Mind the Gap –
Uncovering the Android patch gap through binary-only patch analysis

HITB conference, April 13, 2018

Jakob Lell <jakob@srlabs.de>
Karsten Nohl <nohl@srlabs.de>

Security
Research
Labs

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
- 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

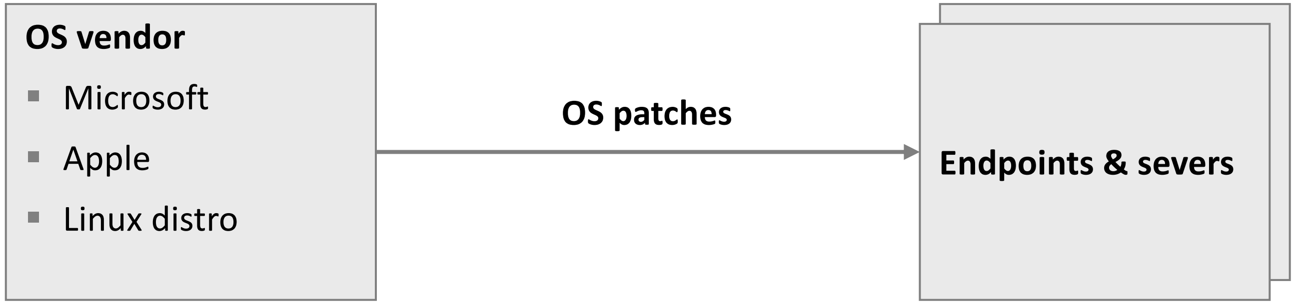
Android patching is a known-hard problem

Patching challenges

Patch ecosystems

Patching is hard to start with

- 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

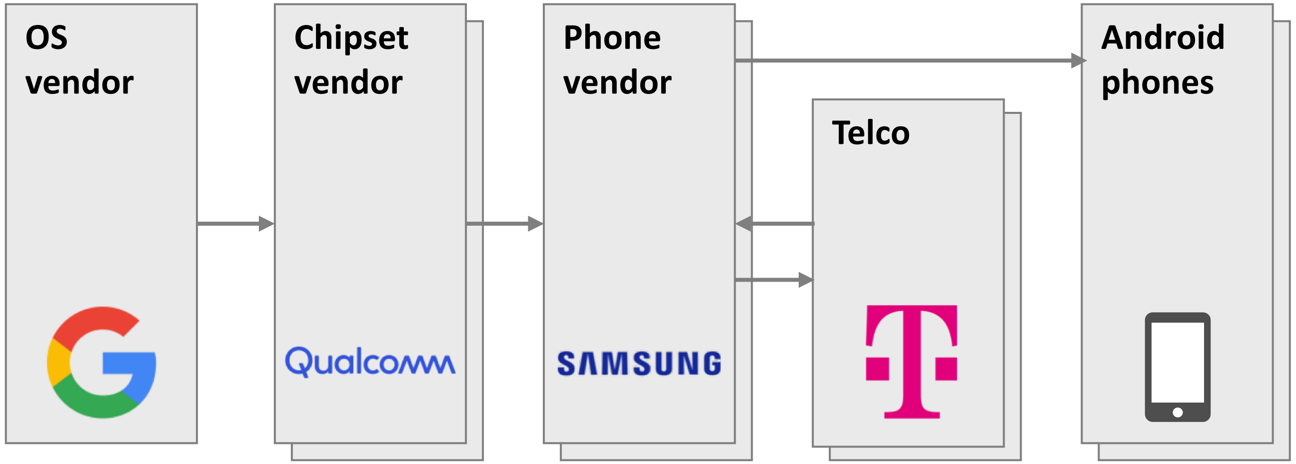


OS vendor

- Microsoft
- Apple
- Linux distro

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 [1]
- Patches are handed down a long chain of typically four parties before reaching the user
- Only some devices get patched (2016: 17% [2]). We focus our research on these “fully patched” phones

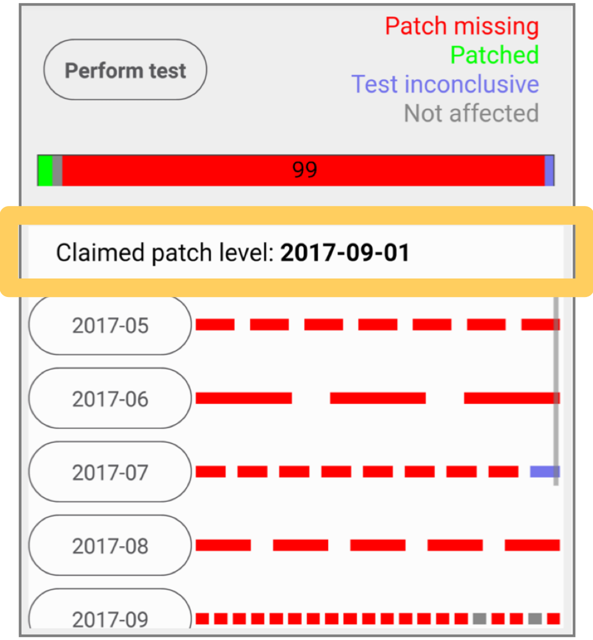


Our research question – How many patching mistakes are made in this complex Android ecosystem? That is: how many patches go missing?

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?



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”)	Linux kernel patching
Responsibility	▪ Android Open Source Project (AOSP) is maintained by Google ▪ In addition, chipset and phone vendors extend the OS to their needs	▪ 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
Security relevance	▪ Most exposed attack surface: The OS is the primary layer of defense for remote exploitation	▪ Attackable mostly from within device ▪ Relevant primarily for privilege escalation (“rooting”)
Patch situation	▪ Monthly security bulletins published by Google ▪ Clear versioning around Android, including a patch level date, which Google certifies for some phones	▪ 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

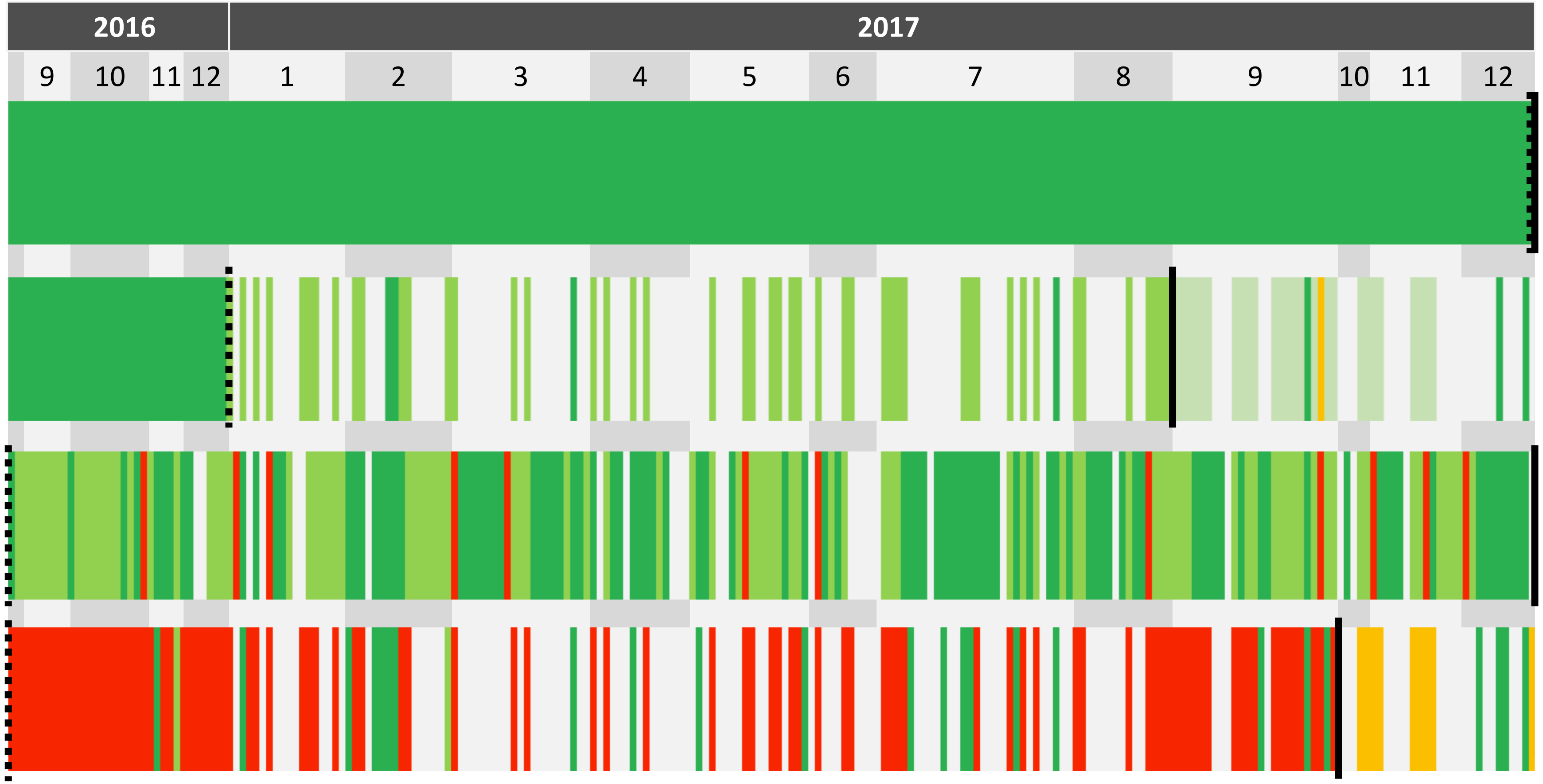
Agenda

- Research motivation
- **Spot the Android patch gap**
- Try to exploit Android phones

We want to check hundreds of patches on thousands of Android devices

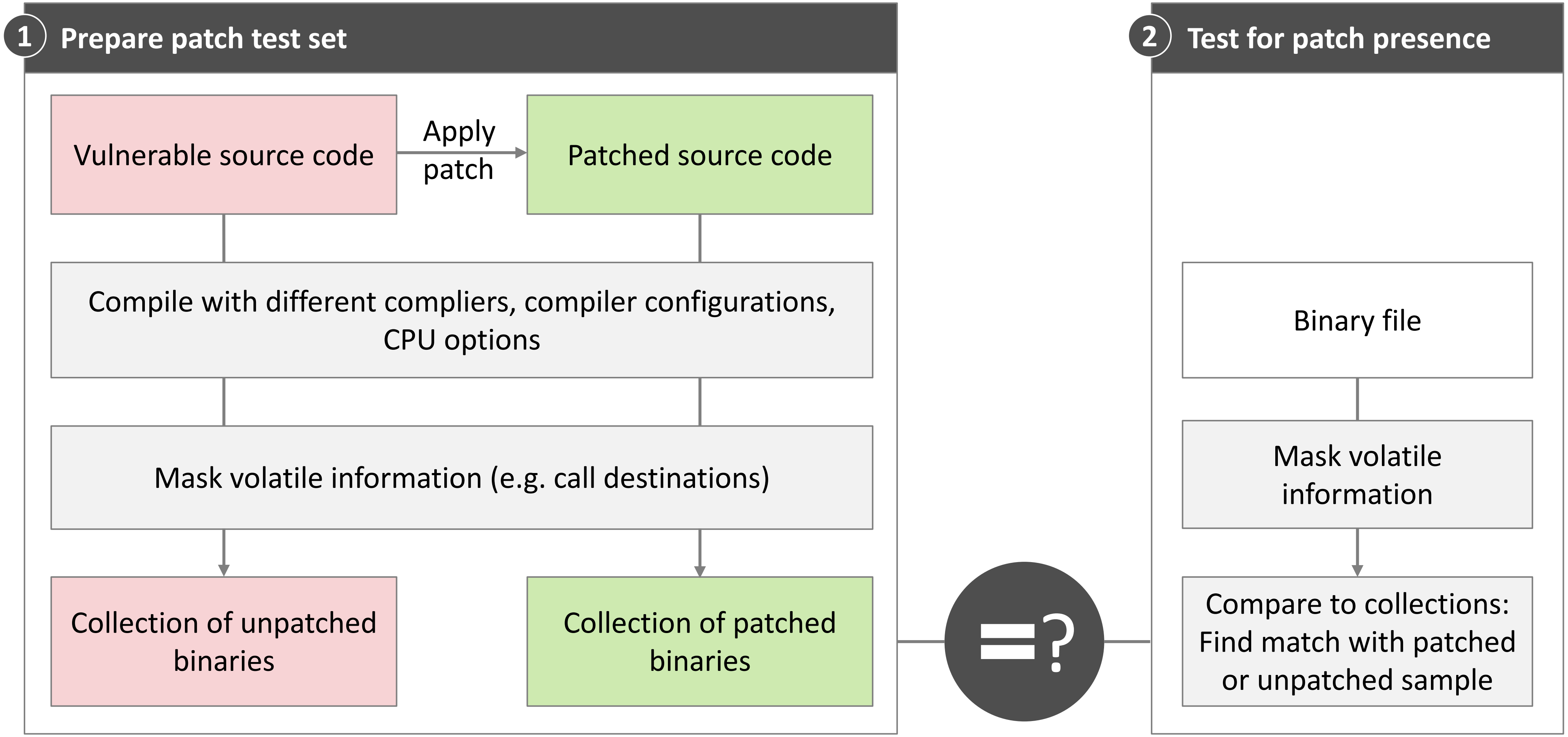
Android userland patch analysis	Android’s 2017 security bulletins list **~280** bugs (~CVEs) with Critical or High severity	Source code is available for **~240** of these bugs	Of these userland bugs, **~180** originate from C/C++ code (plus a few Java)	So far, we implemented heuristics for **164** of the corresponding patches	× The heuristics would optimally work on hundreds of thousands of Android firmwares: – 60,000 Android variants [3] – Regular updates for many of these variants
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 patch gap: Android patching completeness varies widely for different phones



	Patches "missing"	
	Critical	High
Google Pixel 2 Android version 8.1 Patch level: Feb 2018	0	0
Samsung J5 (2016) Android version 7.1.1 Patch level: Aug 2017	0	0
Samsung J3 (2016) Android version 5.1.1 Patch level: Jan 2018	2	10
Wiko Freddy Android version 6.0.1 Patch level: Sep 2017	18	62

Binary-only analysis: Conceptually simple



1 Prepare patch test set

Vulnerable source code

Apply patch

Patched source code

Compile with different compliers, compiler configurations, CPU options

Mask volatile information (e.g. call destinations)

Collection of unpatched binaries

Collection of patched binaries

=?

2 Test for patch presence

Binary file

Mask volatile information

Compare to collections: Find match with patched or unpatched sample

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

```
#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

```
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.

```
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

1

Prepare patch test set

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 <impeg2_buf_mgr_release>   2c: R_AARCH64_CALL26  impeg2_buf_mgr_release
[…]
```

Sanitize instructions
Toss out irrelevant destination addresses of the instruction

Instruction format of the bl instruction

```
100x 01 ii iiii iiii iiii iiii iiii iiii
```

Create hash of remaining binary code

Generate signature containing function length, position/type of relocation entries, and hash of the code

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

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

Signature generation requires many source code trees

- **Hundreds of different Android revisions** (e.g. android-7.1.2_r33)
- **Device-specific source code trees** (From Qualcomm Codeaurora CAF)

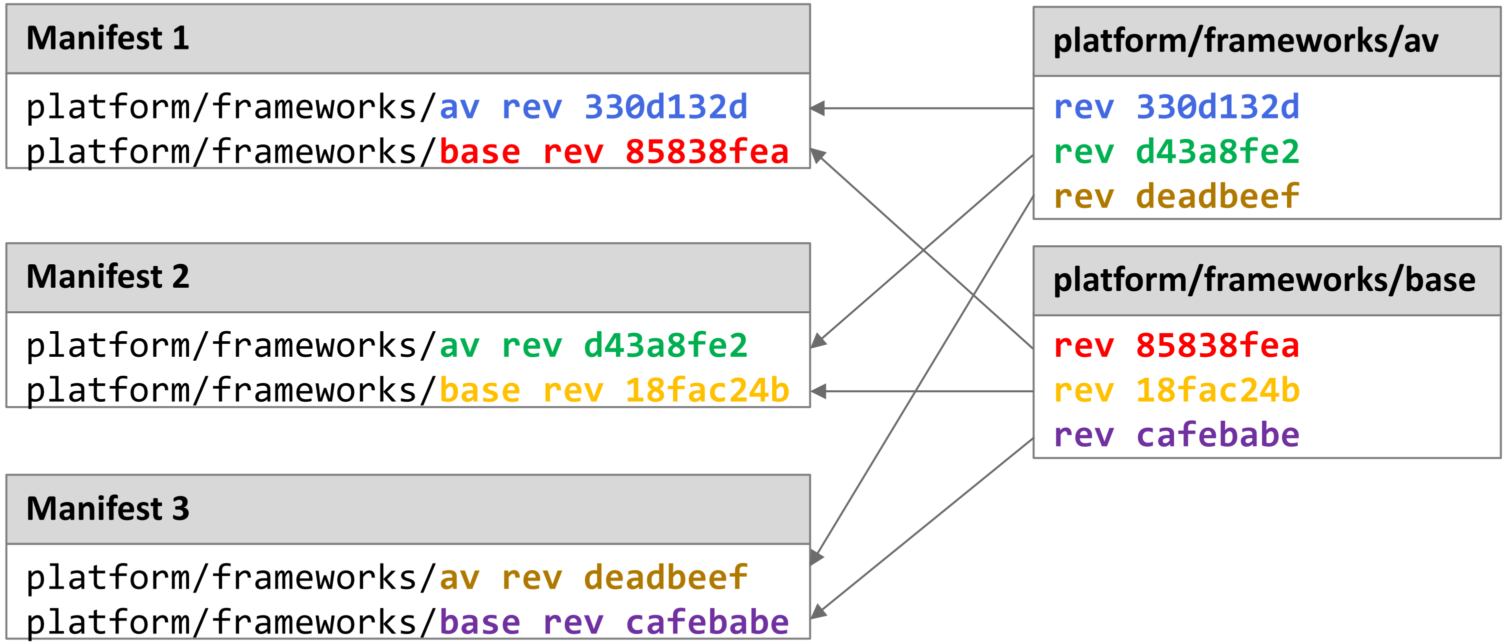
Currently ~1100 source code trees are used in total (many more exist!)

1100 x 50 GiB = 55 TiB

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

We leverage a FUSE (filesystem in userspace) to retrieve files only on demand

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



Manifest 1

```
platform/frameworks/av rev 330d132d
platform/frameworks/base rev 85838fea
```

Manifest 2

```
platform/frameworks/av rev d43a8fe2
platform/frameworks/base rev 18fac24b
```

Manifest 3

```
platform/frameworks/av rev deadbeef
platform/frameworks/base rev cafebabe
```

platform/frameworks/av

```
rev 330d132d
rev d43a8fe2
rev deadbeef
```

platform/frameworks/base

```
rev 85838fea
rev 18fac24b
rev cafebabe
```

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

Using our custom FUSE, we can finally generate a large collection of signatures

1

Prepare patch test set

Mount source code tree

- Read manifest
- Use 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

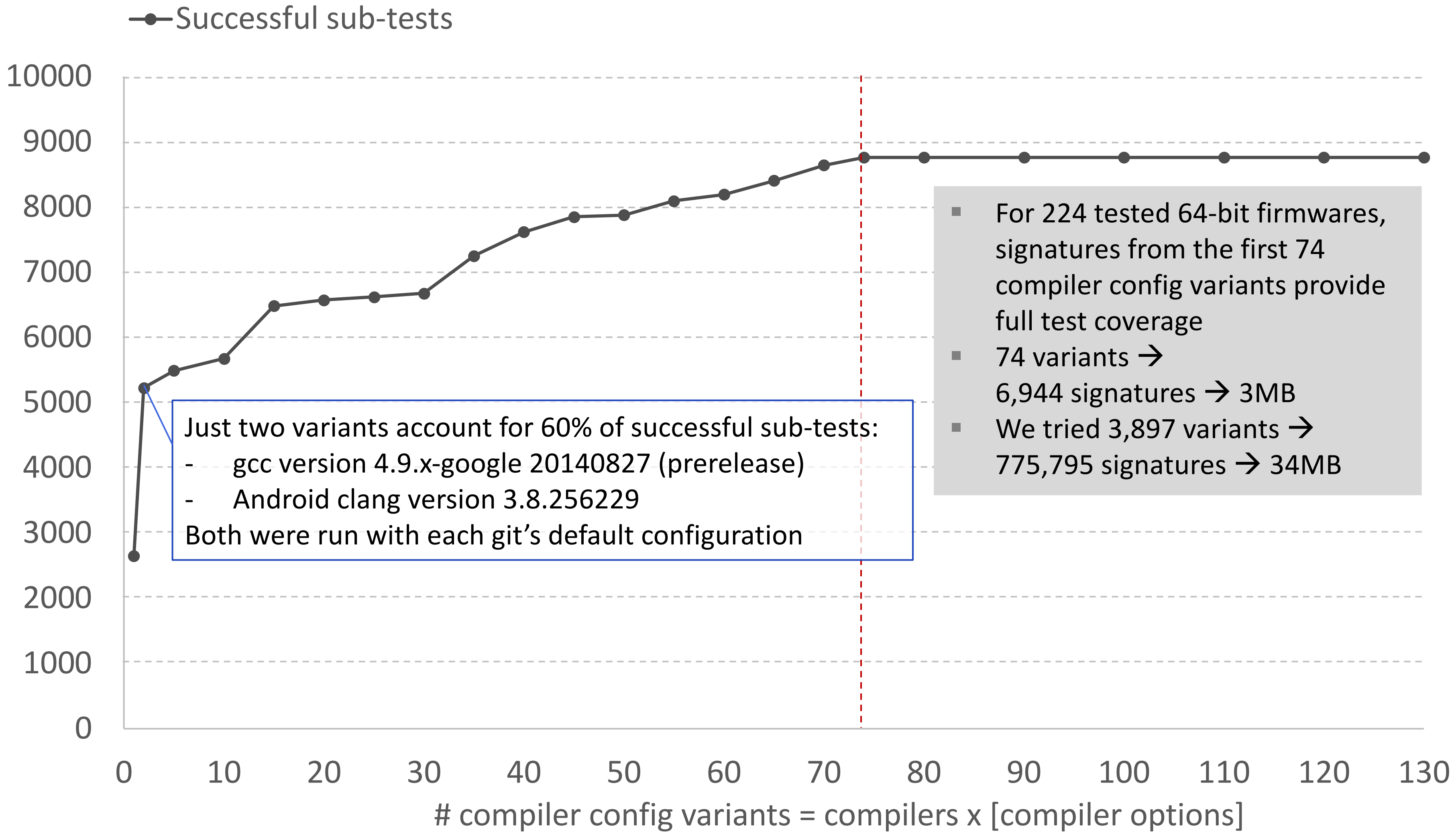
- Evaluate relocation entries and create signatures **for each compiler variant**

Next question: **How many different compiler variants do we need?**

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.



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

- 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

Finding needles in a haystack: What do we do if there is no symbol table?

2

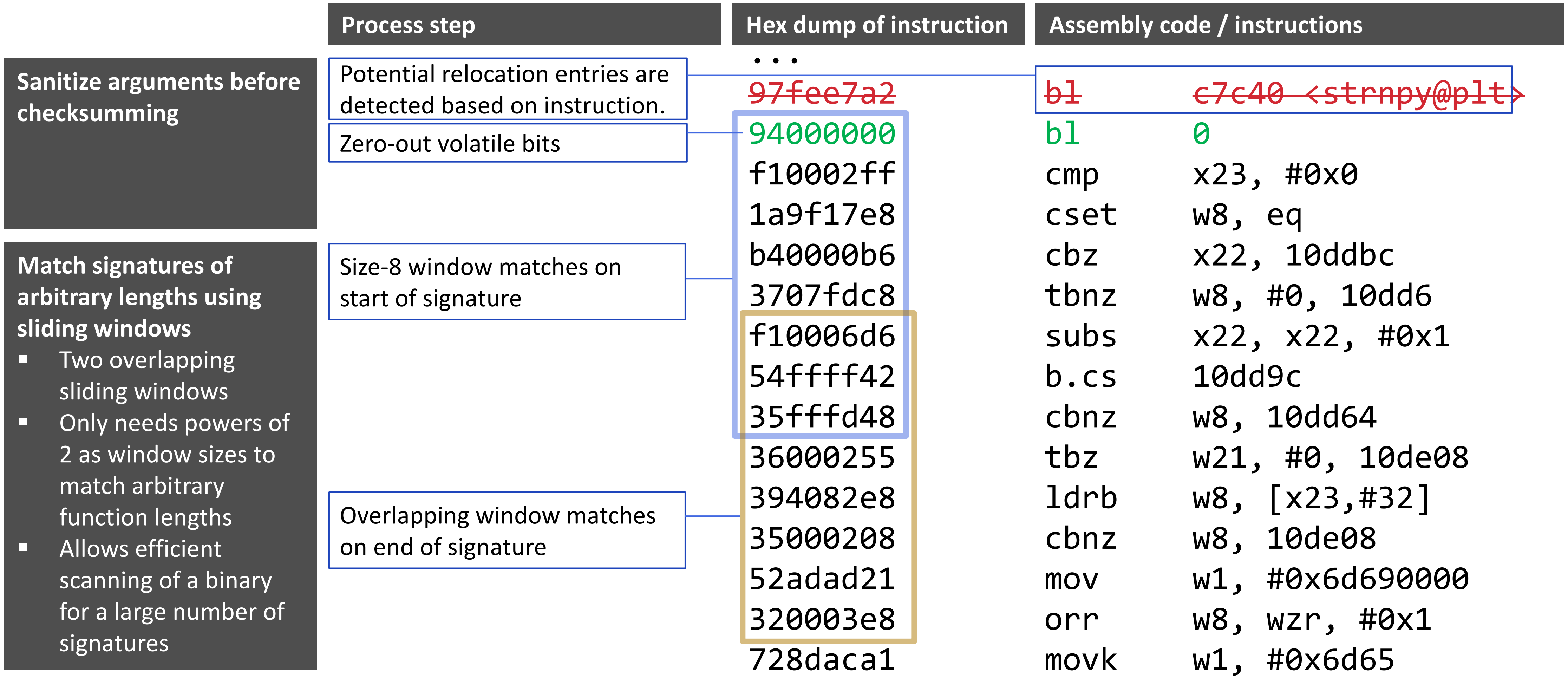
Test for patch presence

Function found in symbol table	Simply compare function with pre-computed samples

Function not in symbol table

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

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



Sanitize arguments before checksumming

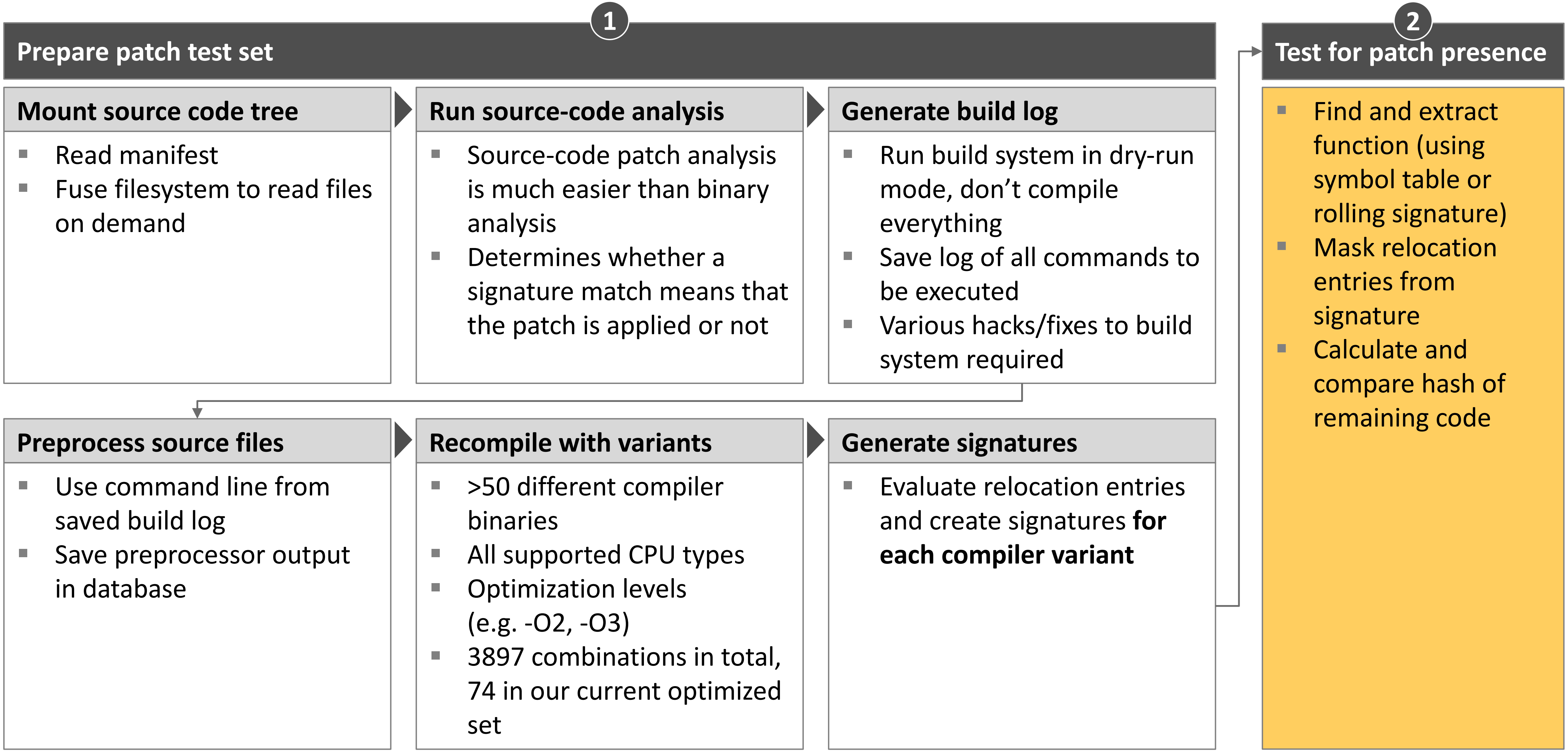
Match signatures of arbitrary lengths using sliding windows

- 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
	. . .	
Potential relocation entries are detected based on instruction.	~~97fee7a2~~	~~bl c7c40 <strnpy@plt>~~
Zero-out volatile bits	94000000	bl 0
	f10002ff	cmp x23, #0x0
	1a9f17e8	cset w8, eq
Size-8 window matches on start of signature	b40000b6	cbz x22, 10ddbc
	3707fdc8	tbnz w8, #0, 10dd6
	f10006d6	subs x22, x22, #0x1
	54ffff42	b.cs 10dd9c
	35fffd48	cbnz w8, 10dd64
	36000255	tbz w21, #0, 10de08
Overlapping window matches on end of signature	394082e8	ldrb w8, [x23,#32]
	35000208	cbnz w8, 10de08
	52adad21	mov w1, #0x6d690000
	320003e8	orr w8, wzr, #0x1
	728daca1	movk w1, #0x6d65

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



1 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**

2 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

Patch gap: Android vendors differ widely in their patch completeness

Vendors differ in how many patches are missing from their phones

Missed patches	Vendor	Samples*
0 to 1	Google	Lots
	Sony	Few
	Samsung	Lots
	Wiko	Few
1 to 3	Xiaomi	Many
	OnePlus	Many
	Nokia	Few
3 to 4	HTC	Few
	Huawei	Many
	LG	Many
	Motorola	Many
More than 4	TCL	Many
	ZTE	Few

Notes
– The tables shows the average number of missing Critical and High severity patches before the claimed patch date
* Samples – Few: 5-9; Many: 10-49; Lots: 50+
– Some phones are included multiple times with different firmwares 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

Some of the patch gap is likely due to chipset vendors forgetting to include them

Missed patches	Chipset	Samples*
< 0.5	Samsung	Lots
1.1	Qualcomm	Lots
1.9	HiSilicon	Many
9.7	Mediatek	Many

Notes
– Again, we show the average of missing High and Critical patches for phones that use these chipsets
– Samsung phones can run on a Samsung or Qualcomm chipset

Agenda

- Research motivation
- Spot the Android patch gap
- **Try to exploit Android phones**

Can we now hack Android phones due to missing patches?

At first glance, Android phones look hackable

- We find that most phones miss patches within their patch level
- While the number of open CVEs can be smaller than the number of missing patches, we expect some vulnerabilities to be open
- Many CVEs talk of “code execution”, suggesting a hacking risk based on what we experience on Windows computers

VS.

Mobile operating systems are inherently difficult to exploit

- Modern exploit mitigation techniques increase hacking effort
- Mobile OSs explicitly distrust applications through sandboxing, creating a second layer of defense
- Bug bounties and Pwn2Own offer relatively high bounties for full Android exploitation

Do criminals hack Android? Very rarely.

Criminals generally use three different methods to compromise Android devices

	Social engineering	Local privilege escalation	Remote compromise
Approach	**Trick user into insecure actions:** - Install malicious app - Then grant permissions - Possibly request 'device administrator' role to hinder uninstallation	- Trick user into installing malicious app - Then exploit kernel-level vulnerability to gain control over device, often using standard "rooting" tools	- Exploit vulnerability in an outside-facing app (messenger, browser) - Then use local privilege escalation
Used for	- Ransomware [File access permission] - 2FA hacks [SMS read] - Premium SMS fraud [SMS send]	- Targeted device compromise, e.g. FinFisher and Crysaor (Same company as infamous Pegasus malware) - Advanced malware	- (Google bug bounty, Pwn2Own)
Frequency in criminal activity	- Almost all Android "Infections" ●●●	- Regular observed in advanced malware and spying ○○●	- Very few examples of recent criminal use ○○○
Made harder through patching	✗	✓ (userland or kernel)	✓ (userland and kernel)

An exploitable vulnerability implies a missing patch, but not the other way around

Missing patches in source code

− Code parts that are ignored during compilation

= **Missed patches in binary**

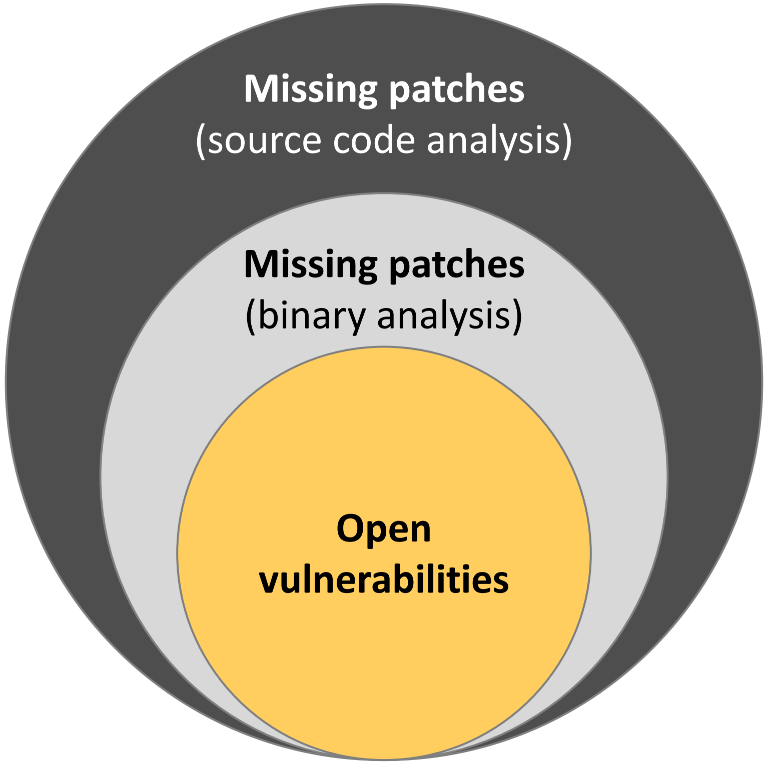
− Vendor created alternative patch

− Vulnerability requires a specific configuration

− Bug is simply not exploitable

− Errors in our heuristic (it happens!)

= **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)

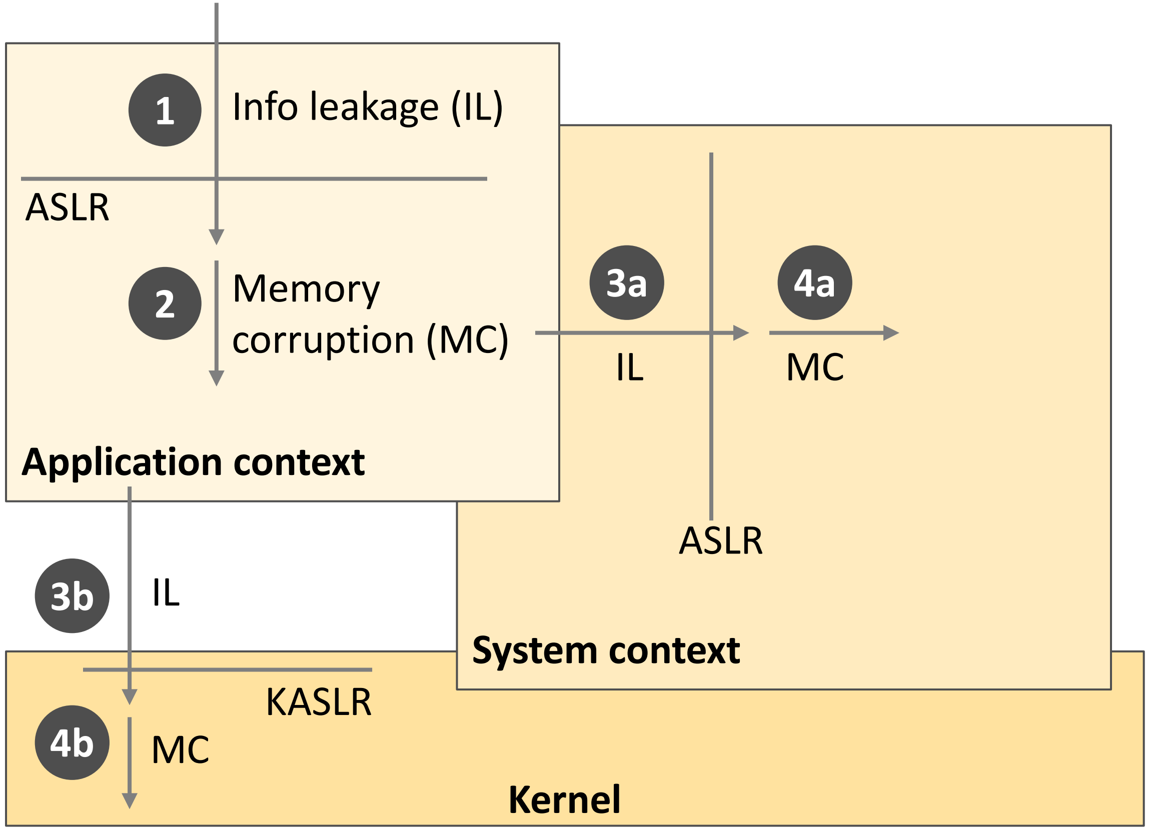
2 **Corrupt memory** in an application. Examples:
- Malicious video file corrupts memory using Stagefright bug
- Malicious web site leverages Webkit vulnerability

➢ This gives an attacker control of the application including the apps access permission

3 4 Do the same again with two more bugs to gain access to system context or kernel

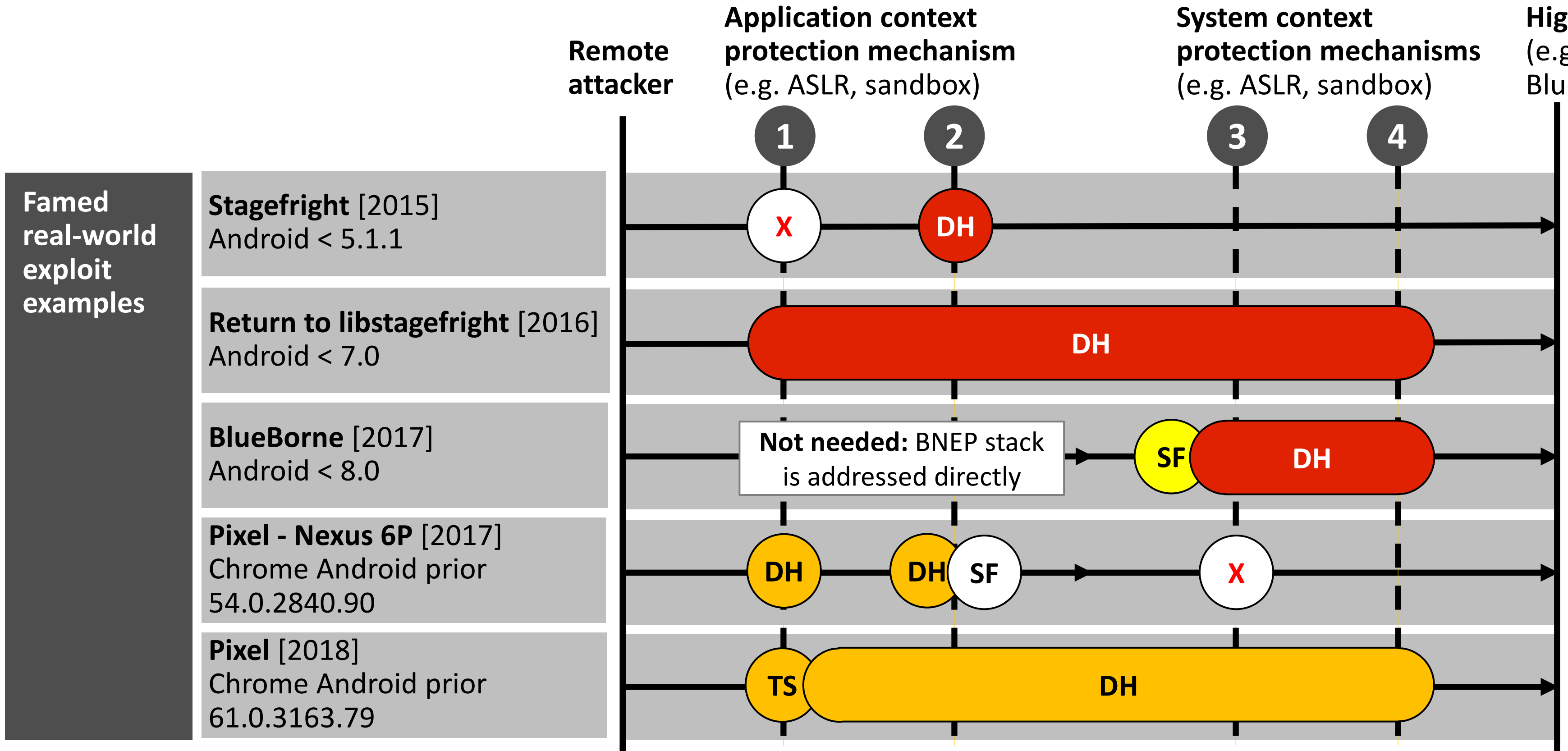
➢ This gives an attacker all possible permissions (system context), or full control over the device (kernel)

Simplified exploit chain examples with 4 bugs



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

Stagefright [2015]
Android < 5.1.1

Return to libstagefright [2016]
Android < 7.0

BlueBorne [2017]
Android < 8.0

Pixel - Nexus 6P [2017]
Chrome Android prior 54.0.2840.90

Pixel [2018]
Chrome Android prior 61.0.3163.79

Step 1: Remote Code Execution and Information disclosure
In many cases, one critical or high-severity weakness is exploited to allow for Remote Code Execution (RCE). (In the special case of BlueBorne, no sandbox exists.)

Step 2: Escalation of Privilege
At least one other weakness **(or the users themselves)** helps the attacker overcome protection mechanisms and gain access to higher privileges

Weakness severities

🔴 **Critical**
🟠 **High**
🟡 **Moderate**

Weakness classes

DH **Data handling errors (CWE-19)** e.g. buffer errors, input validation mistakes

SF **Security features gaps (CWE-254)** e.g. permission errors, privileges mishandling, access control errors

TS **Time and state errors (CWE-361)** e.g. race conditions, incorrect type conversions or casting

In case you want to dive deeper: More details on well-documented Android exploit chains

Famed real-world exploit examples

	1	2	3	4
Return to libstagefright 2016	Heap pointer leak to bypass ASLR protection	ROP execution in mediaserver process	Module pointer leak to get address of executable code	Call mprotect to get RCE into privileged system-server domain **DH**
BlueBorne 2017	BlueBorne is a vulnerability in the Android Bluedroid/Fluorid userland stack, which is already a high-privileged domain	**Not needed**	Information leak vulnerability leaks arbitrary data from the stack, which allows an attacker to derive ASLR base address for a bypass **SF**	Trigger memory corruption in BNEP service that enables an attacker to execute arbitrary code in the high privileged Bluetooth domain **DH**
Pixel / Nexus 6P 2017	Content view client in Chrome allowed arbitrary intent scheme opening, which allows escaping the Chrome sandbox **DH**	Attacker perform arbitrary read/write operations leading to code execution based on incorrect optimization assumption in Chrome v8 **DH**	Open intent controlled URL in Google Drive to get shell in untrusted app context **SF**	Exploit chain does not include break-out of untrusted app context **X**
Pixel 2018	Chrome V8 bug to get RCE in sandbox using a OOB bug in GetFirstArgument-AsBytes function **TS**	Use map and unmap mismatch in libgralloc to escape Chrome sandbox and inject arbitrary code into system-server domain by accessing a malicious URL in Chrome **DH**		

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

GET IT ON Google Play Search: *SnoopSnitch*

SnoopSnitch
App ID: 5be9c979

Android patch level analysis

102

Network Security | Done - You may want to test both 2G and 3G | Start new

Provider results in comparison: | Intercept | Impersonation

Your test result
Your network T-Mobile
Vodafone

Last analysis: 11 Apr 2018 14:10:09

Android patch level analysis
App ID: 5be9c979

Start test

Patched	102
Patch missing	3
After claimed patch level	0
Test inconclusive	15
Not affected	0

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

Take aways

- **Android patching is more complicated and less reliable than a single patch date may suggest**
- **Remote Android exploitation is also more much complicated than commonly thought**
- **You can finally check your own patch level thanks to binary-only analysis, and the app SnoopSnitch**

Many thanks to Ben Schlabs, Stephan Zeisberg, Jonas Schmid, Mark Carney, Luas Euler, and Patrick Lucey!

Questions?

Jakob Lell <jakob@srlabs.de>
Karsten Nohl <nohl@srlabs.de>

References

1. Federal Trade Commision, **Mobile Security Updates: Understanding the Issues**, February 2018 https://www.ftc.gov/system/files/documents/reports/mobile-security-updates-understanding-issues/mobile_security_updates_understanding_the_issues_publication_final.pdf
2. Duo Labs Security Blog, **30% of Android Devices Susceptible to 24 Critical Vulnerabilities**, June 2016 https://duo.com/decipher/thirty-percent-of-android-devices-susceptible-to-24-critical-vulnerabilities
3. Google, **Android Security 2017 Year In Review**, March 2018 https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf