Using Targeted Symbolic Execution for Reducing False-Positives in Dataflow Analysis

Steven Arzt[1] Siegfried Rasthofer[1] Robert Hahn[1] Eric Bodden[1,2]

[1]Center for Advanced Security Research Darmstadt (CASED)
Technische Universität Darmstadt, Germany
[2]Fraunhofer SIT, Darmstadt, Germany
{siegfried.rasthofer, steven.arzt, robert.hahn, eric.bodden}@cased.de

Abstract

Static data flow analysis is an indispensable tool for finding potentially malicious data leaks in software programs. Programs, nowadays often consisting of millions of lines of code, have grown much too large to allow for a complete manual inspection. Nevertheless, security experts need to judge whether an application is trustworthy or not, developers need to find bugs, and quality experts need to assess the maturity of software products. Thus, analysts take advantage of automated data flow analysis tools to find candidates for suspicious leaks which are then further investigated.

While much progress has been made in the area with a broad variety of static data flow analysis tools proposed in academia and being offered commercially, the number of false alarms raised by these tools is still a concern. Many of the false alarms are reported because the analysis tool detects data flows along paths which are not realizable at runtime, e.g., due to contradictory conditions on the path. Still, every single report is a potential issue and must be reviewed by an expert which is labor-intensive and costly. In this work, we therefore propose TASMAN, a post-analysis based on symbolic execution that removes such false data leaks along unrealizable paths from the result set. Thus, it greatly improves the usefulness of the result presented to the human analyst.

In our experiments on DroidBench examples, TASMAN reduces the number of false positives by about 80% without pruning any true positives. Additionally, TASMAN also identified false positives in real-world examples which we confirmed by hand. With an average execution time of 5.4 seconds per alleged leak to be checked on large real-world applications, TASMAN is fast enough for practical use.

General Terms Languages, Verification

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.4.6 [*Security and Protection*]: Information flow controls

Keywords Data Flow Analysis, False Positives, Precision, Symbolic Execution, TASMAN

SOAP'15, June 14, 2015, Portland, OR, USA.

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

1. Introduction

Static data flow analysis tools have been adopted for a broad variety of tasks in software development, quality engineering, testing, and security assessment. Most modern software systems have grown much too large for a human expert to fully manually analyze them and detect data flows that may hint at bugs, security vulnerabilities, or poor software quality. On the other hand, such analyses become more and more important with software being increasingly used in safety-critical environments or to deal with highly privacy-sensitive information such credit card data, or personal health records.

This issue has given rise to a market of static analysis tools which automatically perform checks on software, reducing the required labour of a human expert. The human expert now only needs to assess the results produced by the tool. While this is already a large improvement over manual analysis and there has been much work on the quality of static analysis tools, today's tools still report a number of false positives. For the human analyst, this has the consequence of wasted effort spent on understanding the program until he finally arrives at the decision to ignore a certain tool report at hand. For businesses, this wasted expert labour can quickly add up to a considerable amount of money being spent in vain.

Many of todays's static analysis tools which are available on the market or which have been proposed in academia are based on data flow analysis. In particular, analyzing Android applications is a very attractive field for security researchers since it is known that apps handle and may also leak personal data [1]. We have looked into the main sources of false positives produced by static data flow analysis tools for Android that arise when applying them to large real-world applications. Our approach focuses on Android applications, but can also be applied to Java applications. In most cases, the path on which the tool reports a data flow through the program is unrealizable, i.e., can never be taken at runtime. Most prominently, those paths contain contradictory conditions such as *the operating system version is greater than 5* and at the same time *the operating system version is smaller than 3*. Many other false positives are similar, likely caused by the fact that the developer missed to remove some checks during the testing phase. Many of these conditions are however not local and thus not directly eminent from the code. Therefore, even tools that employ a simple dead-code elimination as a pre-processing still report these false positives.

We therefore propose TASMAN (**Ta**rgeted **Sym**bolic Execution for **An**droid), a fully automated post-analysis step based on symbolic execution. TASMAN runs after the main data flow analysis is completed and prunes data leaks with infeasible paths from the result. Only those results that TASMAN could not disprove

```
1  void onCreate() {
2    show(null);
3  }
4
5  void show(AdDisplayListener listener) {
6    this.callback = listener;
7    boolean bool1;
8    if (this.callback != null)
9      bool1 = this.ad.show();
10   else
11     bool1 = false;
12   ...
13   String secret = getSecret();
14   if (bool1)
15     leak(secret);
16 }
```

Listing 1. False Positive Example

are displayed to the human analyst, greatly reducing the effort that remains to be spent on confirming the findings.

This however gives rise to an additional challenge. Most static data flow analysis tools only report those fractions of the program on the data flow path which directly manipulate the data being tracked, but not all of its control-flow dependencies. Such a reduced flow however is insufficient for collecting the full set of conditions that must be met for the path to be feasible at runtime. Even if a fraction of the path does not manipulate any data, it might abort the control flow at runtime. Therefore, our post-analysis must reconstruct the missing parts of the data flow path from the tool's output.

In summary, this paper presents the following original contributions:

- an approach for collecting the relevant conditions that can influence the validity of a data flow path,
- an analysis based on symbolic execution which scans for contradictions on a data flow path,
- an evaluation of the effectiveness of our approach based on 80 micro benchmarking applications and 10 real-world apps from the Google Play Store, and
- an extension of DroidBench containing our 80 micro benchmarking applications for testing static data flow analyses based on symbolic execution

The remainder of this paper is structured as follows. Section 2 presents a motivating example which will then be used to explain how the framework works in Section 3. In Section 4, we evaluate the performance and precision of our approach. Section 5 discusses limitations, while Section 6 presents related work. We conclude in Section 7.

2. Motivating Example

Listing 1 shows a shortened real-world example taken from the *Love calculator* app [1], slightly modified for readability reasons. The method `show` contains a potential data leak (line 13 and 15) reported by the state-of-the-art static data flow tracker FlowDroid [1]. In this example, assume that `onCreate()` is the only caller of `show()`. It is then easy to follow that there is no data leak since the variable *bool1* in line 14 can never hold a value of `true`.

FlowDroid nevertheless reports the leak as the tracking is agnostic to conditionals which is a common design choice for taint tracking tools. Taint tracking is a *meet-over-all-paths* problem, and one would thus need to track all values that could possibly reach a conditional which is infeasible. TASMAN on the other hand only works on the result paths reported by the taint tracker and is thus far more efficient. In the remainder of the paper, we discuss the problem in more detail and propose a solution that reduces the number false positives produced by static data flow tracking tools.

[1] com.mobilplug.lovetest

3. Framework

TASMAN checks whether a taint propagation path reported by a data flow analysis tool is feasible. A necessary precondition for taint feasibility is the existence of at least one control-flow path containing all statements on the taint propagation path. Furthermore, if there are conditions along this control-flow path, they must not be contradictory. Thus, to check for infeasible paths, one must first match the taint proapagation path on possible control flow paths and then collect all the conditions along this path.

However, the taint propagation paths reported by most static taint tracking tools only contain those statements that actively propagate or transform taint information, i.e., assignments as well as method calls and returns. In the example in Listing 1, this means that the taint propagation path only consists of statements 13 and 15:

```
String secret = getSecret();
leak(secret);
```

To conclude that this path is infeasible, one however also needs the conditional in line 14. Deciding the possible outcomes of the conditional in turn requires knowledge about the possible values of the `callback` field, which yet in turn requires a back-tracking into the callers. Only then, we can conclude that the field must always be null, the condition can never hold, and the leak can never happen.

Therefore, TASMAN must extend the propagation paths reported by the taint tracking tool with all statements that can influence control flow reachability plus all assignments to variables or fields on which the conditionals transitively depend. This collection must happen inter-procedurally. All statements on the extended path are then transformed into constraints. Only if the resulting constraint set is satisfiable, the leak is possible at runtime. To check whether the resulting constraint set is satisfiable or not, an off-the-shelf SMT solver is used.

In this section, we will first explain the algorithm used by TASMAN to construct the constraint set in the intra-procedural case. In Section 3.1, we show how the code from the motivating example can be turned into an unsatifiable constraint set which can then be used to disprove the corresponding data leak. In Section 3.2, we explain in detail how inter-procedural control flows are handled, and Section 3.4 refers to checking exceptional control flows. Section 3.3 tackles how TASMAN processes loops and recursions, while Section 3.5 explains how specific data types are handled. Our implementation is discusssed in Section 3.7.

3.1 Basic Constraint Generation

TASMAN starts its backwards analysis at the sink statement (line 15) which gets associated with an empty constraint set. Empty constraint sets are trivially satisfiable as they do not impose any restrictions on the values of any variable in the program. TASMAN then performs a backwards propagation along the interprocedural control flow graph. While going through the graph, TASMAN checks for conditions. If a predecessor of the current statement is a conditional, it is directly translated into a new constraint. Variables are treated as free symbols to be refined later on. We will

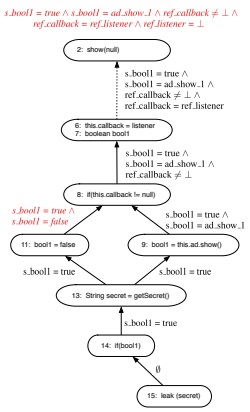

Figure 1. Constraint Propagation Along the Inverse CFG

now show how the propagation is done on the motivating example from Listing 1. A graphical representation is shown in Figure 1. The formulas on the edges of the control-flow graph represent the constraint set aggregated so far. Formulas printed in red and italic font are infeasible.

For line 14, TASMAN generates the constraint set `s_bool1 = true`. The conditional must evaluate to `true`, otherwise the sink statement would not be reached. However, since the possible values for `bool1` are not yet known to the backwards analayis, a new, free symbol `s_bool1` is created for this variable. For a constraint solver, this means that the symbol can take any value possible according to its type. At the moment, the constraint set would thus be trivially satisfiable. While iterating further back in the control flow, more and more constraints on the variable will however be generated, further restricting its possible values. In other words, the more information the backwards analysis picks up for a variable, the more precise becomes the constraint set on the possible values. Only if all constraints obtained in the end can be satisfied at the same time, there is a set of values for which the taint propagation path is feasible.

The next line in the backwards iteration would normally be line 13. This line is however neither a conditional nor does it refine any of the variables for which we already have symbols in the constraint set. Thus, no additional information can be gained from this line and TASMAN skips it.

Once the analysis has arrived at line 12, there are multiple possible predecessors. Depending on the outcome of the conditional at line 8, the predecessor of line 12 is either line 11 or line 9. Semantically, the data flow is feasible if at least one of the possible control flow path paths containing the propagation path reported by a static data flow analysis tool is feasible. At this point, TASMAN thus splits the analysis and proceeds with two distinct (but initially equal) constraint sets for two possible control flow paths that could lead to the leak in question.

Path 1: Assume that statement 11 is the predecessor. The value of variable `bool1` gets changed which refines the set of possible values for the corresponding symbol `s_bool1`. This extends the constraint set to `s_bool1 = true` $\wedge$ `s_bool1 = false`. Obviously, this constraint is unsatisfiable (printed in red in the figure) and processing can stop at this point. No further predecessors need to be taken into account.

Path 2: Assume that statment 9 is the predecessor. This statement also assigns variable `bool1` and thus also refines symbol `s_bool1`. Unlike for the other possible control-flow path, we however get a satisfiable constraint set: `s_bool1 = true` $\wedge$ `s_bool1 = ad_show_1`. The new, free symbol `ad_show_1` models the return value of the call to `ad.show()` and will once again be refined during the further backwards processing. For the moment, we will ignore the concrete refinement of the call for the sake of simplicity and leave this new symbol free. Method handling will be discussed in Section 3.2.

The predecessor of line 9 is the condition in line 8. Since we leave the *then* branch of a conditional, the condition must hold or otherwise the control flow would not have reached this branch. Therefore, our constraint set gets extended one more time: `s_bool1 = true` $\wedge$ `s_bool1 = ad_show_1` $\wedge$ `ref_callback` $\neq \bot$.

Note that object references such as `this.callback` are modeled by assigning unique numeric identifiers to allocation sites. When an object is first referenced, a new, unbound `ref` symbol is introduced. If the backwards constraint collection then reaches the allocation site of the corresponding object, TASMAN generates a constraint maping the symbol to this allocation site's unique number. This technique reduces object references to numbers and allows for a simple handling of Java object identity comparisons: For a check `a == b`, TASMAN generates a constraint which compares the unique identifiers of the corresponding allocation sites. The special pseudo allocation-site $\bot$ models null references. The remainder of the method can again be skipped as it does not provide any further information on conditionals or existing symbols.

At the beginning of the method, the backwards analysis must continue into the callers. If there are multiple callers, the control flow path must be split into multiple possibilities with one constraint set each just like in the case of the `if` statement. More information on the interprocedural part will be given in Section 3.2. In this example, there is only one caller and the parameter can trivially be mapped to the respective call site argument (line 2), giving the final constraint set: `s_bool1 = true` $\wedge$ `s_bool1 = ad_show_1` $\wedge$ `ref_callback` $\neq \bot \wedge$ `ref_callback` $= \bot$. This constraint set is clearly unsatisfiable.

In summary, TASMAN has now explored all possible control flow paths that can lead to the alleged leak. It has collected all constraints along these paths and proven that they are unsatisfiable. As a consequence, this leak can only be a false positive.

3.2 Interprocedural Constraint Collection

When the backwards constraint collection reaches a method call as in Line 9, it must continue with the constraint collection inside the callee. More specifically, the return value of the callee yields a new constraint on the symbol corresponding to left side of the assignment in the caller.

In general, there can be multiple callees. TASMAN therefore splits the control flow path into the various possibilities with one constraint set each. This splitting is equivalent to the intraprocedural case of the `if` statement in line 8 of the motivating example: In both cases, the predecessor of the current statement is not uniquely identifiable and all possibilities must be explored with their own copy of the constraint set built up so far.

Note that TASMAN never joins constraint sets, even if two possible control flows do join again at some point. As an example, if a call site has multiple possible callees, the two copies of the constraint set will continue as independent possibilities even after the backwards analysis has returned to the same return site for both possible callees. While this can lead to more constraint sets than necessary, it makes every single set simpler and easier to decide for the solver. To keep runtime low, TASMAN however performs early checking to not collect any further constraints if the set is already infeasible.

3.3 Loops and Recursion

Some propagation paths are along control flow paths containing loops or recursive method calls. Whether these paths are feasible

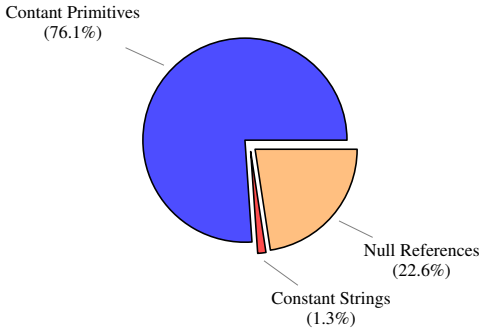

Figure 2. Usage of Constants on Path Conditions in Android Applications

or not often depends on the number of iterations that have passed inside the loop, e.g., if a certain value is only assigned after the 10th iteration. Precisely modeling such constructs is hard and subject to ongoing research work [8, 13, 14]. TASMAN unrolls loops - either to the number of iterations after which no new constraints are generated or to a pre-defined maximum iteration count, whichever is lower.

3.4 Exception Handling

The motivating example in Listing 1 induces one more constraint which we omitted in the description up to now for increased clarity: In Line 8, `this` may not be `null`. In general, operations that can throw exceptions according to the Java / Dalvik language specifications may make the program depart from normal control flow at runtime. This can make sink statements found by a static data flow analysis tool unreachable on the path detected by the data flow tracker. TASMAN generates conditions for field accesses and virtual invocations that require the respective base objects not to be null. Other similar cases such as checks on array indices (must be in range) or divisors in arithmetic expressions (must not be zero) can easily be added to the tool to increase the detection rate even further.

3.5 Special Data Types

Some data types available in Java-based programming languages pose challenges to constraint solvers. Strings, for instance, have no corresponding type in many SMT solvers and thus need to be emulated by TASMAN. Fully modeling strings as sequences of Unicode characters can however lead to very large constraint sets and is thus infeasible in practice. In a pre-analysis on the top 100 application in the Google Play Store, we found this is however usually not even necessary: string operations are mostly equality checks in real-world Android applications. Even those only made up 1.3% of all data types of variables involved in conditions on taint propagation paths as shown in Figure 2. Note that data types which are used in the application as such, but not for variables in conditions along taint propagation paths, need not be modeled.

This enables us to, instead of modelling the Java String API, treat strings as immutable atomic objects that are represented by unique symbols. Equality comparisons are then transformed in constraints matching two symbols. If an app does perform a string operation such as a substring, TASMAN picks a new unbound symbol, modelling that any value is possible. In the future, a more precise model may be possible when novel approaches in constraint solving [3, 9] become available in production-grade solvers.

According to our preliminary study, the majority of all data types to be considered were numeric types (integer, long, etc.), and object types. The latter were however only used for null checks in almost all cases which is trivial to model in the constraint set using the special $\perp$ symbol as shown in Section 3.1.

The SMT library interface we use (see Section 3.7) contains an `Array` type which is however closer to a key-value map than to a Java-style array. Though arrays and lists were not shown as prevalent in our pre-analysis, TASMAN uses this SMTLib feature to map Java arrays to SMT Lib maps using numeric keys. Java-style mutable lists, maps, and sets are currently treated as unbound symbols as their complex runtime semantics cannot easily be expressed.

3.6 Static Single Assingment Form

If the code is not in static single assignment (SSA) form and a variable gets overwritten, this may lead to contradictory conditions if handled trivially. Assume two assignments `a = 3; a = a + 5`. If these are translated to a constraint set `s_a = 3` $\wedge$ `s_a = s_a + 5`, this set would be unsatisifiable, though the code corresponds to a perfectly valid control flow path.

To avoid this problem, TASMAN introduces a new symbol for every consecutive assignment of the same variable. This is similar to converting the code to SSA form before collecting constraints. In the example, the new constraint set would be `s_a0 = 3` $\wedge$ `s1_a = s_a0 + 5` which is satisfiable.

3.7 Implementation

We implemented our approach on top of FlowDroid [1] which is a highly precise data flow tracker with support for Android applications as well as normal Java programs. Furthermore, FlowDroid allows easy access to the actual taint propagation paths, though in a condensed form as described above. Since FlowDroid does not take conditionals into account[2], contradictory conditions are a source of precision by design. TASMAN helps solve this issue.

FlowDroid is in turn based on Soot [10] which gives TASMAN access to a call graph and unit graphs for the individual methods. Additionally, Soot already provides many useful analyses that could be re-used such as checking for definitions of certain variables.

For checking the satisfiability of the generated constraint set, an off-the-shelf SMT solver is used. TASMAN is based on SMT Lib v2.0 [2] to allow for easily exchanging the concrete solver. By default, the Z3 solver [4] created by Microsoft Research is used.

4. Evaluation

For TASMAN to be useful, it must identify a substantial number of false positives in the FlowDroid output for a common set of applications, it must not flag any actual data leaks as false positives, and it must finish its processing in an adequate timeframe. In this section, we therefore describe how we assessed the precision and performance of TASMAN both on artificial micro-benchmark challenges and on real-world Android applications taken from the Google Play Store.

4.1 Test Suite

An independent researcher contributed a test suite with 80 test cases containing challenges for static data flow tools based on symbolic execution. We will integrate these test cases into the DroidBench open-source project [1]. These tests are grouped into various categories such as array handling, casts, static and dynamic field use, inter-procedural data flows, lists and collections, loops and recursion, exceptions, operators, and strings. Table 1 shows the performance of TASMAN on the test suite. The first column lists the groups of test cases in the benchmark suite. For every

[2] In this work, we assume the implicit flow tracking option is disabled which is the default.

Table 1. Symbolic Benchmark Suite

Test-case group	False Positives	True Positives
Arrays	2/3	3/3
Casts	3/3	3/3
Collections	1/1	1/1
Exceptions	2/2	-
Fields and Objects	15/16	16/16
General Java	2/2	2/2
Interprocedural Data Flow	2/3	3/3
Library Handling	1/2	2/2
Loops and Recursion	4/4	4/4
Operators	2/3	3/3
Strings	1/4	4/4
Sum	35/43 (81%)	41/41 (100%)

test group, there are test cases with real leaks and ones without real leaks. We limited our evaluation to those test cases for which FlowDroid reported results as our goal was to assess TASMAN as a post-processing step on the FlowDroid results.

TASMAN was able to erase 81% of all false positives reported by FlowDroid and did not remove any true positive. This means that applying TASMAN as a post-processing step on the FlowDroid results is safe due to conservative over-approximation in cases that cannot be decided by symbolic execution.

4.2 Real-World Android Applications

In this section, we show how TASMAN performs on 10 randomly picked real-world Android applications taken from the Google Play Store. Table 2 shows the package names of the respective applications (column 1) together with the total number of leaks reported by FlowDroid (column 2), and the number of leaks from that set identified as false positives by TASMAN (column 3). We verified all of these alleged false positives by hand. This manual verification is the main reason why we had to limit this evaluation to a small set of applications as verifying static analysis results on real-world application without any access to the original source code is non-trivial and time-consuming.

All false positives identified by TASMAN were actual false positives, i.e., TASMAN did not remove any actual leaks from the result set. From our experiments, we can thus conclude that TASMAN is a safe post-processing that significantly reduces the number of false-positives left for analysis by a human expert without affecting the recall of FlowDroid.

The main reason for the false positives identified by TASMAN and confirmend by hand was dead code in the application. In some cases, the methods containing the leaks were not even called, in others, the leaks were guarded by conditionals that can never evaluate to `true`, for instance due to debug checks.

4.3 Performance

Column *FlowDroid(s)* in table 2 shows the performance of FlowDroid and TASMAN on the ten real-world examples on which we hand-verified the reported false positives. While the post-processing by TASMAN takes considerable time, it is still in the order of minutes. With an average of about 8 minutes per app in total, an automated post-processing using symbolic execution is still useful when it reduces the number of leaks that need to be verified by an expensive human expert.

Furthermore, much of the computation time is due to the high number of leaks in large applications. Per leak, TASMAN takes only about 5.4 seconds. Per identified false positive, TASMAN takes about 4.5 minutes on average.

5. Limitations

Our current approach is only applicable to conditions on values that can statically be derived from the program code. If a value is for instance read from the environment (e.g., from a file on disk), TASMAN assumes that the respective variable can take any value. Furthermore, our approach does not support string operations such as concatenation or substring. According to our pre-analysis (see Figure 2), this is however not an issue in practice. Comparisons on strings are scarce on data flow propagation paths, and even if they exist, they only compare constant values. As future work, we will however lift this limitation by combining TASMAN with HARVESTER [11] which is an approach for extracting complex runtime values from Android applications.

The combination of FlowDroid with TASMAN approach is not sound. For every source-to-sink connection it finds, FlowDroid only reports one arbitrarily chosen *witness* as a path, regardless of how many paths between the respective source and the respective sink exist in the program being analyzed. TASMAN collects the conditions on this witness path and checks whether they are contradictory. Thus, if TASMAN finds a contradiction and concludes that the respective path is no realizable at runtime, this only disproves the witness reported by FlowDroid and not the source-to-sink connection as such. In theory, there could be a different witness for the same source-to-sink connection which is indeed realizable. To avoid this issue, one could configure FlowDroid to report all possible witnesses to its findings. This can however not only severely impact performance, but also lead to an exponential increase in reports. Though artificial examples that demonstrate this problem can be created, witnesses are usually equal with respect to realizability in practice. Thus, TASMAN can limit itself to the one witness FlowDroid reports in its default configuration as we have shown in our evaluation.

6. Related Work

Static information flow analysis has been an active research topic for many years and numerous solutions for dealing with false positives and for increasing the precision of the analysis tools have been proposed.

Hammer et al. [5] propose the inference of *path conditions* for Java programs. Such conditions which can then be processed by a constraint solver to automatically derive input values which trigger the illicit paths. If no such inputs can be derived, the path is infeasible and the leak is a false positive.

Previously, Snelting [15] has already shown for procedural languages how path conditions can be used to increase the accuracy of a static program slices. Extracting and simplifying the conditions shows under which circumstances a certain leak can happen.

Robschink et. al. [12] generate path conditions for large procedural programs. These constraints are represented as BDDs for improved scalability. Similarly to Snelting's work, Robschink's conditions are simplified to yield formulae representing necessary conditions for a given information flow. Inputs satisfying these formulae are *witnesses* for the flow.

Taghdiri et. al. [16] extend these approaches with a CEGAR process to further erase false witnesses by executing them and iteratively refining the path conditions with those whitnesses that did not actually yield a data flow at runtime.

Jeon et. al. [6] present a simplified language for Dalvik executables which can more easily be used for symbolic execution than the original bytecode. Secondly, they demonstrate that their tool SymDroid is able to discover the path conditions under which the contact database is accessed in Android apps.

Another approach to eliminate infeasible paths was presented by Jhala et. al. [7] who propose slicing the program and eliminating

Table 2. Evaluation on Real-World Android Applications with Potential Leaks Identified by FlowDroid, False Positives Found by TASMAN, and Runtime Performance. All Identified False Positives Were Verified by Hand.

Test-case group	Leaks	FPs	FlowDroid(s)	TASMAN (s)	Time/Leak(s)	Time/FP(s)
com.buttons.dynamicButtonsfullPro	51	1	82.27	82.49	1.58	82.49
laser.pointer.laserpointer	120	7	166.75	2,453.67	20.44	350.52
com.devuni.flashlight	63	1	107.20	379.35	6.02	379.35
com.CrazyRobot.BatteryBooster	106	1	135.60	179.32	1.69	179.32
com.mobilplug.lovetest	57	1	119.82	71.73	1.25	71.73
goldenshorestechnologies.brightestflashlight.free	33	1	50.54	313.29	9.49	313.29
com.mattia.videos.manager	102	1	160.11	232.04	2.27	232.04
com.surpax.ledflashlight.panel	135	1	572.63	1,005.08	7.44	1,005.08
com.reviloapps.ChistesyFrasesGraciosas	21	1	76.43	40.28	1.91	40.28
love.bigcamerabuttonlite	50	2	124.44	97.50	1.95	48.75
Sum (Leaks) / Average (Times)	739	17	159.58	485.48	5.40	270.29

all statements which do not influence control flow reachability. The resulting minimal program induces necessary conditions which can then be checked.

ALETHEIA [17] applies statistical learning based on user feedback. The user can flag a small subset of the issues found by a data flow analysis tool as false warnings and ALETHEIA will filter out similar allegedly false warnings using the machine-learned model.

7. Conclusion

In this work, we presented TASMAN, a post-processing step for static data flow analysis tools which helps reduce the number of false alarms raised by these tools. Our experiments have shown that TASMAN removes up to 81% of all false positives while not removing any true positives from the list of results. Therefore, TASMAN is safe to apply before investigating the remaining leaks by hand.

As future work, we plan to integrate TASMAN directly into FlowDroid so that infeasible taint propagation paths can already be detected and pruned during the taint propagation and no further effort needs to be spent on extending them any longer. We hope that this will significantly reduce the runtime in comparison to the current post-processing step.

Acknowledgements This work was supported by the BMBF within EC SPRIDE, and by the Hessian LOEWE excellence initiative within CASED.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.

[2] C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0, 2010. *Available at www. SMT-LIB. org*.

[3] N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An smt-lib format for sequences and regular expressions. In *Strings*, page 24, 2012.

[4] L. de Moura and N. Bjrner. Z3: An efficient smt solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78799-0. . URL `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[5] C. Hammer, R. Schaade, and G. Snelting. Static path conditions for java. In *Proceedings of the 3rd Workshop on Programming Languages and Analysis for Security*, pages 55–66. ACM, June 2008. .

[6] J. Jeon, K. K. Micinski, and J. S. Foster. Symdroid: Symbolic execution for dalvik bytecode, 2012. http: //www.cs.umd.edu/ jfoster/papers/symdroid.pdf.

[7] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 38–47, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. . URL `http://doi.acm.org/10.1145/1065010.1065016`.

[8] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[9] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

[10] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[11] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. Technical Report TUD-CS-2015-0031, EC SPRIDE, Feb. 2015.

[12] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *24th International Conference of Software Engineering (ICSE)*, pages 19–25, Orlando, Florida, USA, May 2002. ACM. .

[13] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[15] G. Snelting. Combining slicing and constraint solving for validation of measurement software. In *Static Analysis*, pages 332–348. Springer-Verlag London, UK, Sept. 1996. .

[16] M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *7th International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 65–79, September 2010.

[17] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 762–774, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. . URL `http://doi.acm.org/10.1145/2660267.2660339`.