ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis

Xueling Zhang, Xiaoyin Wang, Rocky Slavin, Jianwei Niu

The University of Texas at San Antonio

Abstract

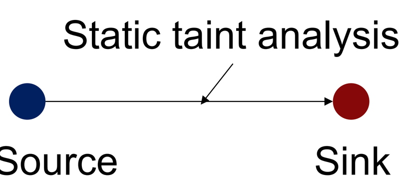
Static taint analyses are widely-applied techniques to detect taint flows in software systems. Although they are theoretically conservative and designed to detect all possible taint flows, static taint analyses almost always exhibit false negatives due to a variety of implementation limitations. Dynamic programming language features, inaccessible code, and the usage of multiple programming languages in a software project are some of the major causes. To alleviate this problem, we developed a novel approach, DySTA, which uses dynamic taint analysis results as additional sources for static taint analysis. However, naïvely adding sources causes static analysis to lose context sensitivity and thus produce false positives. Thus, we developed a hybrid context matching algorithm and a corresponding tool, ConDySTA, to preserve context sensitivity in DySTA. We applied REPRODROID, a comprehensive benchmarking framework for Android analysis tools, to evaluate ConDySTA. The results show that across 28 apps (1) ConDySTA was able to detect 12 out of 28 taint flows which were not detected by any of the six state-of-the-art static taint analyses considered in REPRODROID, and (2) ConDySTA reported no false positives, whereas nine were reported by DySTA alone. We further applied ConDySTA and FLOWDROID to 100 top Android apps from Google Play, and ConDySTA was able to detect 39 additional taint flows (besides 281 taint flows found by FLOWDROID) while preserving the context sensitivity of FLOWDROID.

Taint Anlalysis

Taint analysis can detect taint flows in software programs and has been widely used in privacy leak detection. Static taint analyses propagate taints based on an overestimation of all possible program paths leading to the detection of all possible taint flows with no false negatives but some false positives due to infeasible paths.

```
1  public class LeakageApp extends Activity{
2  private User user = null;
3  protected void onRestart(){
4    EditText usernameText =
       (EditText)findViewById(R.id.username);
5    EditText passwordText =
       (EditText)findViewById(R.id.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18                  String.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
        user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault();
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

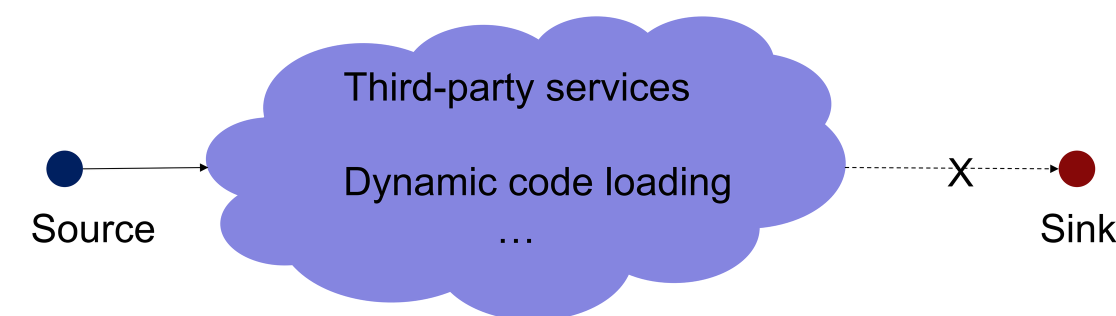
Source: Read Password



Sink: Send password via SMS

False negative

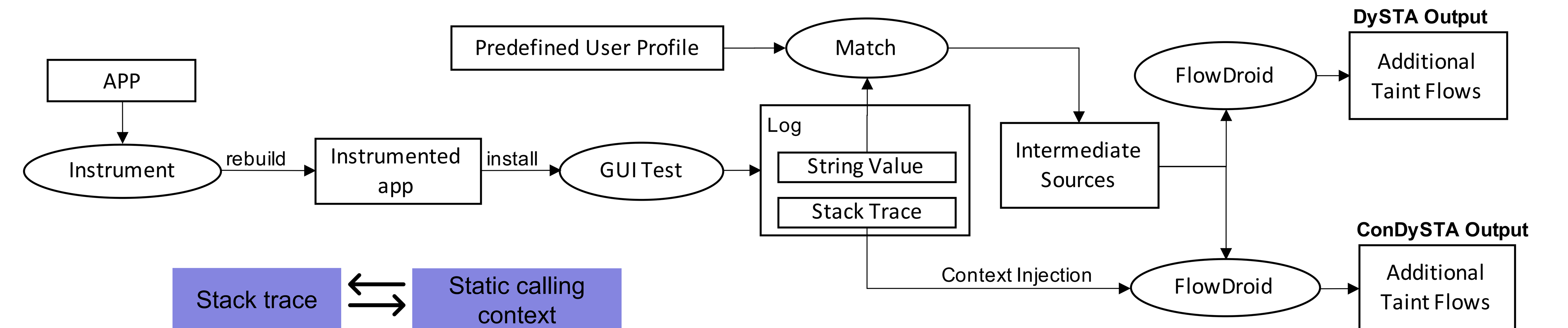
Despite the theoretical soundness of static taint analyses, various practical complexities often lead to false negatives in real-world scenarios. As an example, our evaluation shows that while FlowDroid, the state-of-the-art static taint analysis tool for Android apps, finds 281 taint flows in 100 top Android apps but misses at least 19 taint flows which are confirmed by dynamic taint analysis. Earlier studies also show the existence of false negatives in static taint analyses. A later study performed an evaluation of six state-of-the-art static taint analysis tools for Android and also reported many common false negatives not detected by any of the evaluated tools. Such false negatives may result in undetected vulnerabilities, privacy leaks, malicious apps, etc. The reason behind these false negatives can often be attributed to dynamic programming language features such as reflection calls in Java, dynamically loaded or generated code, external code execution through database servers and network servers, and multi-language code (e.g., native code and shell scripts). We refer to such features as blockers as they block the static taint analyses from tracing taint flows.



Contribution

- We demonstrate that dynamic taint analysis results can be used as a supplement to static taint analysis to reduce false negatives in practice.
- We developed a novel approach, ConDySTA, to preserve the context sensitivity of static taint analysis when supplemented by dynamic taint analysis.
- We performed evaluations using the ReproDroid benchmark and 100 top Android apps from Google Play demonstrating that ConDySTA can reduce many false negatives reported by state-of-the-art taint analysis tools and largely reduce false positives from our baseline solution

Approach



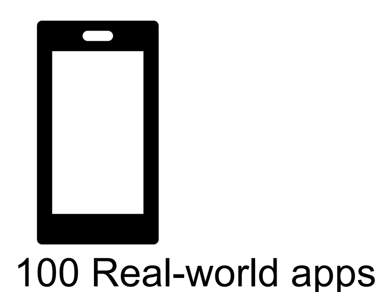
We propose an approach that uses the results of dynamic taint analysis as additional sources to supplement static taint analysis as a means to reduce false negatives. The base version of our approach is referred as DySTA (Dynamic Supplement of static Taint Analysis). DySTA first runs static taint analysis and dynamic taint analysis with the same set of initial sources, respectively. Once DySTA observes a variable holding a tainted value in the dynamic taint analysis that is not observed as tainted by the static taint analysis, the variable will be considered a new source (referred to as an intermediate source to be differentiated from the original sources). For the set of all intermediate sources, DySTA runs the static taint analysis again to find additional taint flows. Unlike static analysis, dynamic analysis is performed at run time, so it is less affected by blockers and is able to trace taint flows through dynamically loaded or generated code. However, the basic design of DySTA has an important limitation. Since it simply concatenates static and dynamic taint flows without any constraints, the context sensitivity of the original static taint analysis will be lost. Therefore DySTA alone will lead to additional false positives besides those in the original static taint analysis for cases where blockers were analyzed. To overcome this, we further propose hybrid context matching in which the context of dynamic taint flows is injected into the intermediate sources. DySTA is then augmented so the subsequent static taint analysis considers only taint flows that matching the injected context. By incorporating context matching, we implemented ConDySTA (Context-aware DySTA) as an extension of FlowDroid, a state-of-the-art static taint analysis tool for Android apps.

Evaluation

12 taint flows

False Positive:
- DySTA: 9
- ConDySTA: 0

ID	Feature	Apk	Source & Sink
DroidBenchExtend			
124	ImplicitFlows	ImplicitFlow1	android.telephony.TelephonyManager.getDeviceId() android.util.Log.i(java.lang.String,java.lang.String)
191	Native	SinkInNativeLibCode	android.telephony.TelephonyManager.getDeviceId() mod.ndk.ActMain.cFuncSendData(java.lang.String)
192	Native	SourceInNativeCode	mod.ndk.ActMain.cFuncGetIMEI(android.content.Context) android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
203	Reflection_ICC	OnlyIntent	android.telephony.TelephonyManager.getDeviceId() android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
206	Reflection_ICC	OnlyTelephony	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[]) android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
207	Reflection_ICC	OnlyTelephony_Dynamic	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[]) android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
208	Reflection_ICC	OnlyTelephony_Reverse	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[]) android.telephony.SmsManager. sendTextMessage(java.lang.String, ...)
209	Reflection_ICC	OnlyTelephony_Substring	java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[]) android.telephony.SmsManager.sendTextMessage(java.lang.String, ...)
ICCBench			
24	IccTargetFinding	icc_dynregister1	android.telephony.TelephonyManager.getDeviceId() android.util.Log.d(java.lang.String,java.lang.String)
25	IccTargetFinding	icc_dynregister2	android.telephony.TelephonyManager.getDeviceId() android.util.Log.d(java.lang.String,java.lang.String)
27	IccTargetFinding	icc_explicit1	android.telephony.TelephonyManager.getDeviceId() android.util.Log.d(java.lang.String,java.lang.String)
32	IccTargetFinding	icc_implicit_mix1	android.telephony.TelephonyManager.getDeviceId() android.util.Log.d(java.lang.String,java.lang.String)



39 taint flows from 12 apps

App Package Name	Size (KLOC)	FlowDroid	DySTA	ConDySTA (Dynamic)	ExecTime(s) DySTA+ ConDySTA	ExecTime(s) FlowDroid
com.amazon.mShop.android.shopping	10881	1	25	2(0)	25	257
com.dianxinos.dxbs	3034	1	77	15(6)	4266	1162
com.disney.WMWLite	1489	2	11	2(2)	357	131
com.forthblue.pool	1778	3	22	2(0)	1630	270
com.gameloft.android.ANMP.GloftDMHM	2540	20	3	3(0)	29	18
com.mxtech.videoplayer.ad	4044	3	4	1(1)	574	27
com.pinterest	5534	0	2	4(4)	95	138
com.sgiggle.production	6015	0	1	1(0)	44	32
com.tubitv	7660	0	5	3(2)	38	273
com.waze	2996	1	1	1(0)	16	115
org.mozilla.firefox	2155	24	74	4(4)	18	1265
paint.by.number.pixel.art.coloring.drawing.puzzle	4795	0	14	1(0)	23	64
...	...	...	...		...	
Total	N/A	281	1068	39(19)	N/A	