Mind the Gap – Dissecting the Android patch gap

Ben Schlabs <ben@srlabs.de>
Allow us to take you on two intertwined journeys

This talk in a nutshell

Research journey

- Wanted to understand how fully-maintained Android phones can be exploited
- Found surprisingly large patch gaps for many Android vendors – some of these are already being closed
- Also found Android exploitation to be unexpectedly difficult

Engineering journey

- Wanted to check thousands of firmwares for the presence of hundreds of patches
- Developed and scaled a rather unique analysis method
- Created an app for your own analysis
Android patching is a known-hard problem

Patching challenges

- Computer OS vendors regularly issue patches
- Users “only” have to confirm the installation of these patches
- Still, enterprises consider regular patching among the most effortful security tasks

The nature of Android makes patching so much more difficult

- “The mobile ecosystem’s diversity […] contributes to security update complexity and inconsistency.” – FTC report, March 2018
- Patches are handed down a long chain of typically four parties before reaching the user
- Only some devices get patched (2016: 17%). We focus our research on these “fully patched” phones

Patch ecosystems

OS vendor

- Microsoft
- Apple
- Linux distro

OS patches

Endpoints & servers

Our research question – How many patching mistakes are made in this complex Android ecosystem? That is: how many patches go missing?
Vendor patch claims can be unreliable; independent verification is needed

How do we determine whether an Android binary has a patch installed, without access to the corresponding source code?

- Trust vendor claims?
  - Perform test
    - Patch missing
    - Patched
    - Test inconclusive
    - Not affected
  - Claimed patch level: 2017-09-01

- Try exploiting the corresponding vulnerability?
  - No exploits publicly available for most Android bugs
  - A missing patch also does not automatically imply an open vulnerability (It’s complicated. Let’s talk about it later)

- Apply binary-only patch heuristics
  - Find evidence in the binary itself on whether a patch is installed
  - Scale to cover hundreds of patches and thousands of phones
  - The topic of this presentation

Important distinction: A missing patch is not automatically an open security vulnerability. We’ll discuss this a bit later.
Patching is necessary in the Android OS and the underlying Linux kernel

### Android OS patching ("userland")

- Android Open Source Project (AOSP) is maintained by Google
- In addition, chipset and phone vendors extend the OS to their needs
- Most exposed attack surface: The OS is the primary layer of defense for remote exploitation
- Monthly security bulletins published by Google
- Clear versioning around Android, including a patch level date, which Google certifies for some phones

**Responsibility**
- Android Open Source Project (AOSP) is maintained by Google
- In addition, chipset and phone vendors extend the OS to their needs

**Security relevance**
- Most exposed attack surface: The OS is the primary layer of defense for remote exploitation

**Patch situation**
- Monthly security bulletins published by Google
- Clear versioning around Android, including a patch level date, which Google certifies for some phones

### Linux kernel patching

- Same kernel that is used for much of the Internet
- Maintained by a large ecosystem
- Chipset and phone vendors contribute hardware drivers, which are sometimes kept closed-source
- Attackable mostly from within device
- Relevant primarily for privilege escalation ("rooting")
- Large number of vulnerability reports, only some of which are relevant for Android
- Tendency to use old kernels even with latest Android version; e.g., Kernel 3.18 from 2014, end-of-life: 2017

We focus our attention on userland patches
Agenda

- Research motivation
- Spot the Android patch gap
- Try to exploit Android phones
We want to check hundreds of patches on thousands of Android devices

Android userland patch analysis
- Android’s 2017 security bulletins list ~280 bugs (~CVEs) with Critical or High severity
- Source code is available for ~240 of these bugs
- Of these userland bugs, ~180 originate from C/C++ code (plus a few Java)
- So far, we implemented heuristics for 164 of the corresponding patches

Out-of-scope (for now)
- ~700 kernel and medium/low severity userland patches
- The remaining bugs are in closed-source vendor-specific components
- We do not yet support most Java patches

The heuristics would optimally work on hundreds of thousands of Android firmwares:
- 60,000 Android variants [3]
- Regular updates for many of these variants
The patch gap: Android patching completeness varies widely for different phones

<table>
<thead>
<tr>
<th>Phone</th>
<th>Android version</th>
<th>Patch level</th>
<th>Patches “missing”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Pixel 2</td>
<td>8.1</td>
<td>Apr 2018</td>
<td>0</td>
</tr>
<tr>
<td>Samsung J5 (2017)</td>
<td>7.0</td>
<td>Apr 2018</td>
<td>0</td>
</tr>
<tr>
<td>NOKIA 3</td>
<td>7.1.1</td>
<td>Mar 2018</td>
<td>0</td>
</tr>
<tr>
<td>Wiko Freddy</td>
<td>6.0.1</td>
<td>Sep 2017</td>
<td>17</td>
</tr>
</tbody>
</table>
Binary-only analysis: Conceptually simple

1. Prepare patch test set

   - Vulnerable source code
     - Compile with different compilers, compiler configurations, CPU options
     - Mask volatile information (e.g. call destinations)
     - Collection of unpatched binaries
     - Apply patch
     - Patched source code
     - Collection of patched binaries

2. Test for patch presence

   - Binary file
     - Mask volatile information
     - Collection of patched binaries
     - Compare to collections: Find match with patched or unpatched sample

---

Security Research Labs
At scale, three compounding challenges need to be solved

**Too much source code**
- There is too much source code to collect
- Once collected, there is too much source code to compile

**Too many compilation possibilities**
- Hard to guess which compiler options to use
- Need to compile same source many times

**Hard to find code “needles” in binary “haystacks”**
- Without symbol table, whole binary needs to be scanned
- Thousands of signatures of arbitrary length
Signature generation would require huge amounts of source code

One Android source code tree is roughly 50 GiB in size

Source code trees are managed in a manifest, which lists git repositories with revision and path in a source code tree

```xml
...<project name="platform/external/zxing" revision="d2256df36df8778a3743e0a71eab0cc5106b98c9"/>
<project name="platform/frameworks/av" revision="330d132dfab2427e940cfaf2184a2e549579445d"/>
<project name="platform/frameworks/base" revision="85838feaea8c8c8d38c4262e74d911e59a275d02"/>
...+~500 MORE REPOSITORIES
```

Currently ~1100 source code trees are used in total (many more exist!)

1100 x 50 GiB = 55 TiB

Would require huge amount of storage, CPU time, and network traffic to check out everything

- Hundreds of different Android revisions (e.g. android-7.1.2_r33)
- Device-specific source code trees (From Qualcomm Codeaurora CAF)
We leverage a FUSE (filesystem in userspace) to retrieve files only on demand

**Insight: The same git repositories are used for many manifests.**

<table>
<thead>
<tr>
<th>Manifest 1</th>
<th>Manifest 2</th>
<th>Manifest 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform/frameworks/av rev 330d132d</td>
<td>platform/frameworks/base rev 85838fea</td>
<td>platform/frameworks/av rev deadbeef</td>
</tr>
<tr>
<td>platform/frameworks/base rev 85838fea</td>
<td>platform/frameworks/base rev 18fac24b</td>
<td>platform/frameworks/base rev cafebabe</td>
</tr>
</tbody>
</table>

**How this can be leveraged**

**Filesystem in userspace (FUSE)**
- Store each git repository only once (with `git clone --no-checkout`)
- Extract files from git repository on demand when the file is read
- Use database for caching directory contents

**Reduces storage requirement by >99%:**

55 TiB => 300 GiB

Saves network bandwidth and time required for checkout

Prevents IP blocking by repository servers
Brute-forcing 1000s of compiler variants finds 74 that produce valid signatures for all firmwares tested to date

Tests are regularly optimized
- Our collection includes 3897 compiler configuration variants, only 74 of which are required for firmwares tested to date.
- To ensure a high rate of conclusive tests, test results are regularly checked for success.
- The test suite is amended with additional variants from the collection as needed.
- The collection itself is amended with additional compiler configuration variants as they become relevant.

Successful sub-tests

- For 224 tested 64-bit firmwares, signatures from the first 74 compiler config variants provide full test coverage
  - 74 variants → 6,944 signatures → 3MB
- We tried 3,897 variants → 775,795 signatures → 34MB

Just two variants account for 60% of successful sub-tests:
- gcc version 4.9.x-google 20140827 (prerelease)
- Android clang version 3.8.256229

Both were run with each git’s default configuration

Security Research Labs
Using improved rolling signatures, we can efficiently search the binary ‘haystack’ for our code ‘needles’.

Sanitize arguments before checksumming
- Potential relocation entries are detected based on instruction.
- Zero-out volatile bits

Match signatures of arbitrary lengths using sliding windows
- Two overlapping sliding windows
- Only needs powers of 2 as window sizes to match arbitrary function lengths
- Allows efficient scanning of a binary for a large number of signatures

Process step

Hex dump of instruction

Assembly code / instructions

To avoid false positives (due to guessed relocation entries), signature is matched from the first window to the end of the overlapping window.
Putting it all together: With all three scaling challenges overcome, we can start testing

**Prepare patch test set**

- Mount source code tree
  - Read manifest
  - Fuse filesystem to read files on demand

- Run source-code analysis
  - Source-code patch analysis is much easier than binary analysis
  - Determines whether a signature match means that the patch is applied or not

- Generate build log
  - Run build system in dry-run mode, don’t compile everything
  - Save log of all commands to be executed
  - Various hacks/fixes to build system required

**Preprocess source files**

- Use command line from saved build log
- Save preprocessor output in database

**Recompile with variants**

- >50 different compiler binaries
- All supported CPU types
- Optimization levels (e.g. -O2, -O3)
- 3897 combinations in total, 74 in our current optimized set

**Generate signatures**

- Evaluate relocation entries and create signatures **for each compiler variant**

**Test for patch presence**

- Find and extract function (using symbol table or rolling signature)
- Mask relocation entries from signature
- Calculate and compare hash of remaining code
SnoopSnitch version 2.0 introduces patch analysis for all Android users

**Tool name**

SnoopSnitch

**Purpose**

- [new in 2.0] Detect potentially missing Android security patches
- Collect network traces on Android phone and analyze for abuse
- Optionally, upload network traces to GSMmap for further analysis

**Requirements**

- Android version 5.0
- Patch level analysis: All phones incl. non-rooted
- Network attack monitoring: Rooted Qualcomm-based phone

**Source**

Search: SnoopSnitch

---

![SnoopSnitch interface](image)

**Android patch level analysis**

<table>
<thead>
<tr>
<th>Status</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patched</td>
<td>102</td>
</tr>
<tr>
<td>Patch missing</td>
<td>3</td>
</tr>
<tr>
<td>After claimed patch level</td>
<td>0</td>
</tr>
<tr>
<td>Test inconclusive</td>
<td>15</td>
</tr>
<tr>
<td>Not affected</td>
<td>0</td>
</tr>
</tbody>
</table>

**Claimed patch level: 2018-03-05**

*Graphical representation of patch levels for different months,*

Last analysis: 11 Apr 2018 14:10:09
### Missed patches

<table>
<thead>
<tr>
<th>Missed patches</th>
<th>Vendor</th>
<th>Samples*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 1</td>
<td>BlackBerry</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Essential</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Fairphone</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>GM</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Google</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>LG</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Motorola</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>OnePlus</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Samsung</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>SHARP</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Sony</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Wileyfox</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Xiaomi</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>ZUK</td>
<td>Lots</td>
</tr>
<tr>
<td>1 to 2</td>
<td>Asus</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>HONOR</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>HTC</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Huawei</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Nokia</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>vivo</td>
<td>Lots</td>
</tr>
<tr>
<td>2 to 4</td>
<td>Alps</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Infinix</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>LEAGOO</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Lenovo</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>TCL</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>TECNO</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Vernee</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>WIKO</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>ZTE</td>
<td>Many</td>
</tr>
<tr>
<td>4 or more</td>
<td>Blackview</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>DOOGEE</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>GIONEE</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>Itel</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>LeEco</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Meizu</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Micromax</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Nubia</td>
<td>Lots</td>
</tr>
<tr>
<td></td>
<td>OPPO</td>
<td>Many</td>
</tr>
<tr>
<td></td>
<td>Oukitel</td>
<td>Many</td>
</tr>
</tbody>
</table>

### Notes

- The tables show the average number of missing Critical and High severity patches before the claimed patch date.
- Some phones are included multiple times with different firmware releases.
- Not all patch tests are always conclusive, so the real number of missing patches could be higher.
- Not all patches are included in our tests, so the real number could be higher still.
- Only phones are considered that were patched October-2017 or later.
- A missing patch does not automatically indicate that a related vulnerability can be exploited.

---

* Samples – Few: 5-9; Many: 10-49; Lots: 50+
Vendors are improving the speed to bring patches to their phones, especially for newer Android versions – Samsung

Patch delay for different Android versions over time - Samsung
Vendors are improving the speed to bring patches to their phones, especially for newer Android versions – Huawei
Phones more recently receive updates faster and with fewer gaps

**Patch delay and number of missing patch over time - Huawei P9 Lite**

- Days between patch bulletin and firmware build
- Missed patches [crit+high]

**Linear (Days between patch bulletin and firmware build)**

**Linear (Missed patches [crit+high])**
Agenda

- Research motivation
- Spot the Android patch gap
  
  Try to exploit Android phones
Can we now hack Android phones due to missing patches?

<table>
<thead>
<tr>
<th>At first glance, Android phones look hackable</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ We find that most phones miss patches within their patch level</td>
</tr>
<tr>
<td>▪ While the number of open CVEs can be smaller than the number of missing patches, we expect some vulnerabilities to be open</td>
</tr>
<tr>
<td>▪ Many CVEs talk of “code execution”, suggesting a hacking risk based on what we experience on Windows computers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VS. Mobile operating systems are inherently difficult to exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Modern exploit mitigation techniques increase hacking effort</td>
</tr>
<tr>
<td>▪ Mobile OSs explicitly distrust applications through sandboxing, creating a second layer of defense</td>
</tr>
<tr>
<td>▪ Bug bounties and Pwn2Own offer relatively high bounties for full Android exploitation</td>
</tr>
</tbody>
</table>
**Do criminals hack Android?** Very rarely.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Social engineering</th>
<th>Local privilege escalation</th>
<th>Remote compromise</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trick user into insecure actions:</strong></td>
<td>- Install malicious app</td>
<td>- Trick user into installing malicious app</td>
<td>- Exploit vulnerability in an outside-facing app (messenger, browser)</td>
</tr>
<tr>
<td>- Then grant permissions</td>
<td>- Then exploit kernel-level vulnerability to gain control over device, often using standard “rooting” tools</td>
<td>- Targeted device compromise, e.g. FinFisher and Crysaor (Same company as infamous Pegasus malware)</td>
<td>- Then use local privilege escalation</td>
</tr>
<tr>
<td>- Possibly request ‘device administrator’ role to hinder uninstallation</td>
<td>- Advanced malware</td>
<td>- (Google bug bounty, Pwn2Own)</td>
<td></td>
</tr>
</tbody>
</table>

**Used for**
- Ransomware [File access permission]
- 2FA hacks [SMS read]
- Premium SMS fraud [SMS send]

**Frequency in criminal activity**
- Almost all Android “Infections”
- Regular observed in advanced malware and spying

**Made harder through patching**
- (userland or kernel)
- (userland and kernel)
An exploitable vulnerability implies a missing patch, but not the other way around

<table>
<thead>
<tr>
<th>Missing patches in source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code parts that are ignored during compilation</td>
</tr>
<tr>
<td>Missed patches in binary</td>
</tr>
<tr>
<td>Vendor created alternative patch</td>
</tr>
<tr>
<td>Vulnerability requires a specific configuration</td>
</tr>
<tr>
<td>Bug is simply not exploitable</td>
</tr>
<tr>
<td>Errors in our heuristic (it happens!)</td>
</tr>
<tr>
<td>Open vulnerabilities</td>
</tr>
</tbody>
</table>

Diagram:
- Missing patches (source code analysis)
- Missing patches (binary analysis)
- Open vulnerabilities
A single Android bug is almost certainly not enough for exploitation

Android remote code execution is a multi-step process

1. **Information leakage** is used to derive ASLR memory offset (alternatively for 32-bit binaries, this offset can possibly be brute-forces)
2. **Corrupt memory** in an application. Examples:
   - Malicious video file corrupts memory using Stagefright bug
   - Malicious web site leverages Webkit vulnerability
   - This gives an attacker control of the application including the apps access permission
3. Do the same again with two more bugs to gain access to system context or kernel
   - This gives an attacker all possible permissions (system context), or full control over the device (kernel)

Simplified exploit chain examples with 4 bugs

- **Application context**
  - 1. Info leakage (IL)
  - 2. Memory corruption (MC)

- **System context**
  - 3a. IL
  - 4a. MC

- **Kernel**
  - 3b. IL
  - 4b. MC

Aside from exploiting MC and IL programming bugs, Android has experienced logic bugs that can enable alternative, often shorter, exploit chains
Remotely hacking a modern Android device usually requires chains of bugs.
### Famed real-world exploit examples

<table>
<thead>
<tr>
<th>Year</th>
<th>Device</th>
<th>Exploit Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016</td>
<td>Return to libstagefright</td>
<td>Heap pointer leak to bypass ASLR protection</td>
</tr>
<tr>
<td>2017</td>
<td>BlueBorne</td>
<td>Content view client in Chrome allowed arbitrary intent scheme opening, which allows escaping the Chrome sandbox</td>
</tr>
<tr>
<td>2017</td>
<td>Pixel / Nexus 6P</td>
<td>Attacker perform arbitrary read/write operations leading to code execution based on incorrect optimization assumption in Chrome v8</td>
</tr>
<tr>
<td>2018</td>
<td>Pixel</td>
<td>Chrome V8 bug to get RCE in sandbox using a OOB bug in GetFirstArgument AsBytes function</td>
</tr>
</tbody>
</table>

### Exploit chain

1. Heap pointer leak to bypass ASLR protection
2. ROP execution in mediaserver process
3. Module pointer leak to get address of executable code
4. Call mprotect to get RCE into privileged system-server domain

### BlueBorne

BlueBorne is a vulnerability in the Android Bluedroid/Fluorid userland stack, which is already a high-privileged domain.

### Information leak

Information leak vulnerability leaks arbitrary data from the stack, which allows an attacker to derive ASLR base address for a bypass.

### Exploit chain does not include break-out of untrusted app context

Exploit chain does not include break-out of untrusted app context.
Take aways

- Android patching is more complicated and less reliable than a single patch date may suggest – but the situation has started improving

- Remote Android exploitation is also much more complicated than commonly thought

- You can finally check your own patch level thanks to binary-only analysis, and the app SnoopSnitch

Many thanks to Jakob Lell, Stephan Zeisberg, Jonas Schmid, Mark Carney, Lukas Euler, Patrick Lucey, and Karsten Nohl!

Questions?

Ben Schlabs  <ben@srlabs.de>
References

1. Federal Trade Commision, **Mobile Security Updates: Understanding the Issues**, February 2018


3. Google, **Android Security 2017 Year In Review**, March 2018
Backup
A bit more background: Android firmwares go from source code to binaries in two steps

**Source code**
contains placeholders that are filled in during preprocessing

```c
#include <limits.h>
#include <string.h>
void foo(char* fn){
    char buf[PATH_MAX];
    strncpy(buf, fn, PATH_MAX);
}
```

**Compiler**
preprocesses and compiles source code into object files that are then fed into the linker

```asm
stp x28, x27, [sp,#-32]!
[...]
orr w2, wzr, #0x1000
mov x1, x8
bl 0 <strncpy>
[...]
ret
```

**Linker**
combines the object files into an executable firmware binary.

```asm
stp x28, x27, [sp,#-32]!
[...]
orr w2, wzr, #0x1000
mov x1, x8
bl 11b3e8 <strncpy@plt>
[...]
ret
```
The basic idea: Signatures can be generated from reference source code

Compile reference source code (before and after patch)

Parse disassembly listing for relocation entries

Disassembly of object file, after compiler but before linker
0000000000000000 <impeg2d_api_reset>:
  0:  a9bd7bfd   stp    x29, x30, [sp, #-48]!
  4:  910003fd   mov    x29, sp
 20:  f9413e60   ldr    x0, [x19, #632]
 24:  52800042   mov    w2, #0x2                        // #2
 28:  b9402021   ldr    w1, [x1, #32]
 2c:  94000000   bl     0
 3c:  94000000   bl     0 <impeg2_buf_mgr_release>
 4c:  00000000   

Instruction format of the bl instruction
100x 01 0000000000000000

Prepare patch test set

Sanitize instructions
Toss out irrelevant destination addresses of the instruction

Create hash of remaining binary code

Generate signature containing function length, position/type of relocation entries, and hash of the code
Using our custom FUSE, we can finally generate a large collection of signatures

1. Mount source code tree
   - Read manifest
   - Use FUSE filesystem to read files on demand

2. Run source-code analysis
   - Source-code patch analysis is much easier than binary analysis
   - Determines whether a signature match means that the patch is applied or not

3. Generate build log
   - Run build system in dry-run mode, don’t compile everything
   - Save log of all commands to be executed
   - Various hacks/fixes to build system required

4. Preprocess source files
   - Use command line from saved build log
   - Save preprocessor output in database

5. Recompile with variants
   - >50 different compiler binaries
   - All supported CPU types
   - Optimization levels (e.g. -O2, -O3)
   - 3897 combinations in total, 74 in our current optimized set

6. Generate signatures
   - Evaluate relocation entries and create signatures for each compiler variant

Next question: How many different compiler variants do we need?
# Finding needles in a haystack: What do we do if there is no symbol table?

<table>
<thead>
<tr>
<th>Test for patch presence</th>
<th>Function found in symbol table</th>
<th>Function not in symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Challenge</strong></td>
<td><strong>Insight</strong></td>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td>Checking signature at each position is computationally expensive</td>
<td>Similar problem already solved by rsync</td>
<td>Take advantage of rsync rolling checksum algorithm</td>
</tr>
<tr>
<td>Relocation entries are not known while calculating checksum</td>
<td>Relocation entries are only used for certain instructions</td>
<td>Guess potential relocation entries based on instruction type and sanitize args before checksumming</td>
</tr>
<tr>
<td>32bit code uses Thumb encoding, for which instruction start is not always clear</td>
<td>Same binary code is often also available in 64bit version based on same source code</td>
<td>Only test 64bit code</td>
</tr>
</tbody>
</table>

## Challenge

- Checking signature at each position is computationally expensive
- Relocation entries are not known while calculating checksum
- 32bit code uses Thumb encoding, for which instruction start is not always clear

## Insight

- Similar problem already solved by rsync
- Relocation entries are only used for certain instructions
- Same binary code is often also available in 64bit version based on same source code

## Solution

- Take advantage of rsync rolling checksum algorithm
- Guess potential relocation entries based on instruction type and sanitize args before checksumming
- Only test 64bit code

---

Simply compare function with pre-computed samples