TriggerScope: Towards Detecting Logic Bombs in Android Applications

Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, Giovanni Vigna

Pietro De Nicolao

Politecnico di Milano

June 21, 2016

The paper

2016 IEEE Symposium on Security and Privacy

TriggerScope: Towards Detecting Logic Bombs in Android Applications

Yanick Fratantonio*, Antonio Bianchi*, William Robertson†, Engin Kirda†, Christopher Kruegel*, Giovanni Vigna*
*UC Santa Barbara
{yanick,antoniob,chris,vigna}@cs.ucsb.edu
†Northeastern University
{wkr,ek}@ccs.neu.edu

Introduction to logic bombs

Logic bombs

Logic bomb: malicious application logic that is triggered only under certain (narrow) conditions.

- *Malicious application logic*: violation of user's reasonable expectations
- Malware is designed to target **specific victims, under certain circumstances**

Example: take a navigation application, supposed to help a soldier in a war zone find the shortest route to a location.

- after a given (hardcoded) date, it gives to him longer, more dangerous routes

Another (real) example

RemoteLock: Android app that allows the user to remotely lock and unlock the device by using an user-defined keyword

- The app code also contains the following check:
  `(!= (#sms/#body equals "adfbdfgfsgyhytdfsw")) 0)`
  - The predicate is triggered when an incoming SMS contains that hardcoded string
- By sending a SMS containing that string, the device unlocks
- That's a backdoor implemented with a **logic bomb**!

Problems with traditional defenses

App Stores employ some defenses, but they are not sufficient.

- **Static analysis**: malicious application logic doesn't require additional privileges or make "strange" API calls
  - Malicious behavior is deeply hidden within the app logic
  - Example: the malicious navigation app... behaved like any navigation app!
- **Dynamic analysis**: likely won't execute code triggered only on a future date or in a certain location
  - Code coverage problems
  - Can be detected and evaded
  - Even if covered, how to discern malicious behavior from benign?
- **Manual audit**: if source code is not available, no guarantees
  - Code can be obfuscated

TriggerScope

Key observation

TriggerScope detects logic bombs by precisely analyzing and characterizing **the checks (conditionals, predicates) that guard a given behavior**.

- It gives less importance to the (malicious) behavior itself.

```
if(sms.getBody().equals("adfbdf...")) // Look here!
{
    myObject.doSomething(); // ...not there.
}
```

Trigger analysis

- **Predicate**: logic formula used in a conditional statement
  - `(&& (!= (#sms/#body contains "MPS:") 0) (!= (#sms/#body contains "gps") 0))`
  - **Suspicious predicate**: a predicate satisfied only under very specific, narrow conditions
- **Functionality**: a set of basic blocks in a program
  - **Sensitive functionality**: a functionality performing, directly or indirectly a sensitive operation
  - All calls to Android APIs protected by permissions, and operations involving the filesystem
- **Trigger**: suspicious predicate controlling the execution of a sensitive functionality

Analysis overview (1)

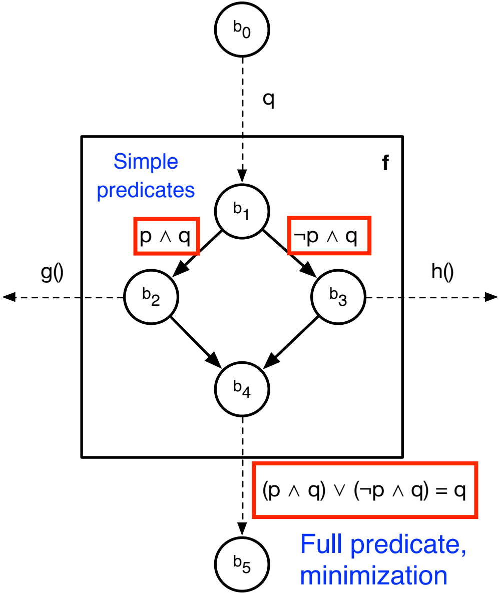
1. **Static analysis** of bytecode; building of Control Flow Graph
2. **Symbolic Values Modeling** for integer, string, time, location and SMS-based objects
3. **Expression Trees** are built and appended to each symbolic object referenced in a check
    - Reconstruction of the *semantics* of the check, often lost in bytecode

Figure 1: Example of expression tree.

Analysis overview (2)

4. **Block Predicate Extraction**: edges of Control Flow Graph are annotated with simple predicates
   - Simple predicate: P in `if P then X else Y`
5. **Path Predicate Recovery and Minimization**
   - Combine simple predicates to get the *full path predicate* that reaches each basic block
   - Minimization: elimination of redundant terms in predicates
     - important to reduce false dependencies

Path Predicate Recovery and Minimization



Analysis overview (3)

6. **Predicate Classification**: a check is **suspicious** if it's equivalent to:
    - Comparison between current *time* value and constant
    - Bounds check on GPS *location*
    - Hard-coded patterns on body or sender of *SMS*
7. **Control-Dependency Analysis**: control dependency between *suspicious predicates* and *sensitive functionalities*.
    - sensitive = privileged Android APIs + fileystem ops
    - **Suspiciousness propagates** with data flows and callbacks
    - Problem: data flows through files
        - When in doubt: suspicious!
8. **Post-processing**: whitelisting for some edge cases

Experiment

Data sets

- **Benign applications**: 9582 apps from Google Play Store
  - They all use time-, location- or SMS-related APIs
  - Actually, TriggerScope identified backdoors in two “benign” apps, confirmed by manual inspection!
- **Malicious applications**: 14 apps from several sources
  - Stealthy malware developed for previous researches
  - Real-world malware samples
  - HackingTeam RCSAndroid

Results of analysis

Analysis step	TP	FP	TN	FN	FPR	FNR
Predicate detection	14	1386	7927	0	14.88%	0%
Suspicious Predicate A.	14	462	8851	0	4.96%	0%
Control-Dependency A.	14	117	9196	0	1.26%	0%
TriggerScope (all)	14	35	9278	0	0.38%	0%

Table 1: Results of analysis after each step. Note how each step is useful to refine the analysis.
$FPR = \frac{FP}{FP+TN}$, $FNR = \frac{FN}{FN+TP}$

False Positives decreasing step after step

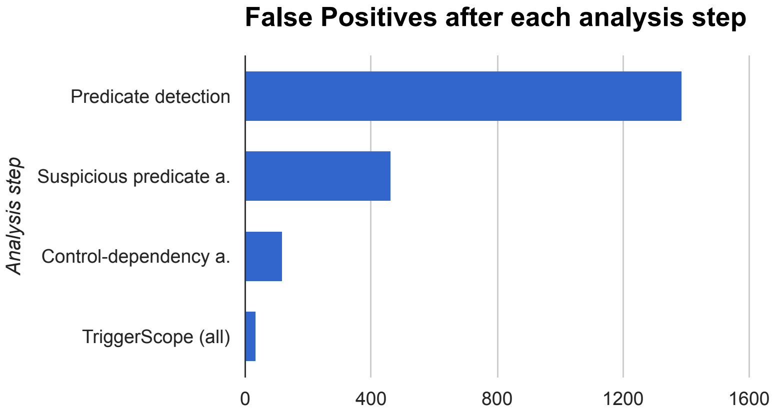


Figure 3: Each analysis stage is useful, because it reduces False Positives.

Critique

Strengths

- TriggerScope provides **rich semantics** on predicates that help manual analysis
  - Great help for manual analysis
  - This makes the tool extensible, open for future research
- Novel approach: **focus on checks**, not malicious behaviors
- **Fewer FPs, FNs** than other tools

Issues: limits of analysis

- Definition of **suspicious predicate** is too narrow
  - Only checks against hardcoded values are considered
- Authors claim **0% FNs**, but the evaluation isn't conclusive
  - *we manually inspected a random subset of 20 applications for which our analysis did not identify any suspicious check. We spent about 10 minutes per application, and we did not find any false negatives.*
  - Difficult to assess FNs if no tool finds anything and source code is unavailable
- This analysis is still **blacklisting**
  - We're competing against attackers' creativity

Issues: evasion techniques

- *Reflection*, *dynamic code loading*, *polymorphism* and *conditional code obfuscation* [Sharif, 2008] can defeat static analysis.
  - Authors say that these techniques are themselves suspicious, but they also have legitimate uses
- Predicate minimization is **NP-complete**
  - Is it possible to design "pathological" code to slow down and defeat analysis?
  - Or result in very complex, meaningless predicates?
- **Exceptions** were not cited as a control flow subversion method
  - *Statically reasoning in the presence of exceptions and about the effects of exceptions is challenging* [Liang, 2014]
  - Unclear how the static analysis engine handles exceptions
  - Unchecked exceptions (e.g. division by zero) could be exploited as stealthy triggers

Related and future work

Related work: AppContext

AppContext [Yang, 2015]: supervised machine learning method to classify malicious behavior statically

1. Starts identifying suspicious actions
2. Context: which category of input controls the execution of those actions?

Similar idea: just looking at the action isn't enough. Differences:

- AppContext only **classifies** triggers as suspicious or not; TriggerScope also provides **semantics** about the predicates, helping manual inspection
- AppContext does not consider the typology of the predicate, only the type of its inputs
- **Higher FP rate** than TriggerScope

Future evolutions

- **Extend trigger analysis** not only to time, location, SMSs
  - The trigger could come e.g. from the network
  - The framework is easily extensible to other types of triggers with more work to model other symbolic values
  - Is there **a more general approach**?
- **Quantitative analysis** of predicate "suspiciousness"
  - Currently, it's defined in a qualitative, ad-hoc way
  - Could be combined with classification methods

References

Sharif M., Lanzi A., Giffin J., Lee W.
Impeding Malware Analysis Using Conditional Code Obfuscation

In: Network and Distributed System Security Symposium, San Diego, CA (2008)

Yang W., Xiao X., Andow B., Li S., Xie T., Enck W.
AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context
In Proceedings of the International International Conference on Software Engineering (ICSE), 2015.

Liang S., Sun W., Might M., Keep A. Van Horn D.
Pruning, Pushdown Exception-Flow Analysis
14th IEEE International Working Conference on Source Code Analysis and Manipulation (2014)