G2: JANK Methodology

By Callum Bradding, Justin Kim & Sam Boyes

Agenda

- Introduction & Motivation
- Related Works
- JANK Methodology
  - Rename Obfuscation
  - Control Flow Disruption
- Evaluation
  - Performance
  - Storage
  - Effectiveness
- Future Improvements

Introduction

- Android OS is now most popular OS
- Over 100,000 new applications monthly
- Increase of cracked and pirated apps
- Need for ways to secure/protect code

Related Works

- Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild[4]
- Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions[5]
- Code Obfuscation Against Symbolic Execution Attacks[2]
- Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code[1]
- Statistical Deobfuscation of Android Applications[3]

Understanding Android Obfuscation Techniques[4]

- Written by Dong et al.
- In the paper they look at and evaluate obfuscation techniques in Java code written for Android applications
- One technique they look at is Identifier Renaming
- This is the process of renaming identifiers so that they do not mean anything
- Makes the logic of the program harder to understand
- Common technique used in malware
- Most identifier names changed to a combination of “a” and “b”s
- Therefore we need to do something more unique to make it a novel solution

Android Code Protection via Obfuscation Techniques[5]

- Written by Faruki et al.
- The paper looks at how malicious developers use obfuscation tools and techniques to evade anti-malware techniques
- One of these techniques was to change the control flow
- We could do this by inserting dead or irrelevant code which has nothing to do with the program
- Batchelder & Hendren proposed the Add Dead-code Switch Statements (ADSS) model
- Adds a Java bytecode switch into a function which is never executed but does increase the complexity of the function

ADSS Model

```
1  //before ADSS obfuscation
2  if (writeImage != null) {
3    try {
4
5      File file = new File("out");
6      ImageIO.write(writeImage, "png", file);
7    }
8    catch (Exception e) {
9    System.exit(1);
10    }
11 }
12  System.exit(0);
```

```
// ADSS obfuscated code
if(obj != null) {
 try {
  ImageIO.write((RenderedImage)obj,png,
new File(out));
    }
  catch(Exception exception2)  {
   i += 2;
   System.exit(1);
    }
}
label_167:
{ while(lI1.booleanValue() == ___)
  { switch (i) {
    default: break;
    case 3: break label_167;
    System.exit(1);
    continue;
       }
    }
    System.exit(0);
}
```

Taken from Fakuri et al.

Code Obfuscation Against Symbolic Execution Attacks[2]

- Sebastian Banescu et al.
- They show obfuscated code that strictly maintains the functionality property is vulnerable to symbolic execution deobfuscation. Modern tools can determine if branches of code will be executed, and will ignore any that will never be executed under the set of valid inputs
- Provide two techniques to improve obfuscation in the case that the functionality property of code is relaxed slightly
- Range Dividers and Input Invariants
- Range Dividers involves creating branches that will execute but will have the exact same outcome. Symbolic execution needs to explore all of these branches
- Input Invariation involves making sure the functionality property is maintained under the original valid set of inputs, but widening the valid set of inputs and outputs to force the attacker to explore a larger set of inputs, with potentially undefined behavior on originally invalid inputs

Stack Based Self-Modifying Code[1]

- Vivek Balachandran and Sabu Emmanuel.
- Machine code level obfuscation
- Propose a obfuscation technique to protect against static analysis
- Hide jump instructions in the stack so that the control flow of the program looks flat, but is restored at runtime
- This allows other obfuscation techniques, such junk data, to be more effective
- In the case of junk data, the hidden jump instructions causes modern static analysis algorithms to think that otherwise unreachable code blocks are reachable, which then corrupts future instructions

Statistical Deobfuscation of Android Applications[3]

- Bichsel et al.
- Used machine learning on large amounts of unobfuscated code
- Created a model that can create useful variable names from code that has been rename obfuscated
- Can recover about 80% of variable names obfuscated by tools such as ProGuard, or by manual obfuscation such as what we did

JANK Methodology

- Based on the Identifier Renaming talked about by Dong et al. and the ADSS model proposed by Batchelder & Hendren
- JANK is a form of obfuscation that can be applied to source code manually
- Very basic idea of what can be achieved with obfuscation
- We applied this to our application we built called KakuroPuzzler
- This application creates and displays a board for the mathematical puzzle Kakuro
- JANK is made up of two parts:
  - Rename Obfuscation
  - Distorting Control Flow

Rename Obfuscation

- Commonly used technique in obfuscation
- Change identifier names into meaningless names
- Complicates human readability
- In JANK, we use a special kind of rename obfuscation
- As stated in the paper by Dong et al., the most common rename obfuscation technique is to use a combination of ‘a’ and ‘b’
- Therefore, we propose a different technique that goes against the conventions of good programming
- Combination of upper and lower case ‘i’s and ‘l’s and ‘1’ characters
- Names exceeding 30 characters

Example of Rename Obfuscation

```
// Original Code
private void CalculateWage(StaffList employees) {
  while (employees.HasMore()) {
    employee = employees.GetNext(true);
    employee.UpdateWage();
    PayWage(employee);
  }
}


// Rename Obfuscation
private void a(a b) {
  while (b.a()) {
    a = b.a(true);
    a.a();
    a(a);
  }
}
```

Distorting Control Flow

- Order in which statements, instructions and/or function calls executed or evaluated
- Relatively easy to map out functionality of app
- “Junk” functions to complicate control flow
  - The junk functions would be executed but it did not affect the end result

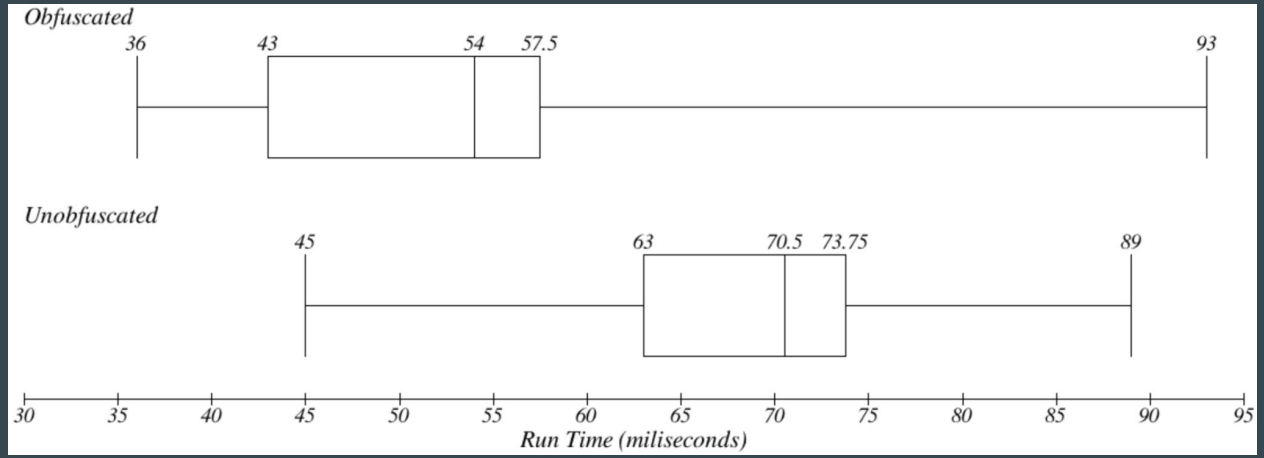
Application of JANK Methodology

```
public void LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1(int IIllIIllIIllllIIIIIIlllllIIIIII, int lllllIIIIIIIIIlll
    IIlllIIlllllllIIIllllllll++;
    IIIIIIIIIIlllllllllllllll[IIllIIllIIllllIIIIIIlllllIIIIII][llllllIIIIIIIIlllllllIIIIIIIIIIIllII].iil1l1l1l1l1l1

    while (IIlllIIlllllllIIIllllllll <= (this.IIllIIllIIllllIIIIIIlllllIIIIII+this.lllllIIIIIIIIIllllllllIIIIIIIIIIIll
        int LLLLi1i1ii1i1l1ll1i11l11il1l1l1 = IlIlIlIlIlIlllllllllIIIIII.nextInt( bound: 2);
        if (IlIlIlIlIlIlllllllllIIIIII.nextInt( bound: 2) == 0) {
            if (LLLLi1i1ii1i1l1ll1i11l11il1l1l1 == 1 && ((IIllIIllIIllllIIIIIIlllllIIIIII - 1) != 0)) {
                LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1( ||||||||||||||||||||||||: IIllIIllIIllllIIIIIIlllllIIIIII - 1, llll
            } else if (IIllIIllIIllllIIIIIIlllllIIIIII + 1 < this.IIllIIllIIllllIIIIIIlllllIIIIII) {
                LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1( ||||||||||||||||||||||||: IIllIIllIIllllIIIIIIlllllIIIIII + 1, llll
            } else LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1(IIllIIllIIllllIIIIIIlllllIIIIII, lllllIIIIIIIIIllll
        } else {
            if (LLLLi1i1ii1i1l1ll1i11l11il1l1l1 == 1 && ((llllllIIIIIIIIIllllllllIIIIIIIIIIIllII - 1) != 0)) {
                LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1(IIllIIllIIllllIIIIIIlllllIIIIII, ||||||||||||||||||||||||: lllll
            } else if (llllllIIIIIIIIIllllllllIIIIIIIIIIIllII + 1 < this.llllllIIIIIIIIIllllllllIIIIIIIIIIIllII) {
                LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1(IIllIIllIIllllIIIIIIlllllIIIIII, ||||||||||||||||||||||||: lllll
            } else LLLLLiiiIIIIIlll1l1l1i11lll1l1l1l1ii1i1i1l1ll1(IIllIIllIIllllIIIIIIlllllIIIIII, lllllIIIIIIIIIllll
        }
    }
}
```

Performance Overhead of JANK

- Java's system library to time both with `nanotime()`
- Measured how long it takes for to generate and display the board
- Each version tested 25 times



Storage Overhead of JANK

- Method deliberately increased length of code
- Final obfuscated APK: `3.9MB`
  - Unobfuscated: `3.5MB`
  - Increase of only 11.4%

Effectiveness of JANK

- Delaying tactic against non-automated attackers
  - Lack of descriptive variable names means difficult to understand
- Only effective against non-automated attackers
  - Tools such as DeGuard can map and refactor variable names
- Requires slight increase in effort from attackers
- Junk code may be optimized out as it does not affect output

Future Works of JANK

- Automate JANK
- Use Range Dividers to protect against White Box
  - Create arbitrary number of branches to execute semantically different but functionally identical functions
  - Significantly slows WB testing
- Create pseudo-random generated number that restarts process if not below certain value to protect against Black Box
  - Minimal performance impact
- Vulnerable to tampering attacks
  - Implement self-check to ensure no modifications

Reference List

- [1] Vivek Balachandran and Sabu Emmanuel. 2013. Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code. Information Forensics and Security, IEEE Transactions on 8 (04 2013), 669–681. DOI:http://dx.doi.org/10.1109/TIFS.2013.2250964
- [2] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation against Symbolic Execution Attacks. In Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16). Association for Computing Machinery, New York, NY, USA, 189–200. DOI:http://dx.doi.org/10.1145/2991079.2991114
- [3] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 343–355. DOI:http://dx.doi.org/10.1145/2976749.2978422
- [4] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. (2018). DOI: https://arxiv.org/abs/1801.01633
- [5] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, and Manoj Gaur. 2016. Android Code Protection via Obfuscation Techniques: Past, Present and FutureDirections. (2016). DOI: https://arxiv.org/abs/1611.10231
- [6] Neil Rowe PhD. 1985. Naming in Programming. Computers in the Schools 2, 2-3 (1985), 241–253. DOI:http://dx.doi.org/10.1300/J025v02n02_24
- [7] Michael Batchelder and Laurie Hendren. 2007. Obfuscating Java: The Most Pain for the Least Gain. Lecture Notes in Computer Science 4420 (03 2007), 96–110. DOI: http://dx.doi.org/10.1007/978-3-540-71229-9_7

Thanks for Listening!