Android Provenance: Debugging Device Disorders

Nathaniel Husted  Sharjeel Qureshi*, Dawood Tariq, Ashish Gehani
Indiana University  SRI International
Android OS

- Smartphone Operating System by Google
- 49.4% Market share in the US [1]
- 75% Market share worldwide [2]
- Over 700,000 apps at the end of 2012 [3]
- Developers average ~$2,700 per app, per month [4]
Traditional Development and Debugging

• Eclipse – Open Source IDE
  • Android Developer Tools and Debugger

• Android SDK (“Java” Apps and the Framework API)
  • Emulator
  • Automatic UI Interactions (Monkey, MonkeyRunner, uiautomator – Not related)
  • System tracing utilities
  • Static code analysis

• Android NDK (Native libraries)
  • A compiler, a linker, and a non-standard C library.
Challenges to Traditional Debugging on Android

• Lots of Inter-Process Communication (IPC)
  • Both within the app and through the framework
• Lots of Asynchronous Functions and Threading
• Background processes, foreground processes in the same App
Complex Device Disorders

• Performance issues
• Bug’s disappearing with the debugger (Heisenbugs)
• Battery life issues
• How can we solve these issues when all our tools focus on a single application?
How can we debug complex disorders?

Provenance!
Provenance for Troubleshooting

• Chiarini’s Provenance for System Troubleshooting [5].
• Focuses on *nix based server environments
• Goals were to improve a system’s administrators mental model of the system.
Our Contribution: Provenance for Debugging

• A manner to gather low level system provenance on Android with minimal performance impact
• A way of quickly querying our provenance output
Provenance for Debugging Requirements

• Low level Information Source: Linux Audit [4]
• A Data Provenance System: SPADE [5]
• A Provenance Querying Method
  • Built in to SPADE
Provenance System: SPADE for Android
Querying the Android SPADE database
Provenance Debugging Methodology

• Installed Android ports of SPADE and Audit on a Samsung Galaxy Nexus phone running a custom Android (AOSP) OS.
  • https://github.com/nwhusted/AuditdAndroid
  • http://spade.csl.sri.com/SPADE/Downloads.html
• Configured Audit to ignore information regarding SPADE and itself
• We ran our example applications and manually interfaced with them
• Final output was analyzed on a desktop machine
  • Output graphs were ~900 vertices and ~5000 edges
• Output was filtered with SPADE’s Interactive Query Client
result = getEdges(location:*wake *lock, null, operation:write)
Provenance for Solving UI Latency

• Enthusiast replaces blocking calls to /dev/random
• Potential solution: Call /dev/urandom instead.
• It’s easy to identify if a call is being made to /dev/random instead of /dev/urandom:
  • result = getEdges(null, location:/dev/*random, operation:read)
Provenance Has Little Performance Impact

<table>
<thead>
<tr>
<th>Configuration</th>
<th>AnTuTu Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory Default</td>
<td>7890</td>
</tr>
<tr>
<td>Audit Only</td>
<td>7770</td>
</tr>
<tr>
<td>Audit with SPADE</td>
<td>7760</td>
</tr>
</tbody>
</table>

- Score Context
  - 8634 (according to website)
  - 16301 (Galaxy S III)
Conclusion

• Our system captures complicated system bugs
• Our system impacts performance negligibly
• Querying system bugs is “straight forward”
• Querying still requires expert knowledge of the system
  • This could be eased by increased developer tools
  • Google could integrate our method in to their tool chain
Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant IIS-1116414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
References