the OutOfMemoryError occured at the value of 6980000 on the machine executing the test. On another machine the failure revealing value can increase or decrease depending on the hardware specifications and system load.

IX. Related Work

X. Conclusion

XI. Future Work

Acknowledgments. The authors are thankful to the Department of Computer Science, University of York for physical and financial support. Thanks are also extended to Prof. Richard Paige for his valuable guidance, help and generous support.

References

[1] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[2] T. Y. Chen, "Adaptive random testing," *Eighth International Conference on Qualify Software*, vol. 0, p. 443, 2008.

[3] B. Beizer, *Software testing techniques (2nd ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[4] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in. net with feedback-directed random testing," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 87–96.

[5] M. A. Ahmad and M. Oriol, "Automated discovery of failure domain," *Lecture Notes on Software Engineering*, vol. 02, no. 4, pp. 331–336, 2014.

[6] ——, "Automated discovery of failure domain," *Lecture Notes on Software Engineering*, vol. 03, no. 1, pp. 289–294, 2013.

[7] M. Oriol. (2011) York extensible testing infrastructure. Department of Computer Science, The University of York. [Online]. Available: http://www.yetitest.org/

[8] 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*, vol. 69, no. 1, pp. 35–45, 2007.

[9] D. Gilbert, "The jfreechart class library version 1.0. 9," 2008.

[10] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, ser. HICSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 261a–. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.462

[11] M. Oriol, "Random testing: Evaluation of a law describing the number of faults found," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, april 2012, pp. 201 –210.

[12] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

TABLE I

TABLE DEPICTING CLASSES WITH STRIP FAILURE-DOMAINS FOUND BY ADFD AND ADFD+ AND MANUAL TESTING

S#	Class	Method	ADFD+	ADFD	Manual
1	LeadPipeInputStream	LeadPipeInputStream(i)	I >= 2147483140 I <= 2147483647	I	I > 698000000
2	BitSet	BitSet.of(i,j)	I <= -1, I >= -18, J <= 7, J >= -12	I one of {-513, -1} J one of {-503, 507}	I <= -1 J != 0
3	ToolPallete	ToolPalette(i,j)	I <= -1, I >= -18	I one of { -515, -1} J one of {-509, 501}	I <= -1, J any value
4	IntMap	idMap(i)	I != 0, I <= -1, I >= -18	I one of {-1, -512}	I <= -1
5	ExpressionFactory	expressionOfType(i)	I <= 13, I >= -7	I one of {-497, 513}	I >= -2147483648 I <= 2147483647
6	ArrayStack	ArrayStack(i)	I >= 2147483636 I <= 2147483647	I one of {2147483142, 2147483647}	I > 698000000
7	BinaryHeap	BinaryHeap(i)	I <= -2147483637 I >= -2147483648	I one of {-2147483648, -2147483142}	I <= 0
8	BondedFifoBuffer	BoundedFifoBuffer(i)	I <= -2147483639 I >= -2147483648	I one of {-505, 0}	I <= 0
9	FastArrayList	FastArrayList(i)	I <= -2147483641 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
10	StaticBucketMap	StaticBucketMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
11	PriorityBuffer	PriorityBuffer(i)	I != 0, I <= -1, I >= -14	I one of {-2147483647, -2147483142}	I <= 0
12	GenericPermuting	permutation(i,j)	I <= 0, I >= -18	I one of { -498, 0} I one of {2, 512}	I <= 0, I >= 2 J != 0
13	LongArrayList	LongArrayList(i)	I <= -2147483640 I >= -2147483648	I one of {-510, -1}	I <= -1
14	OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	I <= -1, I >= -17	I one of {-514, -1}	I <= -1
15	ByteVector	ByteVector(i)	I <= -2147483639 I >= -2147483648	I one of {-2147483648, -2147483141}	I <= -1
16	ElementFactory	newConstantCollection(i)	I >= 2147483636 I <= 2147483647	I one of {2147483141, 2147483647}	I > 698000000
17	IntIntMap	IntIntMap(i)	I <= -2147483638 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
18	ObjectIntMap	ObjectIntMap(i)	I >= 2147483640 I <= 2147483647	I one of {2147483591, 2147483647}	I > 698000000
	IntObjectMap	IntObjectMap(i)	I <= -1, I >= -17	I <= -1, I >= -518	I <= -1
19	ArchiveUtils	padTo(i,j)	I >= 2147483641 I <= 2147483647	I one of {-497, 513} J one of {2147483591, 2147483647}	I any value J > 698000000
20	BloomFilter32bit	BloomFilter32bit(i,j)	I <= -1 I >= -18	I one of {-515, -1} J may be any value	I <-1 J <-1
21	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647} I >= -518	I > 698000000
22	ObjectCacheHashMap	ObjectCacheHashMap(i)	I <= -2147483641 I >= -2147483648	I one of {-512, 0}	I <= 0
23	ObjToIntMap	ObjToIntMap(i)	I <= -2147483636 I >= -2147483648	I one of {-2147483646, -2147483137}	I <= -1
24	PRTokeniser	isDelimiterWhitespace(i)	I <= -2 I >= -18	I one of {-509, -2} I one of {256, 501}	I <= -2 I >= 256
25	PdfAction	PdfAction(i)	I <= -2147483640 I >= -2147483648	I one of {-514, 0} I one of {6, 496}	I <= 0 I >= 6
26	PdfLiteral	PdfLiteral(i)	I <= -1, I >= -14	I one of {-511, -1}	I <= -1
27	PhysicalEnvironment	PhysicalEnvironment(i)	I <= -1, I >= -11	I one of {-2147483646, -2147483137}	I <= -1
28	IntegerArray	IntegerArray(i)	I >= 2147483636 I <= 2147483647	I one of {2147483587, 2147483647}	I > 698000000
29	AttributeMap	AttributeMap(i)	I <= -2147483639 I >= -2147483648	I one of {-514, 0}	I <= 0
30	ByteList	ByteList(i)	I <= -1, I >= -14	I one of {-513, -1}	I <= -1
31	WeakIdentityHashMap	WeakIdentityHashMap(i)	I >= 2147483636 I <= 2147483647	I one of {2147483140, 2147483647}	I >698000000
32	AmmoType	getMunitionsFor(i)	I <= -1 I >= -17	I one of {-514, -1} I one of {93, 496}	I <= -1 I >= 93
33	IntList	IntList(i,j)	I <= -1 I >= -15	I one of {-1, -509} j one of 0	I <= -1 j = 0
34	QMC	halton(i,j)	I <= -1, I >= -12 J <= -1, J >= -15	I <= -1, I >= -508 j <= 499, j >= -511	I <= -1 J any value
35	BenchmarkFramework	BenchmarkFramework(i,j)	I != 0, I <= -1, I >= -13	I one of {-501, -1}	I <= -1
36	IntArray	IntArray(i)	I <= -1, I >= -16	I one of {-2147483650, -2147483141}	I <= -1
37	TDoubleStack	TDoubleStack(i)	I <= -1, I >= -13	I one of {-511, -1}	I <= -1
38	TIntStack	TIntStack(i)	I <= -1, I >= -12	I one of {-2147483648, -2147483144}	I <= -1
39	TLongArrayList	TLongArrayList(i)	I <= -1, I >= -16	I one of {-2147483648, -2147483141}	I <= -1
40	AlgVector	AlgVector(i)	I <= -1, I >= -15	I one of {-511, -1}	I <= -1
41	BinarySparseInstance	BinarySparseInstance(i)	I <= -1, I >= -15	I one of {-506, -1}	I <= -1
42	SoftReferenceSymbolTable	SoftReferenceSymbolTable(i)	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
43	SymbolHash	SymbolHash(i)	I <= -1, I >= -16	I one of {-2147483648, -2147483592}	I <= -1
44	SynchronizedSymbolTable	SynchronizedSymbolTable(i)	I <= -2147483140 I >= -2147483648	I one of {-2147483648, -2147483592}	I <= -1
45	XMLChar	isSpace(i)	I != 0, I <= -1, I >= -12	I one of {-510, -1}	I <= -1
46	XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	I != 0, I <= -1, I >= -13	I one of {-2147483648, -2147483137}	I <= -1
47	XML11Char	isXML11NCNameStart(i)	I <= -1, I >= -16	I one of { -512, -1}	I <= -1
48	AttributeList	AttributeList(i)	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647}	I > 698000000

TABLE II
TABLE DEPICTING CLASSES WITH POINT FAILURE-DOMAINS FOUND BY ADFD AND ADFD+ AND MANUAL TESTING

S#	Class	Method	ADFD+	ADFD	Manual
1	Assert	assertEquals(i,j)	I != J	I != J	I != J
2	Board	getTypeName(i)	I <= -1 I >= -18	I >= -504, I <= -405, I >= -403 I <= -304, I >= -302, I <= -203 I >= -201, I <= -102, I >= -100 I <= -1	I <= -910, I >= -908, I <= -809, I >= -807, I <= -708, I >= -706, I <= -607, I >= -605, I <= -506, I >= -504, I <= -405, I >= -403, I <= -304, I >= -302, I <= -203, I >= -201, I <= -102, I >= -100, I <= -1
3	HTMLEntities	get(i)	I <=- 1 I >= -17	I >= -504, I <= -405, I >= -403 I <= -304, I >= -302, I <= -203 I >= -201, I <= -102, I >= -100 I <= -1	I <= -910, I >= -908, I >= -807, I <= -708, I >= -706, I <= -809, I <= -607, I >= -605, I <= -506, I >= -504, I <= -405, I >= -403, I <= -304, I >= -302, I <= -203, I >= -201, I <= -102, I >= -100, I <= -1
4	Assert	assertEquals(i,j)	I <= 0, I >= 20 J <= 18, j >= -2	I one of {-2147483648, -2147483142} J one of {-501, 509}	I != J

TABLE III
TABLE DEPICTING CLASSES WITH BLOCK FAILURE-DOMAINS FOUND BY ADFD AND ADFD+ AND MANUAL TESTING

S#	Class	Method	ADFD+	ADFD	Manual
1	AnnotationValue	whatKindIsThis(i)	I <= 85, I >= 92, I >= 98 I = 100, I >= 102, I <= 104	I ¡= 63, I = {65, 69, 71, 72} I >= 75, I ¡= 82, I >= 84 I <= 89, I >= 92, I ¡= 98 I = 100, I >= 102, I <= 114 I >= 116	I <= 63, I = 65, 69, 71, 72 I >= 75, I <= 82, I >= 84 I <= 89, I >= 92, I <= 98 I = 100, I >= 102, I <= 114 I >= 116
2	Token	typeToName(i)	I <= -2147483641 I >= -2147483648	I one of {-510, -2} I = {73, 156} I one of {162, 500}	I <= -2, I = 73, 156, I >= 162

TABLE IV
TABLE DEPICTING CLASSES WITH MIX FAILURE-DOMAINS FOUND BY ADFD AND ADFD+ AND MANUAL TESTING

S#	Class	Method	ADFD	ADFD+	Manual
1	ClassLoaderResolver	getCallerClass(i)	I >= 2, I <= 18	I >= 500, I <= -2 I >= 2, I <= 505	I <= -2, I >= 2
2	Variant	getVariantLength(i)	I >=0, I <= 12	I >= 0, I <= 14, I >= 16 I <= 31, I >= 64, I <= 72	I >= 0, I <= 14, I >= 16 I <= 31, I >= 64, I <= 72