Resilient Decentralized Android Application Repackaging Detection using *Logic Bombs*

Qiang Zeng, Lannan Luo, Zhiyun Qian,
Xiaojiang Du, and Zhoujun Li

CGO 2018, Feb 26th
Vienna, Austria

Application Repackaging Attacks

- App repackaging attacks: an app is **unpacked, modified**, and then **repackaged**
  - The attacker then can sell the repackaged app

- Can be **easily** done, and cause **severe threats**
  - ➢ **Huge monetary loss:** app sales; ad revenue; in-app purchases
  - ➢ **Propagating malicious code**

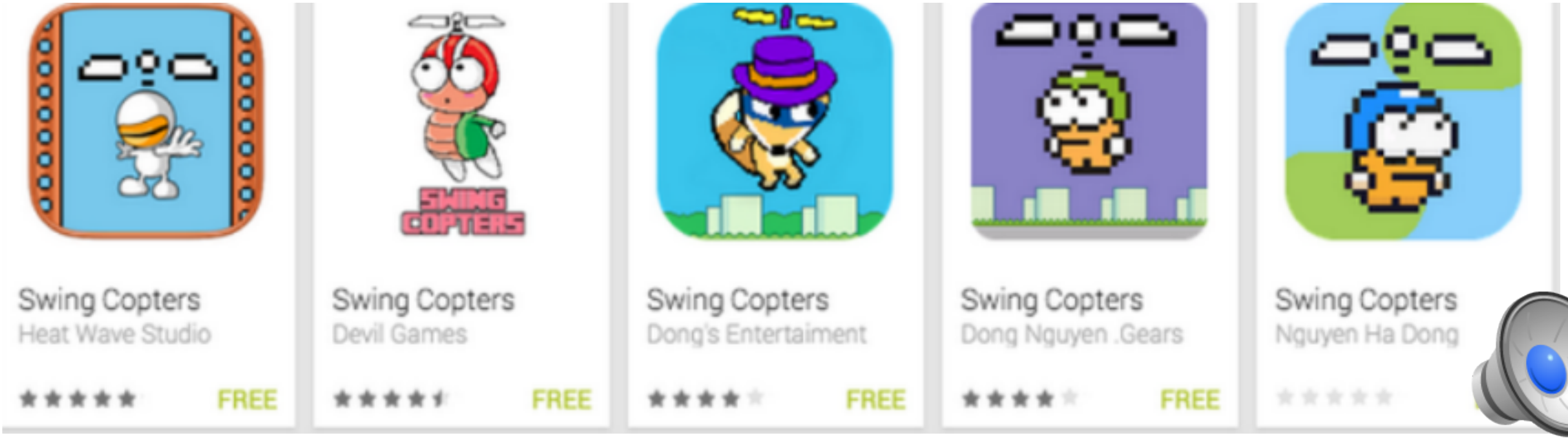
Fact 1: $14B annual monetary loss

➢ E.g., **95%** of "*Monument Valley*" (a popular game app) installations on **Android** are repackaged apps; 60% in the case of iOS

Fact 2: **80%** of malware is built via app repackaging

Existing Countermeasures

- Most app repackaging detection methods rely on
  - **App similarity comparison**
- Disadvantages
  - **Non-scalable** due to comparison with millions of apps
  - **Imprecise** when repackaged apps are obfuscated
  - **Rely on** the app stores to deploy the countermeasures



Goal

- Decentralized App Repackaging Detection
  - **Repackaging Detection Code** is built into apps, so the detection runs on user side when the apps are used

- Advantages
  - **Scalable**
  - **Keeps precise** when handling obfuscated repackaged apps
  - Deployment does **not rely on app stores**
  - **Rich responses** upon detected repackaging attacks
    - Inject crashes; warn the users; notify the developers …

Threat Model and Main Challenge

- The adversary can ***arbitrarily* modify** the protected app
  - **Delete** any suspicious code
  - **Modify** code to bypass repackaging detection

- The adversary can ***arbitrarily* analyze** the protected app to locate/expose Repackaging Detection Code
  - **Blackbox fuzzing**
  - **Whitebox fuzzing**
  - **Program slicing**
  - **Text search**
  - **API hooking**
  - …

The main challenge is *how to protect the Repackaging Detection Code from **various attacks***

Method Used in the Wild

- **Background**
  - The attacker has to re-sign the repackaged app using his private key
  - The public key is part of the app (for signature verification)
  - **Open secret**: the repackaged app's K_{pub} != the original one

```
currKey = getPublicKey(); // Android API
if ( currKey != PUB_KEY) // PUB_KEY is hard coded
     Repackaging detected!
```

- Zero resilience to *any* of the following trivial attacks
  - Text search for calls to "`getPublicKey()`"
  - Change "!=" to "=="
  - Change the value of "`PUB_KEY`"
  - Delete the repackaging detection and response code
  - …

Stochastic Stealthy Network (SSN) [Luo 2016]

- A client-side app repackaging detection technique
- It also used the public key comparison, but tried to be resilient to attacks

Repackaging Detection is invoked at a very low probability to survive **blackbox fuzzing**

```
if(rand() < 0.01) {
   funName = recoverFunName(obfuscatedStr);
   // The reflection call invokes getPublicKey
   currKey = reflectionCall(funName);
   if(currKey != PUBKEY)
      // repackaging detected.
}
```

Reflection is used to hide `getPublicKey()` from **text search**

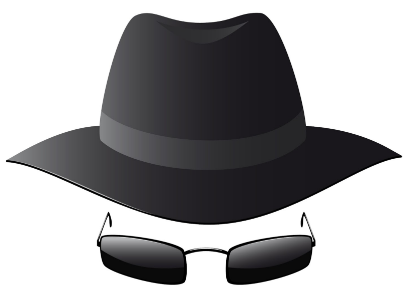
SSN: A Not Successful Attempt

- Vulnerable to ***any*** of the following attacks
  - Force `rand()` to return 0 during fuzzing
  - Symbolic execution to explore suspicious reflection calls
  - Backward program slicing to reveal reflection calls
  - Simple code instrumentation to bypass repackaging detection

The main challenge, i.e., *how to protect the Repackaging Detection Code from attacks*, is **NOT** resolved

Our Insights and Intuition

- Insights: the attacker side is very *different* from the user side
  - **D1:** The hardware/software environments, inputs, and sensor values are **diverse** on the user side, but it is not the case on the attacker side
  - **D2:** A **high code coverage** is usually hard to achieve by attackers, while users altogether play almost every part of the app



Our Insights and Intuition

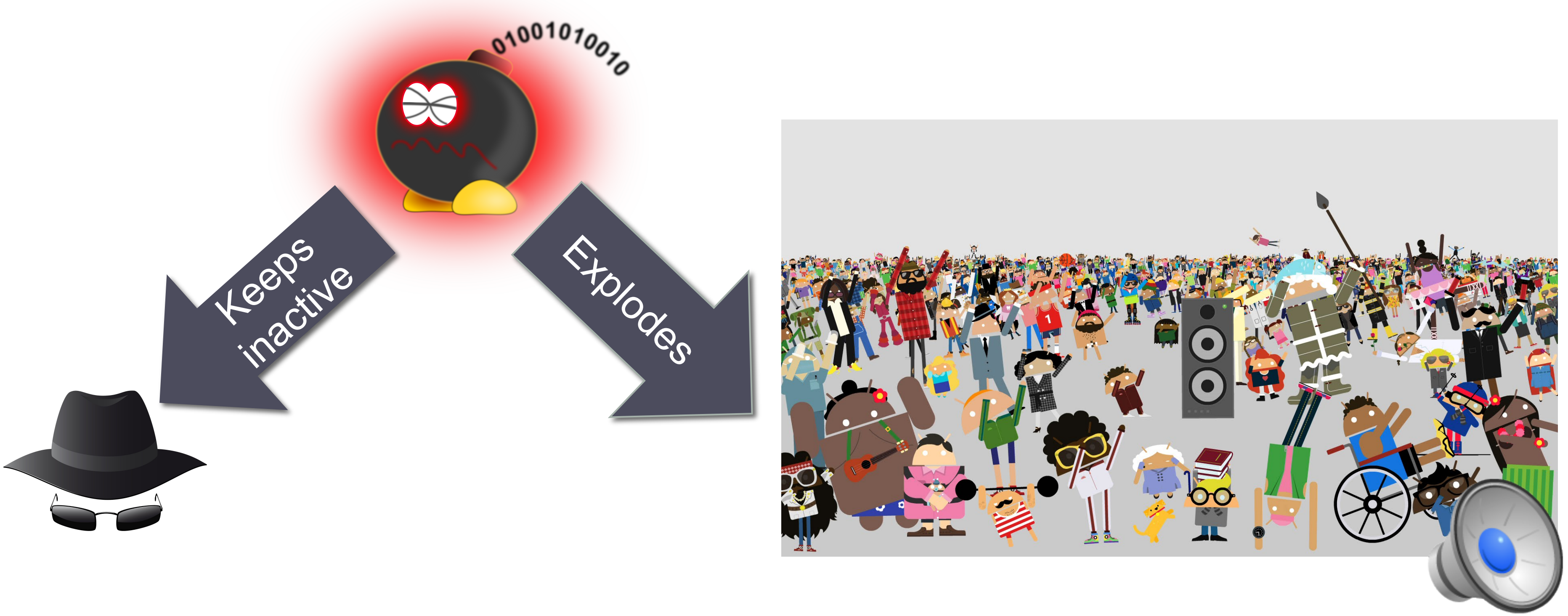
- Insights: the attacker side is very *different* from the user side
  - **D1:** The hardware/software environments, inputs, and sensor values are **diverse** on the user side, but it is not the case on the attacker side
  - **D2:** A **high code coverage** is usually hard to achieve by attackers, while users altogether play almost every part of the app

Background: a Logic Bomb is

- a piece of code that executes under *specific conditions* (e.g., time)
- widely used in malware and **difficult to detect**

Our Insights and Intuition

Intuition: inserting **logic bombs** that exploit the differences between attackers and users, so that they **keep inactive on the attacker side but explode on the user side**



Main Ideas

- The trigger condition of a bomb is met only under *specific inputs, hardware/software environments, or sensor values*
  - Difficult to be activated by an attacker, but easy by diverse users
- Many bombs are inserted
  - Even after some bombs are removed by attackers, many survive
- Taking advantage of the mobile app ecosystem
  - Crashes and pirate warnings lead to a bad app rating
  - Notify the original app developer, who can requests it be taken down

Cryptographically Obfuscated Logic Bombs

- We do *NOT hide* the existence of logic bombs
- We *deter* attackers from deleting/modifying bomb code
  - Given a condition X == c, perform three steps of transformation

```
if (X == c) { // X is a variable, and c is a constant
}
```

(1) Repackaging code is **woven** into the “if” body code

(3) The “if” condition is re-written to **delete the key** “c”

```
if(Hash(X) == H_c) // this line is equivalent to "X==c"
  // "code" is encrypted and can only be decrypted when X=c
  p = decrypt(code, X);
  execute(p);
}
```

(2) The **mixed** code is **encrypted** using the key “c”, and is decrypted during execution if the trigger condition is met

Correctness and Security Analysis

- Correctness: cryptographic hash (~ zero hash collisions) ensures `Hash(X) == `H_c is equivalent to `X==c`

```
if (mMode == 0xfff000) {
    payload;
}
```

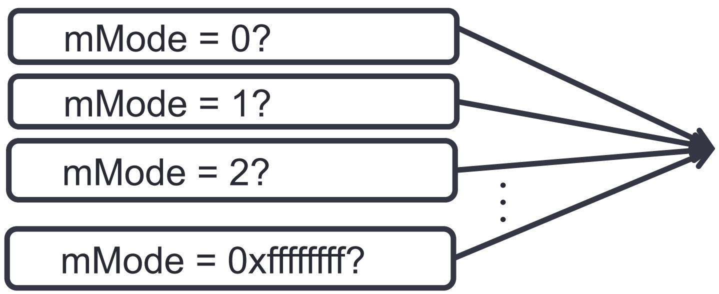
→

```
if (Hash(mMode) ==
da4b9237bacccdf19c0760cab7aec4a8359010b0) {
    p = decrypt (encrypted_payload, mMode);
    execute (p);
}
```

- Security analysis
  - Deleting bombs also **corrupts** the app
  - The encryption key is **removed** from the protected app
  - The hash-involved condition defeats **symbolic execution**

Dealing with Fuzzing

- Fuzzing: attackers may feed the app with massive inputs in order to explode (and thus reveal) logic bombs
  - But it may take billions of times of tries to explode a given bomb




```
if (Hash(mMode) ==
da4b9237bacccdf19c0760cab7aec4a8359010b0) {
    p = decrypt (encrypted_payload, mMode);
    execute (p);
}
```

- Plus, **Artificial Qualified Conditions**
  - A small app may have relatively few Qualified Conditions “if(X==c)”
  - But we can **artificially insert** a large number of Qualified Conditions, each of which can be used to construct a logic bomb

Attackers will have many bombs to fuzz against, while fuzzing is known to be inefficient

Repackaging Detection

- Public key comparison
- Code digest comparison
  - Compare a file’s current digest with the hard-coded one
- Code scanning
  - Checking the integrity of other bombs
  - Checking the function body of `getPublicKey()` in memory

System Design and Implementation

1. Profiling
   - To find hot methods, and we do not insert bombs into them
   - To collect variable values for creating artificial qualified conditions

2. Soot based static analysis to locate existing qualified conditions

3. Javassist to perform bytecode instrumentation

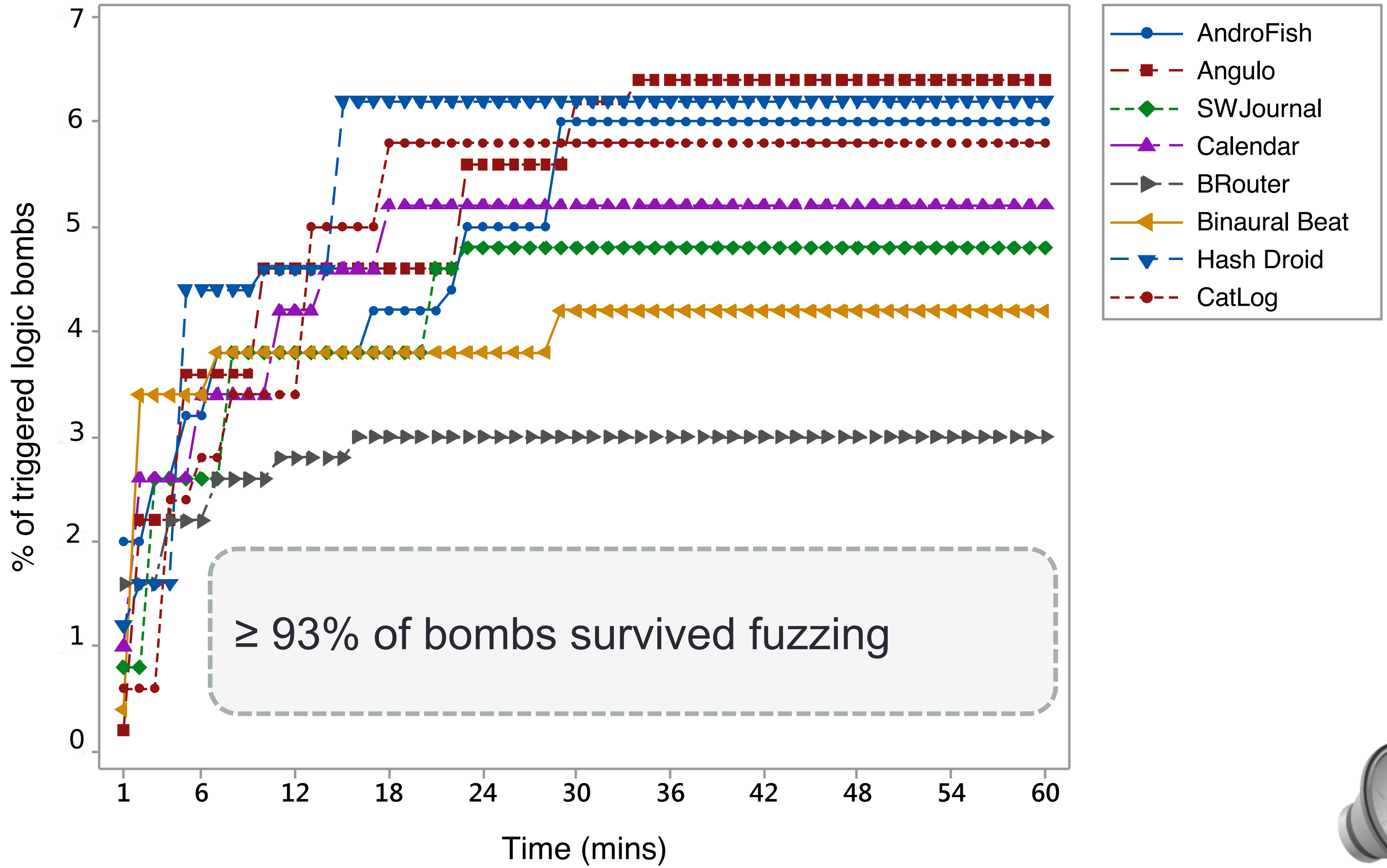
Our system, *BombDroid,* enhances apps without requiring access to their source code

Evaluation: App Statistics and Overhead

Category	# of apps	Avg LOC	Avg # of candidate methods	Avg # of exist. qualified conditions	Avg # of env. var.
Game	105	3,043	95	56	16
Science&Edu.	98	4,046	86	44	8
Sport&Health	87	5,467	113	40	11
Writing	149	7,099	149	67	6
Navigation	121	9,374	185	52	9
Multimedia	108	10,032	203	72	17
Security	152	11,073	242	86	12
Development	143	14,376	373	93	11

1.4% ~ 2.6% slowdown

Evaluation: Bombs Triggered via Fuzzing



Conclusions

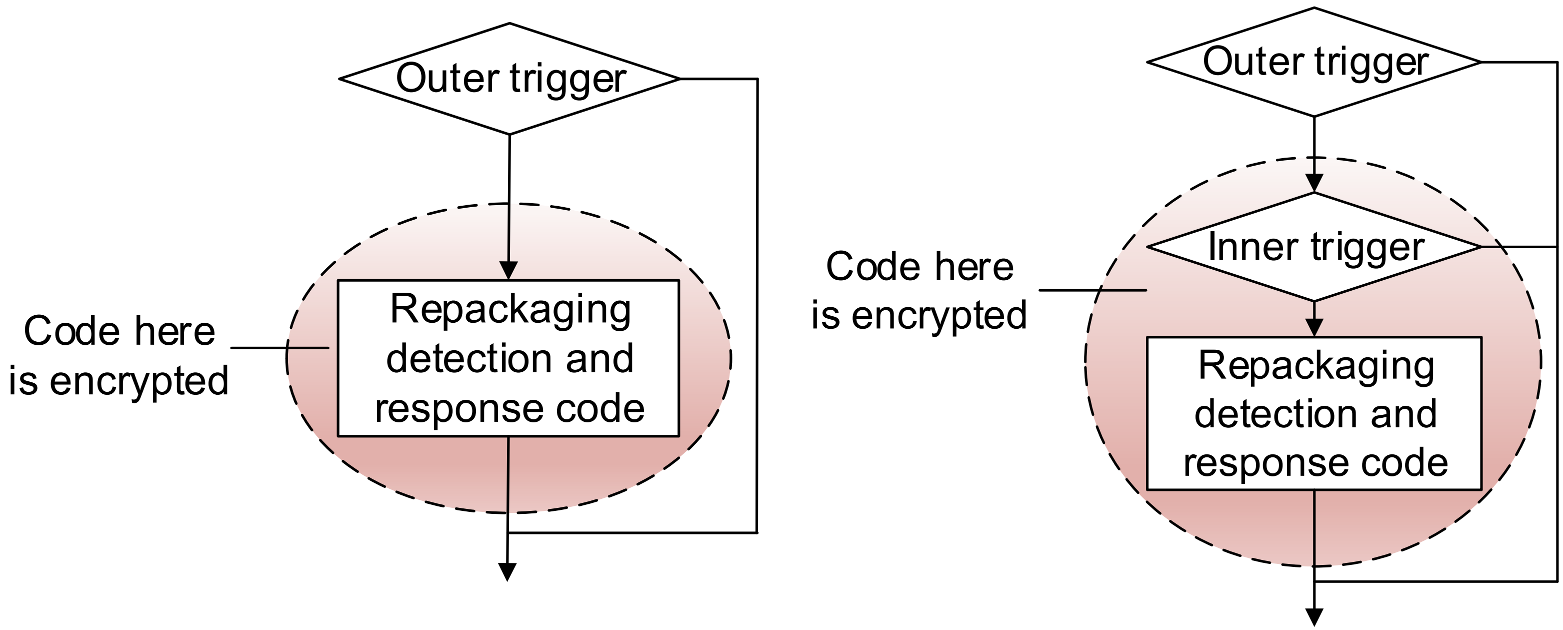
- App repackaging attacks cause huge loss ($14B annual) and propagate (over 80% of) mobile malware
- Centralized repackaging detection has severe limitations
- **Our contributions**
  - The *first resilient decentralized* repackaging detection technique
  - A creative use of logic bombs that protect repackaging detection by exploiting the differences between attackers and users
  - Multiple measures to enhance logic bombs
    - Code weaving, cryptography, artificial qualified conditions, double trigger
  - A bytecode-instrumentation based prototype system

We are in the process of filing a **patent**

Contact me (qzeng@temple.edu) if you are interested in **commercializing it**

Thank you!

Enhancement: Double-trigger Bombs



System Design and Implementation