ValBench: Benchmarking Exact Value Analysis

Marc Miltenberger
marc.miltenberger@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Steven Arzt
steven.arzt@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Abstract

Value analysis is an important building block in static program analysis. While several approaches have been proposed, evaluating and comparing them is not trivial. Up to this day, a reliable and large benchmark specifically for value analysis is missing. Such a suite must not only provide test cases, but also a ground truth with the correct values to be found.

In this paper, we propose ValBench, an extensible value benchmark suite consisting of 372 test cases for Java analysis and 59 test cases for Android analysis tools. Furthermore, we present an evaluation framework that automatically generates a ground truth for these test cases, identifies their respective challenges for program analysis and orchestrates the execution and result collection on the various value analysis tools. We further present an evaluation of 7 existing value analysis tools on ValBench and highlight the challenges faced by these tools as an empirical overview over the state of the art in value analysis.

CCS Concepts: • **Security and privacy → Software security engineering**.

Keywords: string analysis, value analysis, benchmarks

ACM Reference Format:
Marc Miltenberger and Steven Arzt. 2024. ValBench: Benchmarking Exact Value Analysis. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652588.3663322

1 Introduction

Value analysis is a vital building block in static program analysis. When building callgraphs, for example, the target class and method names are required to uniquely identify the callee [1, 9]. Without this information, the callgraph either misses edges or must resort to coarse over-approximation. Android inter-component communication poses similar challenges. Components inside an app may invoke other components in the same or a different app by sending an `Intent`, which contains data for resolving the target component as well as the data to pass to the recipient. To precisely model such calls, the callgraph analysis must infer various fields of the intent [12, 13]. Intents are particularly challenging, because they form a composite value analysis problem. The analysis must ensure not to mix of the values of different fields across objects if multiple intents are in scope.

Value analysis is also essential for vulnerability detection. Many security properties rely on checking API parameter values. A vulnerable program may, for example, pass hard-coded cryptographic keys to a crypto API, may send sensitive information to an unprotected *http* endpoint rather than using TLS, or may pass an outdated algorithm name such as `DES` to an encryption function. In total, the completeness and correctness of the value analysis is of utmost importance.

Most value analysis research has focused on string analysis. Some approaches return concrete strings, while others such as *JSA* [4], *COAL* [12] or *Violist* [8] return regular expressions. The latter approach may support values that depend on external inputs that are unavailable to static analysis e.g., user input or results retrieved from a remote server via a network connection. As we show, in many cases returning regular expressions instead of concrete values can, however, be a consequence of imprecisions inherent to the algorithm. For example, when encountering string decryption methods in obfuscated apps, these approaches may return a generic `.*`, which offers no information.

Despite the existing value analysis research, measuring the accuracy of a value analysis remains non-trivial. It requires a comprehensive dataset with a known ground truth. While real-world applications can provide complex examples, they usually lack the ground truth. Reverse-engineering these examples comes at a prohibitive effort on a large scale. Real-world examples are therefore useful for analyzing the time and memory consumption of the value analysis, but do not help for a precision assessment.

For ensuring a ground truth, papers such as *JSA* propose test suites. These suites, however, are usually limited in size and scope and focus on the features in the focus of the respective tool. When new approaches are published, they extend the existing suites for their own evaluation, leading to a multitude of different tool-specific test suites that do not allow for a broad evaluation of the state of the art in value analysis.

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663322

We consider this a serious drawback hindering progress in value analysis research.

In this paper we present the, to the best of our knowledge, first comprehense test suite for value analysis. Our test suite comprises 431 test cases written in Java based on our test suite framework. To ensure that the ground truth is correct, our framework runs the test case on the JVM and records the computed value. This step generates the ground truth, thereby avoiding the manual effort and the risk for human error. The framework then executes all registered analysis clients and compares the result returned by each analysis client with the ground truth values. As a result, the framework collects detailed evaluation data on a per-analysis and per-test case level as well as summarized recall and precision.

In total, we present the following original contributions in this paper.

- A set of 431 test cases for exact value analysis, which we call ValBench. They can be used to evaluate and compare different exact value approaches on a ground truth. ValBench can be used on Android and JVM analyses tools.
- An extensible evaluation framework, which allows an easy comparison between different value analysis approaches. It supports running various approaches, parsing their results and comparing them. Initially, it supports 7 existing approaches. New approaches can easily be implemented.
- An evaluation of 7 existing value analysis frameworks on the ValBench test suite.

The remainder of this paper is structured as follows: First, we present related work in Section 2. In Section 3 we lay out the construction and evaluation approach of the test suite. In Section 4, we show our evaluation results. Lastly, Section 5 concludes the paper.

2 Related Work

Testing of static analyzers has been explored by prior work. Casso et al [3] proposed a an approach for the Ciao programming language, which generates assertions based on the static analysis results. Using dynamic analysis, these assertions are then used to check whether the generated assertions hold and the static analysis is correct. The approach proposed by Klinger et al [7] generates benchmarks using a given set of seed programs in order to test multiple analyzers. It automatically places assertions regarding numerical properties of variables in the code. In order to check whether the assertion can be violated (true positive) or not (true negative), they use the 'bugs-as-deviant-behavior' strategy proposed by Engler et al [5], where the result of the majority of analyzers is assumed to be correct, while the result determined by the minority is assumed to be incorrect.

ValBench can be used as a tool to test Java and Android value analysis tools. In contrast to both approaches, ValBench offers a ground-truth. Various approaches for extracting values from Java programs or Android apps have been proposed. We used the keywords "Java value analysis" and "Java string analysis" on IEEE Xplore and ACM Digital Library. We tested approaches with an directly available tool or where a tool could be provided upon request by the authors.

JSA [4] focuses on extracting strings. It constructs automata which model the language of all possible strings as elements and string operations as automata transitions. *JSA* returns these automata as regular expressions. *COAL* [12] also yields regular expressions, but is based on static constant propagation and a domain-specific language to model complex data structures such as Intents. *Violist* [8] applies a region-based analysis to identify code segments and loops which compute the value of interest and uses an interpreter to compute values. *Violist* can either perform loop unrolling or output regular expressions similar to *JSA*. The loop unrolling technique does not respect loop conditions. Instead, the loop is unrolled exactly a certain number of times. *BlueSeal* [15] is based on backward slicing and abstract interpretation, but the paper does not give many details on the algorithm.

Other approaches return concrete strings rather than regular expressions. *Harvester* [14] is a hybrid (static & dynamic) approach. It performs a static backward slice and then runs the slices dynamically to obtain the values. *Harvester* uses the term "logging point" to refer to the tuple consisting of the variable of interest with the location of interest. We adopted this notion in this paper. *ValDroid* [10] is similar to *Harvester*, but instead of a dynamic phase, it performs a static simulation. *StringHound* [6] tries to deobfuscate strings. It first uses classification to find string decryption methods. These methods get extracted by static backward slicing and run on the JVM. As the JVM does not have adequate models for Android system APIs, it cannot perform value extraction when Android APIs are involved. Although it was designed primarily for string decryption, it can be easily modified to run a generic value analysis by changing the classification logic.

For evaluation, *JSA*'s test cases are closest to ValBench. *JSA* features a set of 328 test cases and is used by related work[4, 8] to test regular expression based value approaches. Each test cases features three regular expressions as a ground truth. Each regular expression defines a different degree of precision. Listing 1 shows the `AnnotatedField` *JSA* test case. According to the testcase, the most precise regular expression, which models the `field` value is *C*|null*, albeit the exact value *CCCCCCCCCC* is deterministically calculated. While *C*|null* is technically correct, it is imprecise. Also note that the resulting value cannot possibly be **`null`**. We therefore argue that this test-case is insuitable for evaluating precise value approaches.

Furthermore, the *JSA* test suite references *JSA*-specific resolvers in 35 test cases, i.e., the test suite is not independent from the approach under test. Lastly, we observed several mistakes in the ground truth of the *JSA* test suite. The *ArrayOfObjectsToString* test case, for example, expects the `toString()` of an array to equal `"[Ljava.lang.Object; @3e25a5"`. Note that the part after the @ is the hash code, which is implementation-dependent and in most implementations dependent on the memory address of the array.

We conclude that the *JSA* test suite is unsuitable for precisely evaluating arbitrary value analysis tools. Aside from the *JSA* test suite, the DaCapo [2] benchmark suite provides programs for evaluating Java-based static analyses. DaCapo, however, is not specific to value analysis and does not provide a corresponding ground truth.

```
1 private @Type("C*|null") String field;
2 public void foo() {
3    field = "";
4    for (int i=0; i<10; i++) field += "C";
5    StringTest.analyze(field, "C*|null", "C*|
         null", "C*|null");
6 }
```

Listing 1. *JSA* test case `AnnotatedField`

3 Approach

The test framework consists of several steps. It first takes the test case source with optional annotations. If no ground truth is available, it compiles the test case and runs it on the JVM to obtain the ground truth. Afterwards, the framework creates JAR and APK files for each test file that are then presented to the value analysis approaches. Finally, the values returned by the approach are compared to the ground truth for the evaluation report and statistics (overall recall and precision). We next explain these steps in more detail.

3.1 Defining Test Cases

Each entry point method needs to be annotated with the `@ValueComputationTestCase` annotation. Note that the test case is not restricted to the entrypoint, i.e. the entry point may potentially call other methods. There are two different types of test case, return-based and explicit test cases. For return-based test cases, the value analysis must reconstruct the value returned by the test case. Explicit test cases are **`void`** methods and thus have no return value. Instead, they call the method `explicitLoggingPoint(Object)` with a specific variable in the test case, similar to the `analyze` method in *JSA* test case shown in Listing 1. For these test cases, the value analysis goal is to reconstruct the first parameter at the callsite of `explicitLoggingPoint`. Technically, both options are equivalent, because the test framework automatically inserts calls to `explicitLoggingPoint` before every **`return`** statement in each test case. These logging points are used as an input for the value analysis tools.

By default, the ground truth for each test case is obtained automatically by running the test cases on the JVM. The test framework records the values (method return value and explicitly requested values) and places them in a JSON file. Adding new test cases therefore requires only minimal effort. Alternatively, test case authors may use the `expectedValues` attribute of the `ValueComputationTestCase` annotation to manually provide the expected results. In this case, the test case will not be run on the JVM.

Android-specific test cases that are only valid in the context of an app and that cannot be run on a JVM must always be annotated by hand. We decided against integrating an Android runner due to the complexity of integrating an Android build chain and emulator-based execution environment. Such a feature would have hampered the platform independence of our test framework.

Further, each test case is annotated with a machine readable list of challenges. Some test cases may, for example, require the string analysis approach to model certain API calls such as `Integer.valueOf`. Other test cases need field, array handling or loop support. This information is kept for statistics. Each test case is still presented to each tool. If a tool does not support loops, for example, the analyst can cluster the failed test cases around these properties.

If no challenge annotation is given for a test case, ValBench determines the list of challenges per test case automatically using static analysis. It records API calls and notes whether the test case uses fields, arrays, loops etc.

Although ValBench test expect the tools to return strings for easy comparison, many test cases require the tools to handle complex objects. Correctly handling the Android `Intent` class, for example, is a composite value problem including key/value maps that is represented in the string representation of the `Intent`. Note that even though the return value are strings, we consider ValBench a *value* test suite, since the proper handling of complex values is necessary in order to solve many test cases correctly.

3.2 Building JARs and APKs

As most value analysis tools work on binaries rather than source code, our test framework automatically builds JAR files for Java-based test cases and APK files for both Android- and Java-based test cases. Note that Java-based test cases can run on Android as well. Whether a tool is capable of processing JARs, APKs or both is defined by a tool-specific execution adapter class.

In our evaluation we found that some tools run into timeouts when presented with the whole test suite as a single JAR or APK file. To avoid this issue, we have implemented a shrinker utility based on Soot, which creates JARs and APKs with a single test case. Dependencies, e.g., libraries compiled into the APK, are reduced to the minimum set of

methods and classes necessary. This shrinker is best-effort. For JAR files, the framework automatically re-runs the created shrinked JARs and compares the results to the ground truth to avoid introducing bugs into the testcases. Testing the APK versions would also be possible, but would require a phone or emulator during build. We therefore omitted this feature to keep the setup simple. Test cases can be annotated to disable shrinking.

3.3 Running the Approaches

Each approach connects to the test framework via a tool-specific adapter. The adapter is responsible for running the tool with the correct configuration and for collecting the tool's outputs. Adapters may invoke tools via the command-line or may also use APIs offered by the tools.

3.4 Evaluating Results

The test framework compares the values returned by the value analysis approaches with the ground truth using string equality. ValBench focuses on evaluating approaches that return *concrete* values. If a tool instead returns a regular expression, our evaluation framework only checks the concrete parts. An expression `A|B`, for example, is split into the concrete values `A` and `B` which are compared against the ground truth. Recall that the ground truth is always a concrete value. Hence, any wider regular expression is necessarily an over-approximation.

Instead of a single concrete value, test cases may return different values depending on conditions. In such cases, value analyses must return *all* values to achieve 100 % recall in our evaluation framework. Note that running the test case on the JVM may only return a subset of these values if some conditions cannot be fulfilled during execution. In that case, analysts can augment the test case with additional values by hand or ensure that conditions can be forced from the outside, e.g., via environment variables.

4 Evaluation

The ValBench test suite features 372 JVM and 59 Android test cases. In this section, we run the framework against 7 existing value analysis approaches on this test suite.

RQ1: How does ValBench compare to *JSA* test cases?

The *JSA* test suite features only 41 tests (12.5 %) involving arrays and 15 tests (4.57 %) with loops. In contrast, ValBench features 123 array tests (24.83 %) and 100 (23.20 %) with loops. Furthermore, ValBench contains 35 (8.12 %) testcases involving reflection and 29 (6.73 %) involving streams, while *JSA* testcases feature neither.

As noted in Section 3, ValBench automatically infers a list of challenges for each test case. Table 1 and Table 2 show the Top 10 challenges for ValBench and *JSA* test cases. ValBench features a larger number of test case involving

Table 1. Top 10 challenges for ValBench test cases

Challenge	Number of test cases
`StringBuilder.toString`	214 (49.65 %)
`StringBuilder.append`	204 (47.33 %)
`StringBuilder.<init>`	201 (46.64 %)
Primitive type - int	196 (45.48 %)
Interprocedural	134 (31.09 %)
Fields	125 (29.00 %)
Arrays	123 (28.54 %)
Arithmetic calculations	107 (24.83 %)
Loops	100 (23.20 %)
`StringBuilder.append`	91 (21.11 %)

Table 2. Top 10 challenges for *JSA* test cases

Challenge	Number of test cases
Interprocedural	112 (34.15 %)
`StringBuffer.toString`	101 (30.79 %)
If	100 (30.49 %)
`StringBuilder.toString`	97 (29.57 %)
`StringBuilder.<init>`	87 (26.52 %)
`StringBuffer.append`	83 (25.30 %)
Fields	78 (23.78 %)
`String.valueOf`	72 (21.95 %)
`StringBuffer.<init>`	71 (21.65 %)
`StringBuilder.append`	70 (21.34 %)

Fields (29 % in ValBench vs 23% in *JSA*), Arrays (28.54 % vs 12.50 %), Arithmetic Calculations (24.83 % vs 5.79 %) and Loops (23.20 % vs 4.57 %).

Furthermore, ValBench contains 59 Android-specific test cases. These test cases require value analysis tools to support composite data structures such as Intents and Pairs.

RQ2: How effective are value analysis tools on ValBench?

We adapted *JSA*, *Violist*, *BlueSeal* and *COAL* slightly to use the newest Soot release, as the outdated versions shipped with the tools were not capable of analyzing modern Java class files. We modified StringHound to classify testcase methods as string decryption methods in order to use its slicing algorithm. For the evaluation of *Harvester*, *COAL*, *BlueSeal* and *ValDroid* we used the Android version of our test cases. We supply all changed tools (except *BlueSeal* and *ValDroid*, which we cannot redistribute due to missing the license) in our data package.

Note that ValBench supports multiple values per test case, either by using different values in **`return`** statements (e.g. in `testIf` test case) or calling `explicitLoggingPoint` multiple times (e.g. in a loop as in `testExplicit2`). We merely require that the set of results must be finite. For these cases, all expected results must be specified manually in the annotation. In the evaluation, we treated all expected values as if they were individual test cases. For example,

Table 3. Precision & Recall on the ValBench test suite in %. Tools with an asterisk are exact value approaches.

	JVM cases		Android cases	
Approach	**Prec.**	**Rec.**	**Prec.**	**Rec.**
BlueSeal	2.39	1.95	-	0
COAL	10.34	6.57	0	0
*Harvester**	67.67	54.50	92.86	86.67
JSA	0.02	18.73		
*StringHound**	72.53	16.06		
*ValDroid**	91.81	90.02	94.83	91.67
Violist	0.39	9.00		

`testInputStream3` has 4 different expected values. Harvester reports two values correctly and no spurious value, resulting in a precision of 100 % and a recall of 50 %. As stated in Section 3.4, for tools that only output a singular regular expression, we split `A|B` into two results `A` and `B`.

COAL returned .* for 243 JVM test cases, while *JSA* delivered .* for 117 JVM test cases. The other tools did not deliver such results. We counted .* as false negatives, as they do not contain any information. Note that this initial iteration of test cases only features test cases which have a low and fixed number of specific expected string values. Since each expected value is concrete, a precise regular expression tool ideally returns the most precise regular expression, which is a combination of all expected values using the | operator.

In order to calculate the precision, we calculated $t_p/(t_p + f_p)$, while recall was calculated via $t_p/(t_p + f_n)$. To aggregate precision and recall over multiple test cases, we calculated the sum of t_p, f_n and f_p of all aggregated cases.

Table 3 shows the precision and recall of each tool on the JVM and Android test cases. For the Android test cases, we only list tools which support the Android platform. The regular-expression based approaches *BlueSeal*, *COAL*, *JSA* and *Violist* have lower precision than approaches for concrete values such as *Harvester*, *StringHound* and *ValDroid*. Most tools offer a relatively low (< 50 %) recall on the 372 JVM test cases. Note that *BlueSeal* did not return any value for any of the Android test cases. *COAL* returned wildcards for all Android test cases. *Harvester* and *ValDroid* have a high precision and recall on the 59 Android-specific test cases.

RQ3: What insights can we gain by evaluating value tools on ValBench?

Table 4 shows the precision and recall of all test tools on all test cases. We next discuss how the tools perform on various challenges to showcase room for further research in the field.

```
1        String[] x = new String[2];
2        x[1] = "Right";
3        x[0] = "Wrong";
4        return x[1];
```

Listing 2. ValBench test case `testArraySensitivity`

Listing 2 shows an array sensitivity testcase. *JSA* gives the results `"Wrong"`, `"Right"` and **`null`**, despite that they "give a precise treatment of `String`, `StringBuffer`, and multidimensional arrays of strings" [4]. *COAL* returns `"Wrong"` and `"Right"`. *StringHound* has not given any result for that particular test case, but has correct results for other Array test cases. The other tools give the correct results.

```
1        Fields f = new Fields(), f2 = new
             Fields();
2        f.a = "A";
3        f2.a = "Wrong";
4        return f.a;
```

Listing 3. ValBench test case `objectSensitivityTest`

JSA fails the object sensitivity test in Listing 3, giving `"Wrong"`, `"A"` and **`null`** as result. *COAL* and *Violist* yield the strings `"Wrong"` and `"A"`. *BlueSeal* returns the empty String. The other tools return the correct string. *StringHound* has not given any result for that particular test case, but has correct results for other object sensitivity test cases (e.g., `complicatedFieldTest`). *Harvester* and *ValDroid* return the correct result.

```
1        String start = "Start-";
2        StringBuilder b = null;
3        for (int i = 0; i < 5; i++) {
4                if (i == 0) b = new
                     StringBuilder(start);
5                b.append(i);
6        }
7        b.append("-afterLoop");
8        return b.toString();
```

Listing 4. ValBench test case `simpleLoopTest`

In the loop test case in Listing 4, *StringHound*, *Harvester* and *ValDroid* report the correct string `Start-01234-afterLoop`, *Violist* reports `nullafterLoop` and *JSA* overapproximates with `Start-(.*)`. *BlueSeal* only reports `Start-`. *COAL* returns four different regular expressions: `Start-(.*)-afterLoop`, `NULL-CONSTANT-afterLoop`, `(.*)(.*)-afterLoop` and `NULL-CONSTANT(.*)-afterLoop`.

Challenge: Streams. *JSA* overapproximates on the stream related test cases `testInputStream3` and `testInputStream4`, resulting in 4 true positives for each of both test cases, but 253,948 false positives for each possible ASCII character combination which can be returned by the input stream in this test case. Only *Harvester* and *ValDroid* return the correct values for these cases.

Table 4. Results on ValBench in precision %/recall %. Tools marked by * output exact values. # denotes the number of tests.

Test suite name	#	*BlueSeal*	*COAL*	*Harvester**	*JSA*	*SH**	*ValDroid**	*Violist*
AdvancedString	3	0/0	-/0	20/33.33	-/0	-/0	100/100	-/0
Alias	7	0/0	0/0	85.71/85.71	0/0	100/57.14	100/71.43	0/0
Android	59	-/0	0/0	92.86/86.67	50/11.67	0/0	94.83/91.67	0/0
Annotation	8	0/0	0/0	100/75	0/0	85.71/75	100/100	-/0
ApacheHex	4	0/0	-/0	100/25	-/0	100/25	-/0	-/0
Arithmetic	11	-/0	0/0	100/100	-/0	-/0	100/100	0/0
Array	8	0/0	0/0	100/100	25/62.50	100/25	100/100	80/50
ArrayCopyOf	7	0/0	-/0	85.71/85.71	50/14.29	100/57.14	100/100	-/0
BasicString	10	9.09/9.09	14.29/9.09	70/63.64	100/72.73	100/9.09	100/100	0/0
ByteBuffer	6	0/0	0/0	16.67/16.67	-/0	80/66.67	100/100	-/0
Callee	2	50/50	0/0	50/50	100/50	-/0	100/100	-/0
Caller	2	0/0	0/0	0/0	-/0	100/50	100/100	0/0
Calls	1	0/0	-/0	100/100	-/0	-/0	100/100	-/0
Character	2	0/0	-/0	100/100	50/100	100/50	100/100	-/0
Class	21	0/0	0/0	68.42/61.90	0.78/4.76	100/4.76	85.71/85.71	0/0
Collection	20	0/0	0/0	60/60	-/0	-/0	85/85	0/0
ComplexArray	1	-/0	-/0	100/100	-/0	100/100	100/100	-/0
Composite	3	33.33/16.67	50/100	100/50	50/100	-/0	60/100	50/100
ContextSensitive	4	0/0	-/0	0/0	40/33.33	-/0	100/100	-/0
Crypto-JavaImpl	10	0/0	-/0	37.50/30	0/0	-/0	80/40	-/0
CryptoAPI	1	0/0	0/0	100/100	-/0	-/0	100/100	-/0
Dictionary	4	0/0	-/0	100/75	-/0	100/25	100/100	-/0
DynamicInvoke	1	-/0	-/0	-/0	-/0	-/0	-/0	-/0
Enum	11	0/0	0/0	88.89/72.73	-/0	100/9.09	100/90.91	0/0
EnumSet	12	0/0	-/0	83.33/41.67	50/25	100/25	100/100	-/0
Field	17	0/0	0/0	64.71/61.11	30/16.67	66.67/44.44	90.91/55.56	33.33/11.11
FieldDepth	1	16.67/100	0/0	100/100	-/0	-/0	100/100	0/0
File	5	-/0	0/0	80/80	-/0	-/0	100/100	0/0
FlowSensitivity	1	0/0	0/0	100/100	100/100	-/0	100/100	100/100
HashCodeEquals	6	0/0	-/0	16.67/16.67	6.25/16.67	-/0	33.33/33.33	-/0
IOCommons	8	0/0	-/0	0/0	-/0	-/0	100/100	-/0
If	5	50/14.29	42.86/42.86	100/57.14	44.44/57.14	-/0	58.33/100	36.36/57.14
InheritanceLibrary	1	0/0	-/0	0/0	-/0	-/0	100/100	-/0
InstanceOf	13	0/0	-/0	100/92.31	-/0	100/76.92	100/100	-/0
JSON	5	0/0	-/0	100/100	-/0	-/0	100/100	-/0
Loop	19	0/0	0/0	94.74/94.74	0/0	100/10.53	94.74/94.74	0/0
Map	6	0/0	0/0	100/83.33	-/0	0/0	83.33/83.33	0/0
Math	5	0/0	-/0	100/100	100/80	100/100	100/100	-/0
Merge	1	0/0	0/0	0/0	60/100	-/0	60/100	-/0
Misc	4	0/0	0/0	100/100	0/0	-/0	100/100	0/0
MultiArray	3	0/0	-/0	-/0	-/0	-/0	0/0	-/0
MutableObjects	1	0/0	-/0	100/100	-/0	-/0	100/100	-/0
PatternTest	2	0/0	0/0	100/100	0/0	-/0	100/100	-/0
Polymorphic	10	0/0	-/0	0/0	100/10	-/0	100/90	-/0
Properties	2	0/0	0/0	0/0	-/0	-/0	100/100	-/0
PullParserTest	3	0/0	-/0	100/100	-/0	0/0	100/100	-/0
Recursive	1	0/0	-/0	0/0	-/0	-/0	0/0	-/0
Reflection	23	5.88/8.33	0/0	54.55/50	-/0	0/0	91.67/91.67	5.88/4.17
ReflectionArray	1	0/0	0/0	0/0	-/0	0/0	100/100	-/0
SimpleString	3	0/0	0/0	66.67/66.67	50/33.33	-/0	100/100	0/0
Simplest	1	0/0	-/0	0/0	-/0	-/0	100/100	-/0
Stream	25	0/0	0/0	22.22/19.35	0.00/25.81	57.14/12.90	100/100	-/0
String	6	0/0	0/0	66.67/66.67	0/0	50/16.67	100/100	-/0
StringUtils	11	0/0	-/0	-/0	50/9.09	100/36.36	100/100	-/0
Switch	10	25/4.55	36.17/77.27	100/45.45	61.54/72.73	-/0	100/100	0.20/86.36
TestBitwise	11	-/0	0/0	100/100	-/0	100/9.09	100/100	-/0
Thread	1	-/0	-/0	0/0	-/0	-/0	100/100	-/0
Type	1	-/0	0/0	100/100	-/0	-/0	100/100	-/0
UUID	1	-/0	-/0	100/100	-/0	-/0	100/100	-/0

Out of 29 streams test cases, *ValDroid* returns the correct value for 27, *StringHound* returns the correct value for 4 and *Harvester* for 5 test cases.

Challenge: Complex Crypto Algorithms. ValBench features 10 complex user-code implementations of cryptographic algorithms, which require correct array and loop semantics in order to compute the correct values. Only *Harvester* and *ValDroid* could yield the correct values for some crypto test cases. Furthermore, *ValDroid* yields the correct value for the Blake2s algorithm.

Challenge: Recursion. All approaches except *Harvester* do not seem to support recursion and thus fail all recursive test cases. *Harvester* failed the Fibonacci test case and reports a wrong value (0), but yields the correct value in the `findRecursion` test case.

5 Conclusion

In this paper, we have presented ValBench, a benchmark suite for evaluating value analysis approaches, and a testing framework for running value analysis tools against the benchmark suite. As future work, we plan to further extend ValBench and to improve the automation for the Android-based test cases. Furthermore, we plan to add test cases that use partly user-input to compute values.

6 Data Availability

We have made ValBench fully open source [1] under a permissive license, allowing researchers to use and extend the test suite. We also include all approaches except BlueSeal and Harvester and ValDroid [10] due to their software license. However, we include the precomputed results of all approaches. The results can be replicated when obtaining the approaches from their respective authors. While the GitHub page may feature a newer version of ValBench, we made the original replication package available under the DOI 10.1145/3580436 [11].

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D Ernst. 2015. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 669–679.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Ignacio Casso, José Francisco Morales, Pedro López-García, and Manuel V. Hermenegildo. 2020. Testing Your (Static Analysis) Truths. In *International Workshop/Symposium on Logic-based Program Synthesis and Transformation*. https://api.semanticscholar.org/CorpusID:221317992

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 1–18. http://dl.acm.org/citation.cfm?id=1760267.1760269

[5] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001). https://api.semanticscholar.org/CorpusID:1377888

[6] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Oct 2020). https://doi.org/10.1145/3320269.3384745

[7] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2018. Differentially testing soundness and precision of program analyzers. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018). https://api.semanticscholar.org/CorpusID:54474665

[8] Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond. 2015. String Analysis for Java and Android Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 661–672. https://doi.org/10.1145/2786805.2786879

[9] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 756–761.

[10] Marc Miltenberger and Steven Arzt. 2024. Precisely Extracting Complex Variable Values from Android Apps. *ACM Trans. Softw. Eng. Methodol.* (feb 2024). https://doi.org/10.1145/3649591

[11] Marc Miltenberger and Steven Arzt. 2024. Valbench Artifact. https://doi.org/10.1145/3580436

[12] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 77–88. https://doi.org/10.1109/ICSE.2015.30

[13] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. 2016. Composite Constant Propagation and its Application to Android Program Analysis. *IEEE Transactions on Software Engineering* 42, 11 (2016), 999–1014. https://doi.org/10.1109/TSE.2016.2550446

[14] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[15] J. D. Vecchio, F. Shen, K. M. Yee, B. Wang, S. Y. Ko, and L. Ziarek. 2015. String Analysis of Android Applications (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 680–685. https://doi.org/10.1109/ASE.2015.20

[1] https://github.com/Fraunhofer-SIT/ValBench