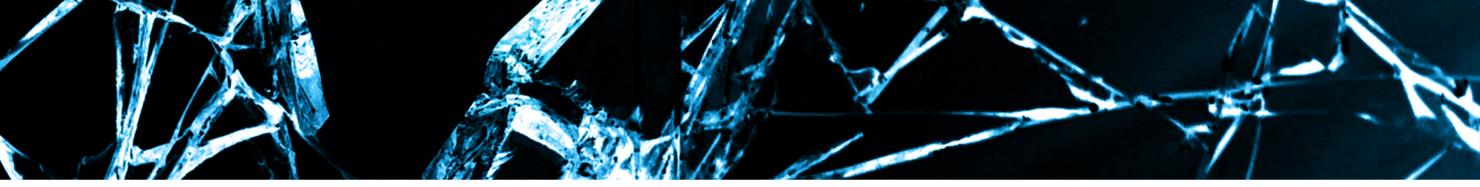


Adaptive Kernel Live Patching: An Open Collaborative Effort to Ameliorate Android N-day Root Exploits

Yulong Zhang and Lenx (Tao) Wei

Baidu X-Lab

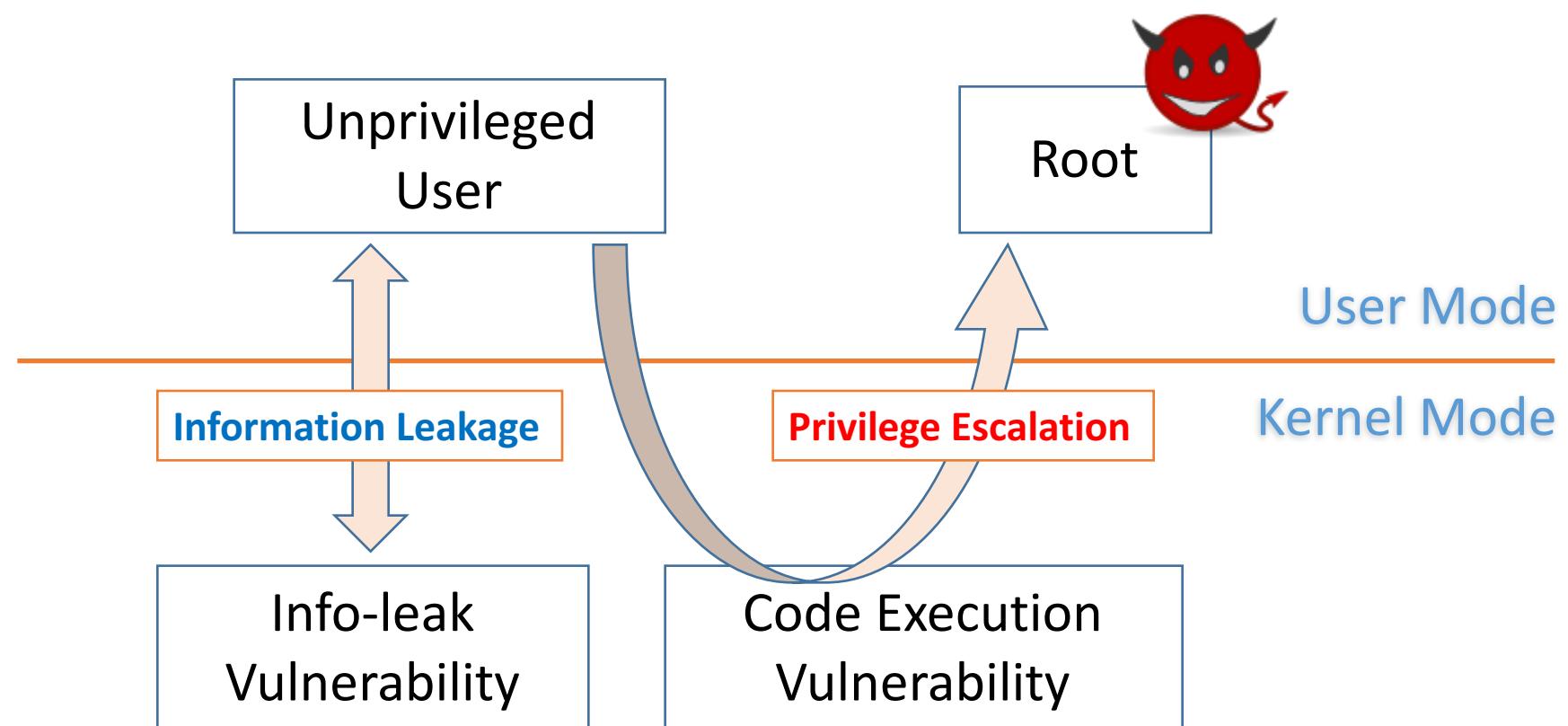
August 2016

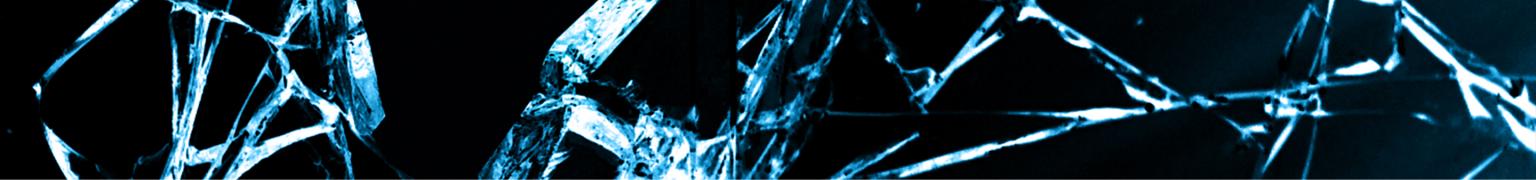


Agenda

- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - LuaKpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem

Threats of Kernel Vulnerabilities

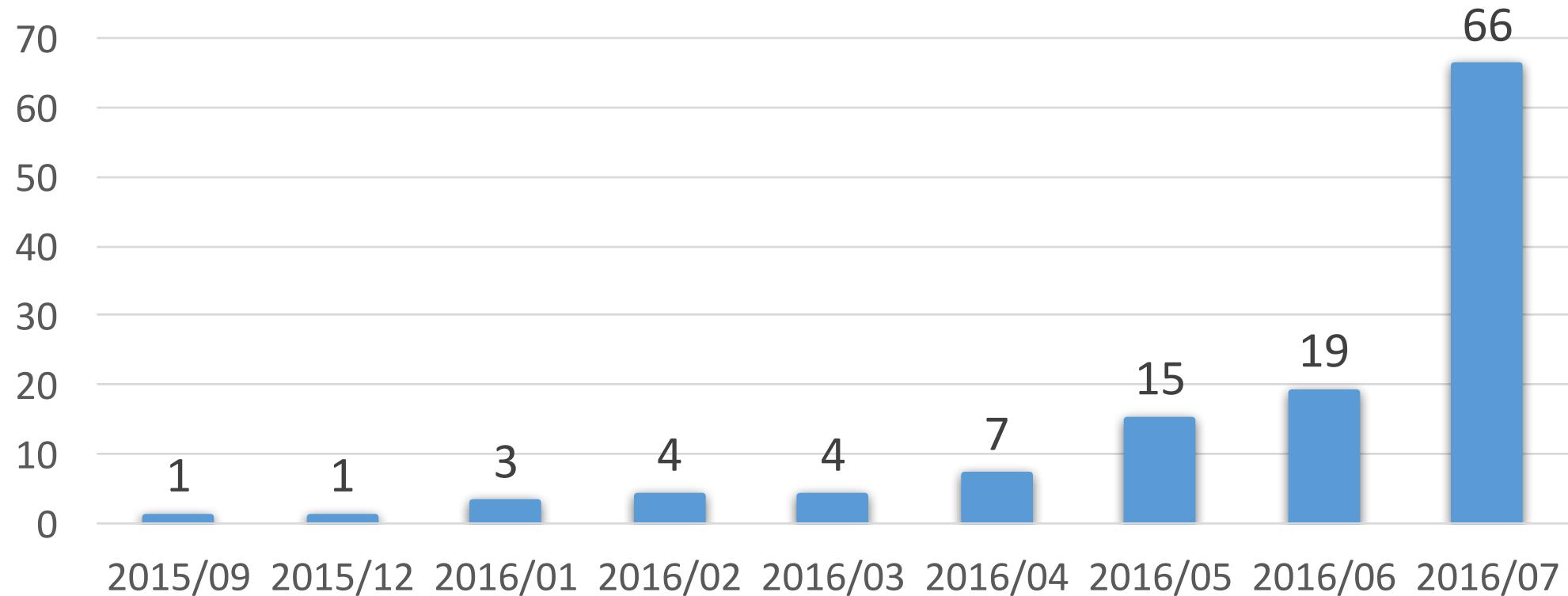




Threats of Kernel Vulnerabilities

- Most security mechanisms relying on kernel integrity/trustworthiness will be broken
 - Access control, app/user isolation
 - Payment/fingerprint security
 - KeyStore
 - Other Android user-land security mechanisms
- TrustZone will also be threatened
 - Attack surfaces exposed
 - Not enough input validation

Monthly Disclosed Number of Android Kernel Vulnerabilities



The Growing Trend Indicates

Month	Count
2015/09	1
...	...
2015/12	1
2016/01	3
2016/02	4
2016/03	4
2016/04	7
2016/05	15
2016/06	19
2016/07	66



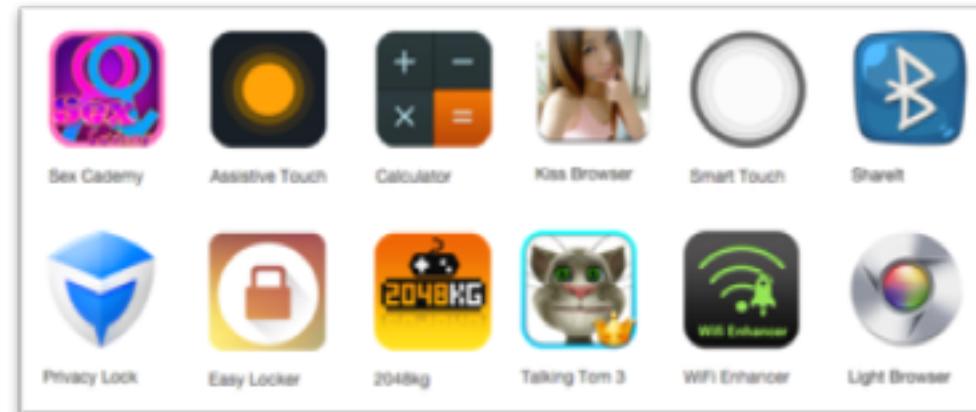
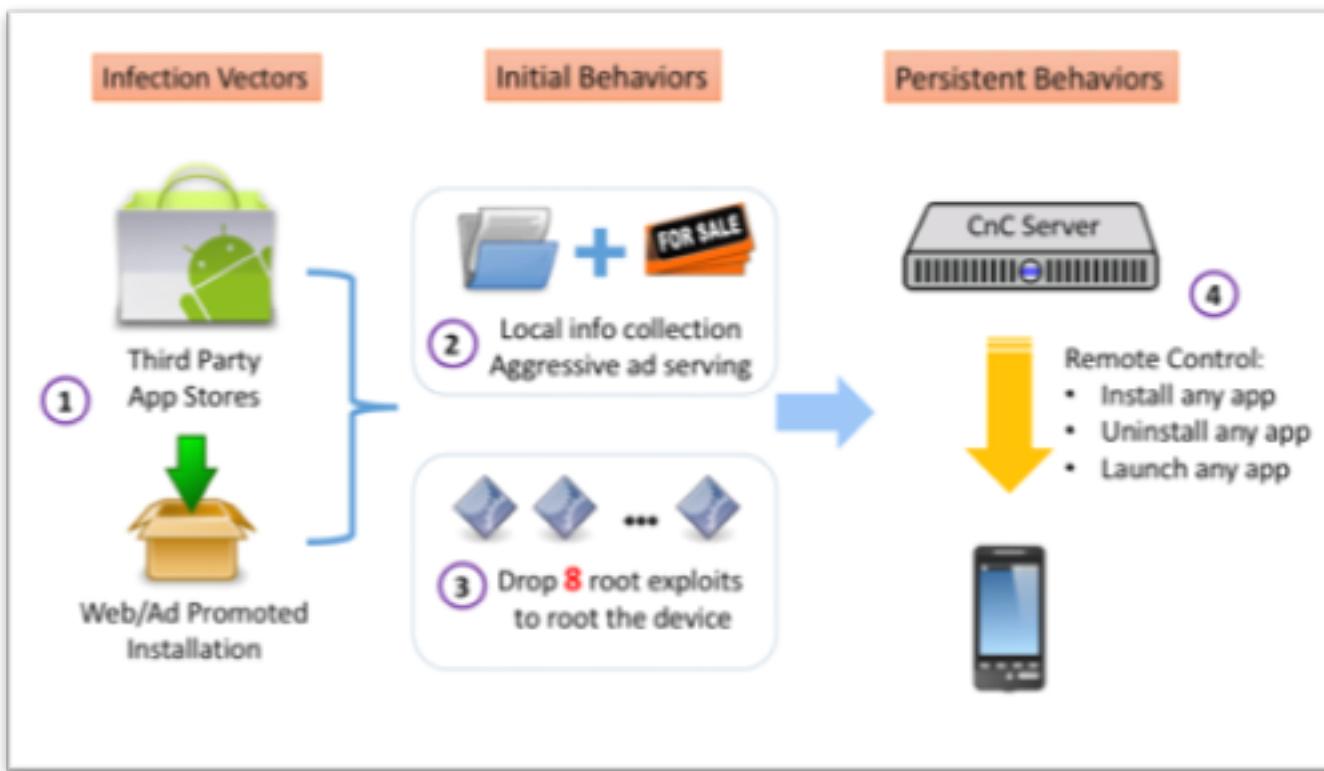
- More and more attentions are drawn to secure the kernel
- More and more vulnerabilities are in the N-Day exploit arsenal for the underground businesses



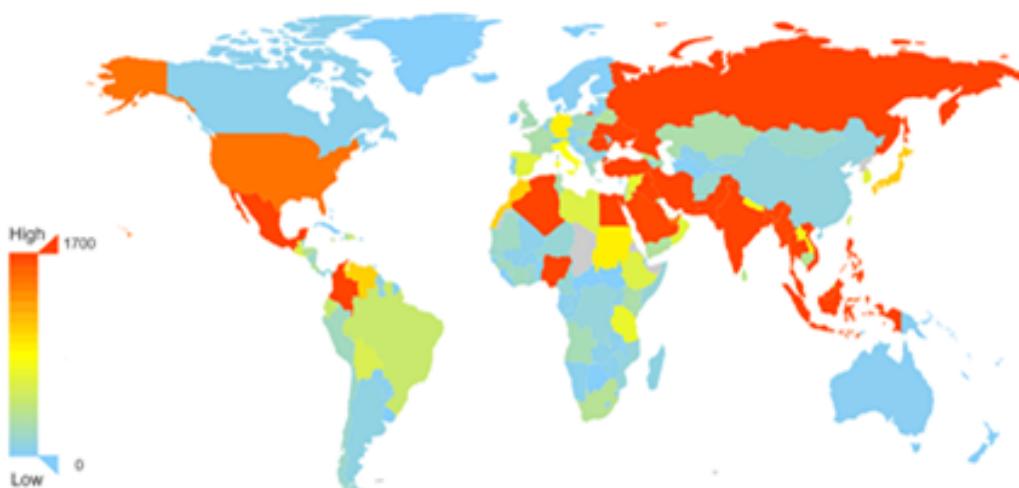
Many Vulnerabilities Have Exploit PoC Publicly Disclosed

Vulnerability/Exploit Name	CVE ID
mempodipper	CVE-2012-0056
exynos-abuse/Framaroot	CVE-2012-6422
diagexploit	CVE-2012-4221
perf_event_exploit	CVE-2013-2094
fb_mem_exploit	CVE-2013-2596
msm_acdb_exploit	CVE-2013-2597
msm_cameraconfig_exploit	CVE-2013-6123
get/put_user_exploit	CVE-2013-6282
futex_exploit/Towelroot	CVE-2014-3153
msm_vfe_read_exploit	CVE-2014-4321
pipe exploit	CVE-2015-1805
Ping Pong Root	CVE-2015-3636
f2fs_exploit	CVE-2015-6619
prctl_vma_exploit	CVE-2015-6640
keyring_exploit	CVE-2016-0728
.....

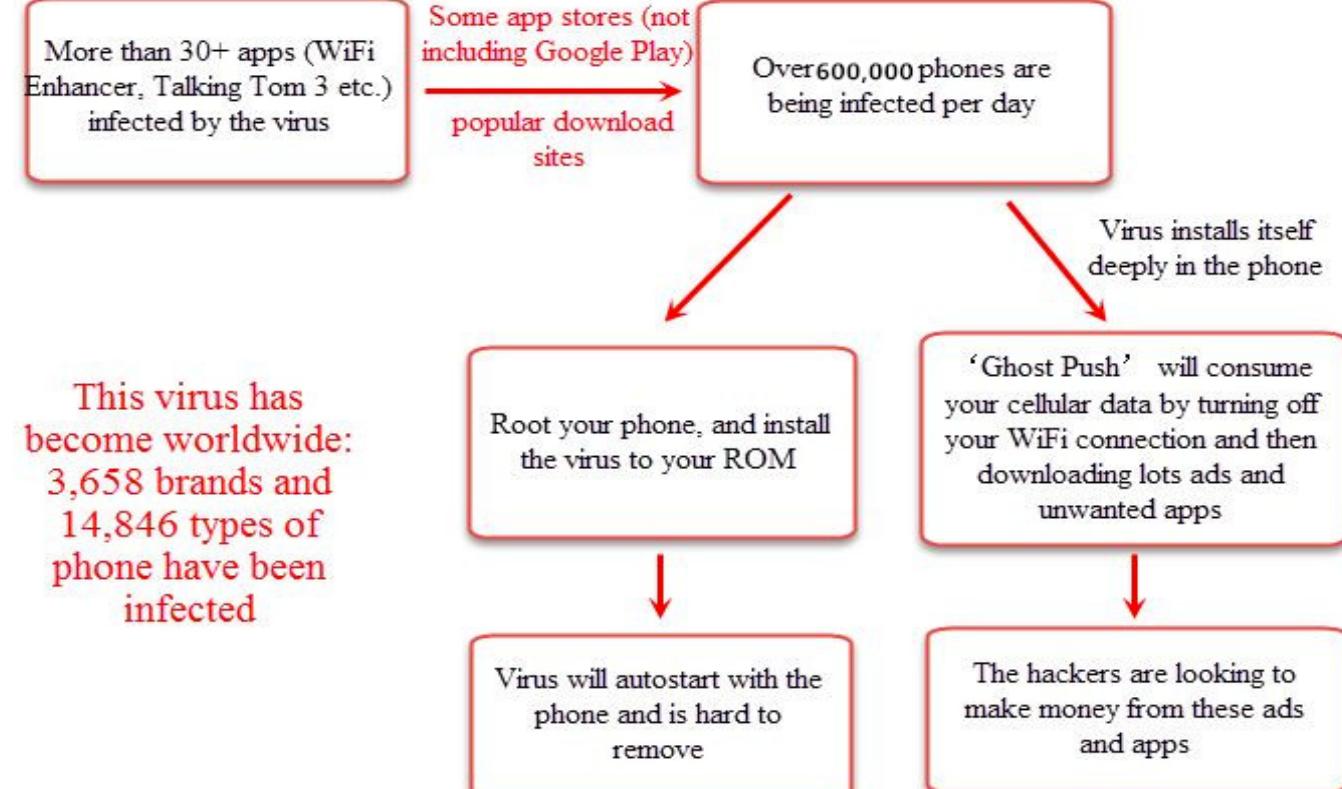
KEMOGE



GHOSTPUSH



This virus has become worldwide:
3,658 brands and 14,846 types of phone have been infected



DOGSPECTUS



“... the payload of that exploit, a Linux ELF executable named module.so, contains the code for **the futex or Towelroot exploit** that was first disclosed at the end of 2014.”

HUMMINGBAD

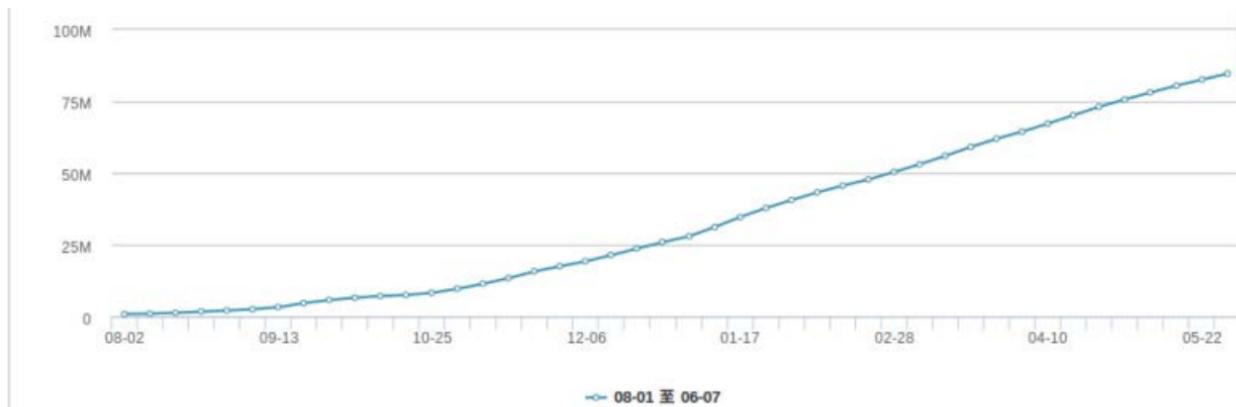
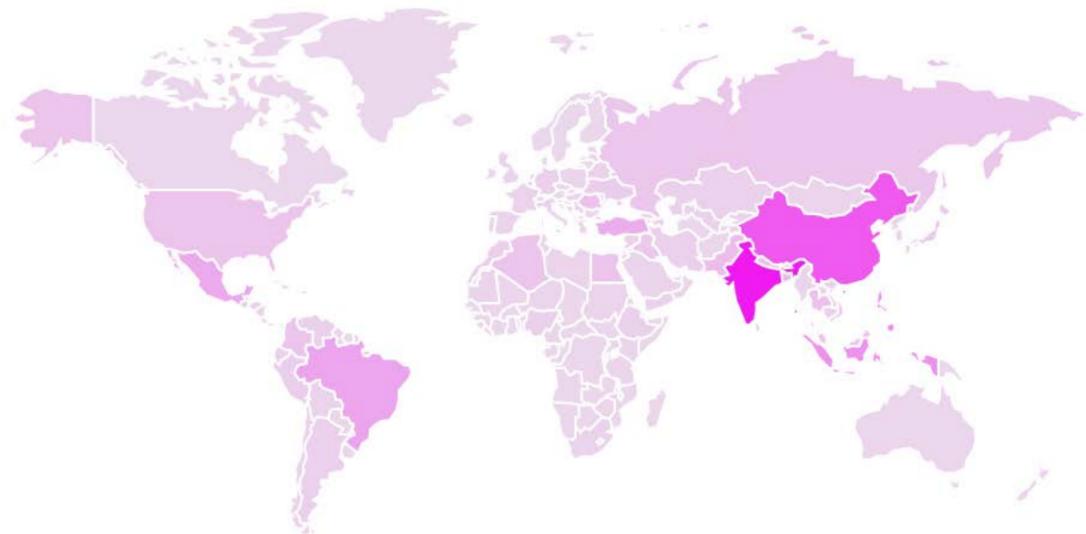


Figure 6: Cumulative Users Over Time



“All combined, the campaign includes nearly 85 million devices...
HummingBad attempts to gain root access on a device with a rootkit that
exploits multiple vulnerabilities... It tries to root thousands of devices
every day, with hundreds of these attempts successful.”

iOS More Secure?



?

A large red question mark centered between the Apple and Android icons.



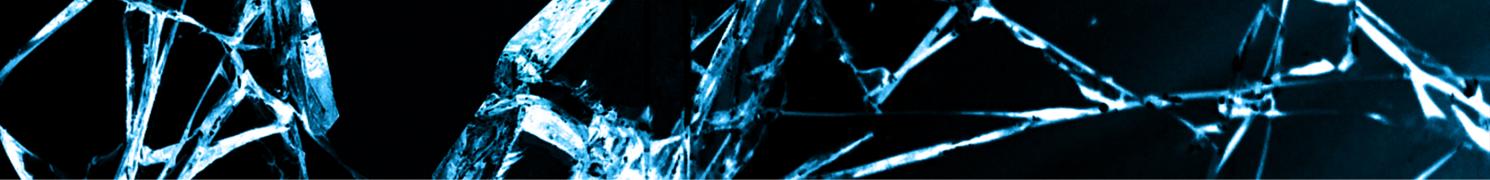
V.S.



iOS Version	Release Date	Kernel Vulnerability #	Android # In This Period
8.4.1	8/13/15	3	-
9	9/16/15	12	1
9.1	10/21/15	6	-
9.2	12/8/15	5	1
9.2.1	1/19/16	4	3
9.3	3/21/16	9	8
9.3.2	5/16/16	11	22



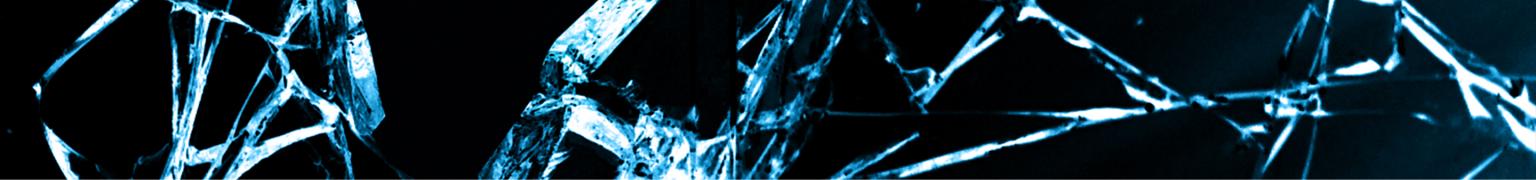
So the problem is: *Android has MORE vulnerabilities*
Vulnerabilities remain UNFIXED over a long time



Agenda

- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - LuaKpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem

- If Apple wants to patch a vulnerability
 - Apple controls the entire (mostly) supply chain
 - Apple has the source code
 - Apple refuses to sign old versions, forcing one-direction upgrade
 - All the iOS devices will get update in a timely manner
- Android
 - Many devices stay unpatched forever/for a long period...



Why Are Android Kernel Vulnerabilities Long Lasting?

- The long patching chain delays the patch effective date
- Fragmentation makes it challenging to adapt the patches to all devices
- Capability mismatching between device vendors and security vendors

Cause A: The long patching chain

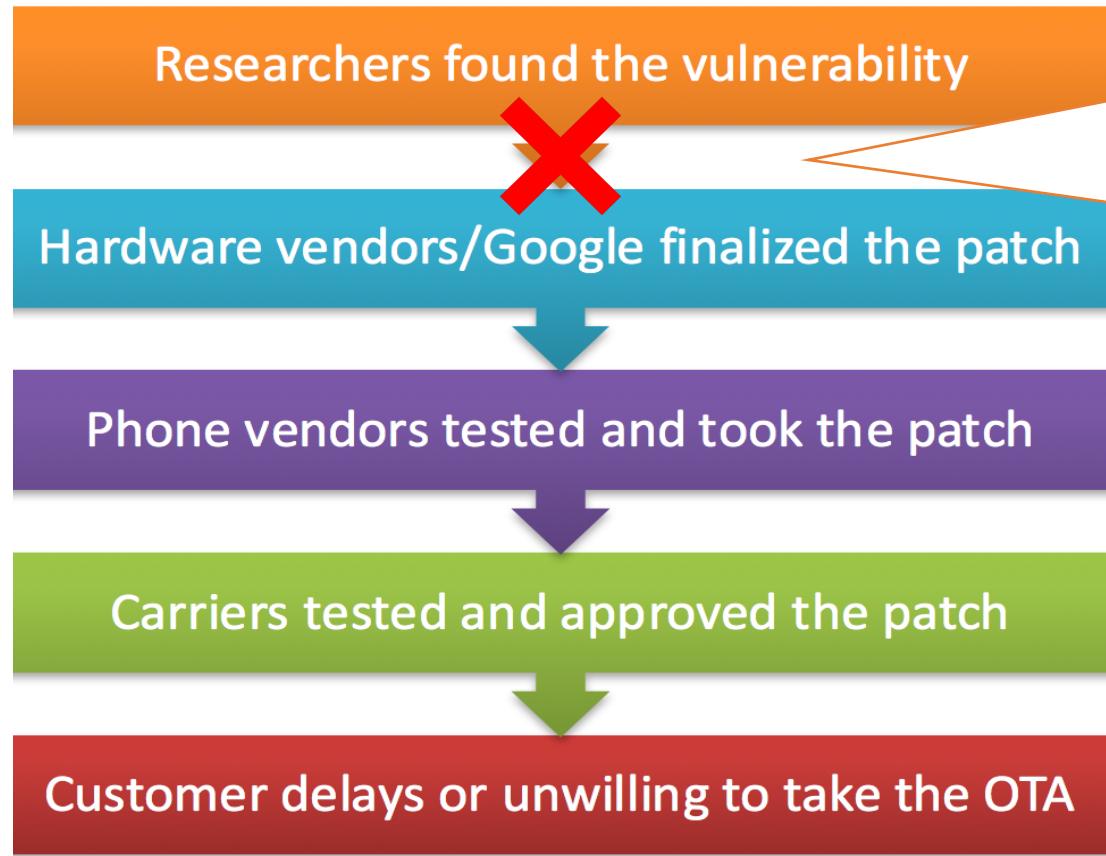
Researchers found the vulnerability

Hardware vendors/Google finalized the patch

Phone vendors tested and took the patch

Carriers tested and approved the patch

Customer delays or unwilling to take the OTA



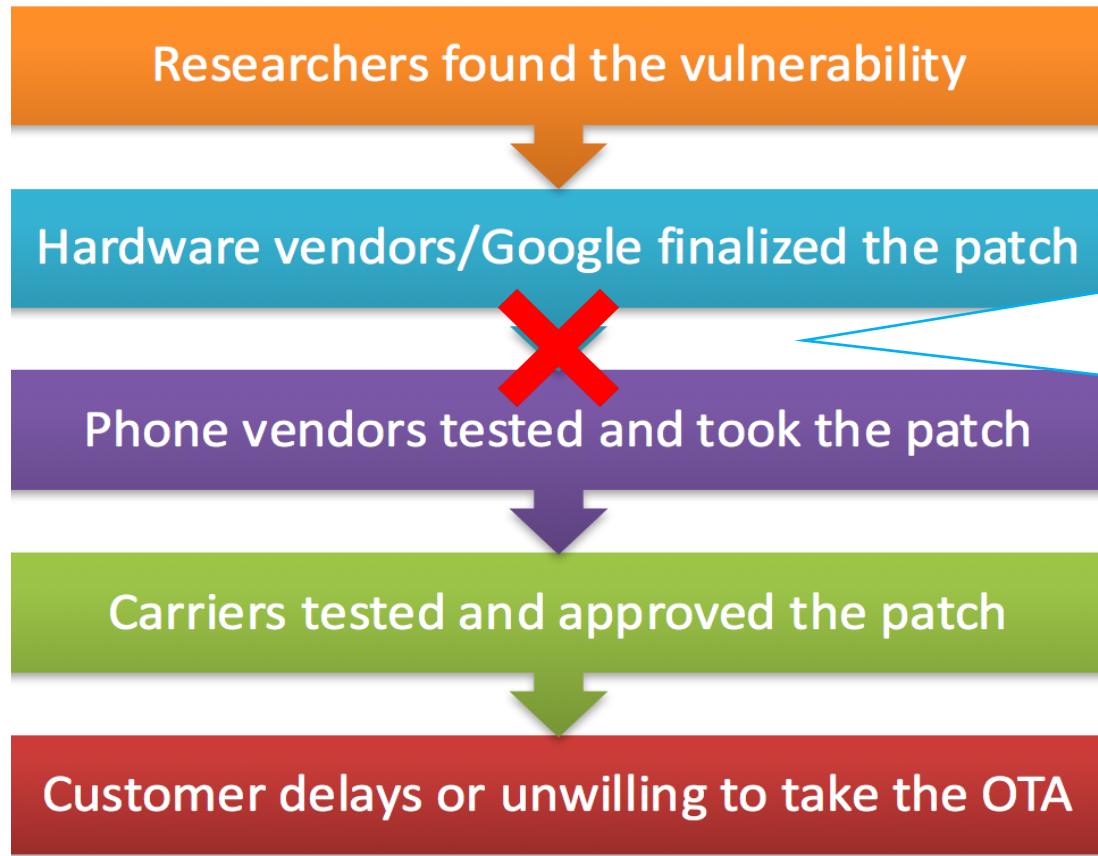
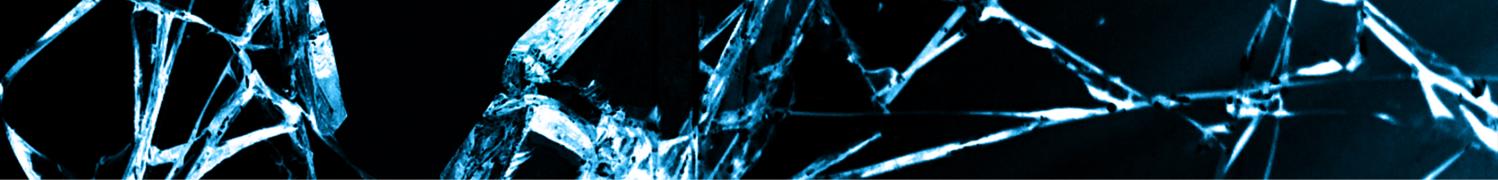
There are exploits appeared in public but

- Never got officially reported to vendors

Exploits made public but not reported

“... We are able to identify at least **10** device driver exploits (from a famous root app) that are **never reported** in the public...”

Android Root and its Providers: A Double-Edged Sword
H. Zhang, D. She, and Z. Qian, CCS 2015



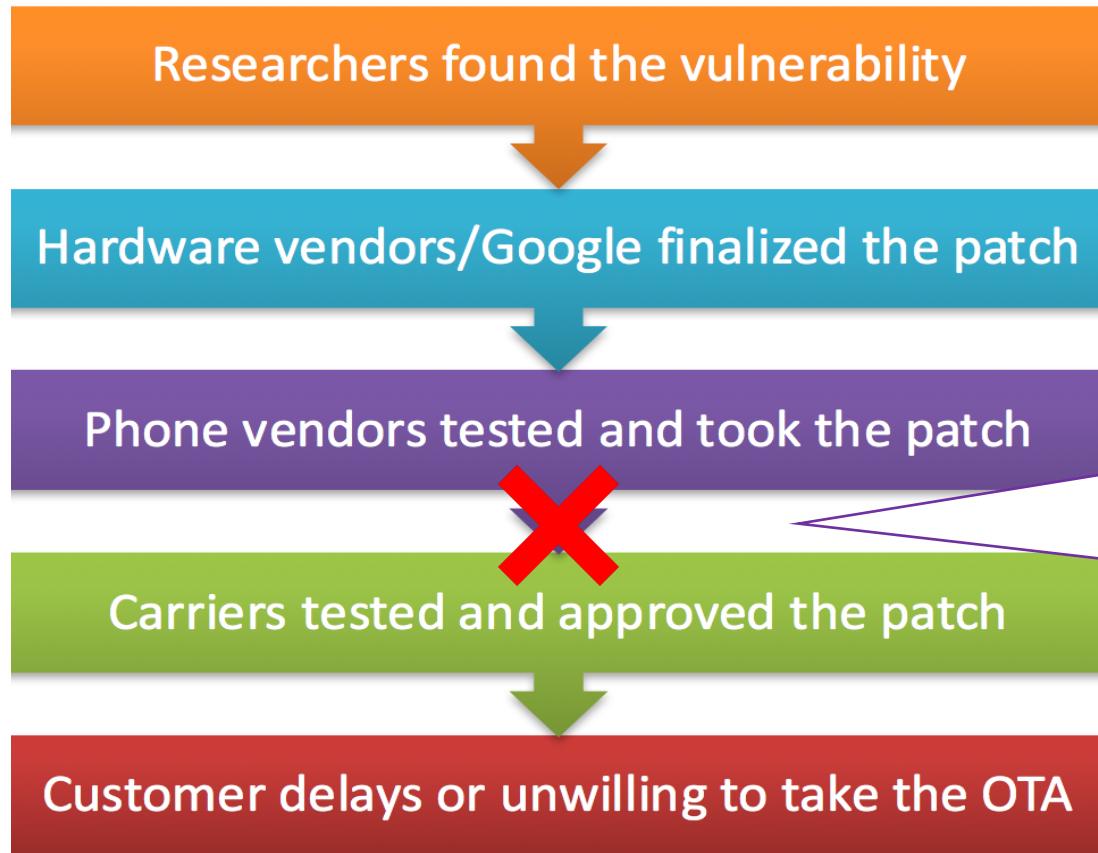
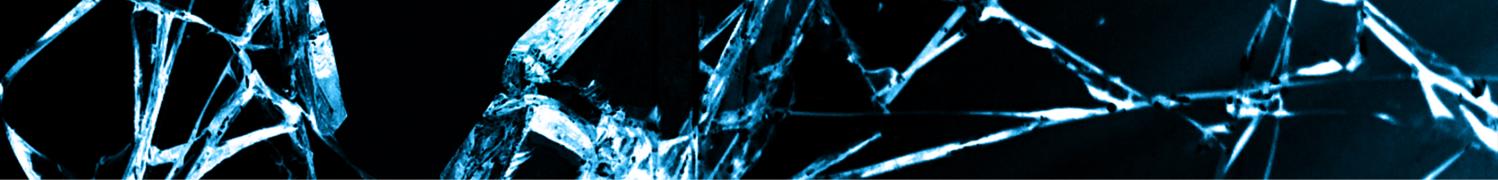
There are exploits disclosed but

- Not getting timely patches

Exploits disclosed but not timely patched

Note that this patch was not applied to all msm branches at the time of the patch release (July 2015) and no security bulletin was issued, so the majority of Android kernels based on 3.4 or 3.10 are still affected despite the patch being available for 6 months.

<https://bugs.chromium.org/p/project-zero/issues/detail?id=734&can=1&sort=-id>



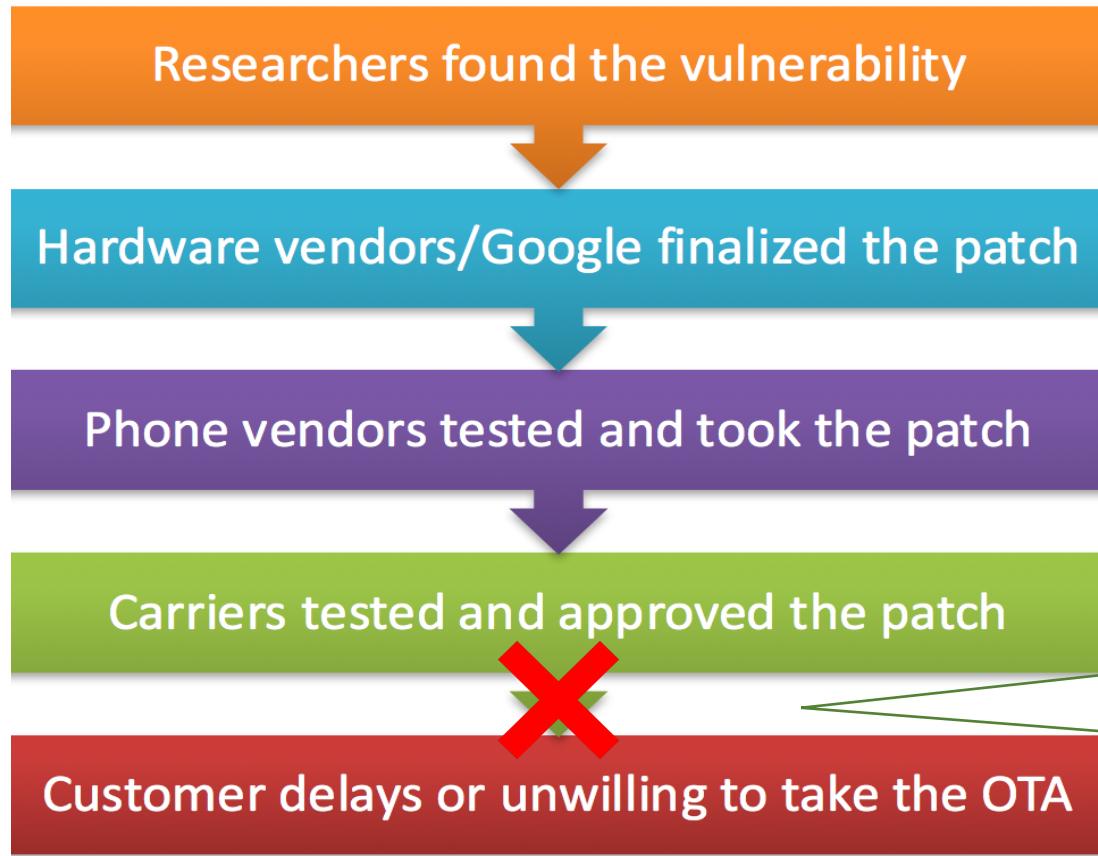
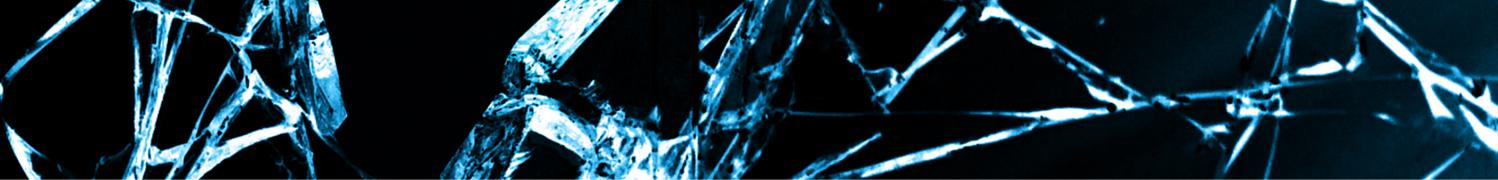
There are exploits
patched but

- Delayed by the carriers

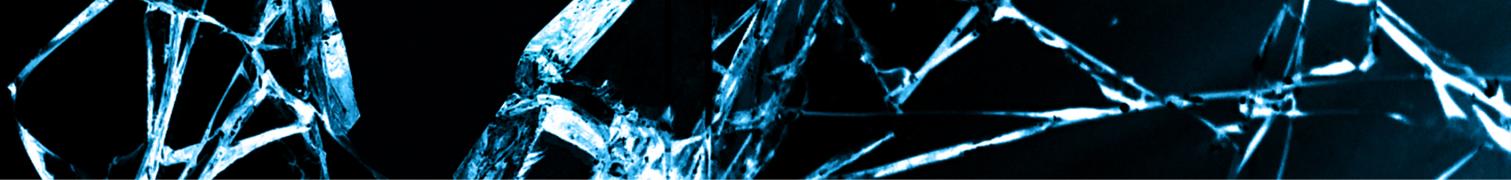
Exploits patched but delayed by carriers

“... It’s each carrier’s job to test all the different updates for all their different smartphones, and they may take **many months** to do so. They may even **decline** to do the work and **never release** the update...”

<http://www.howtogeek.com/163958/why-do-carriers-delay-updates-for-android-but-not-iphone>



User delays the OTA due
to rebooting



Why Are Android Kernel Vulnerabilities Long Lasting?

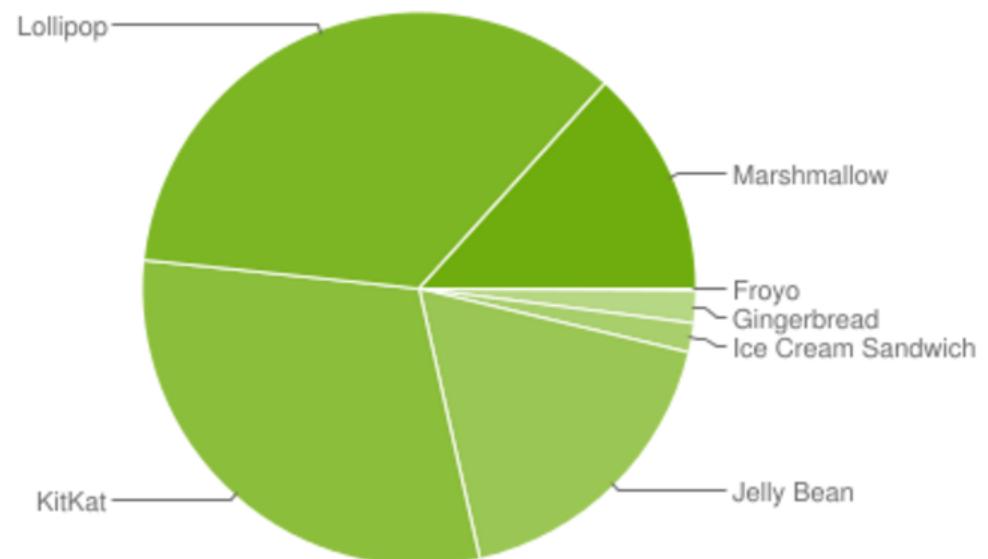
- The long patching chain delays the patch effective date
- Fragmentation makes it challenging to adapt the patches to all devices
- Capability mismatching between device vendors and security vendors

Cause B: Fragmentation



Google Dashboard (2016/07/21)

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.x	Gingerbread	10	1.9%
4.0.x	Ice Cream Sandwich	15	1.7%
4.1.x	Jelly Bean	16	6.4%
4.2.x		17	8.8%
4.3		18	2.6%
4.4	KitKat	19	30.1%
5.0	Lollipop	21	14.3%
5.1		22	20.8%
6.0	Marshmallow	23	13.3%



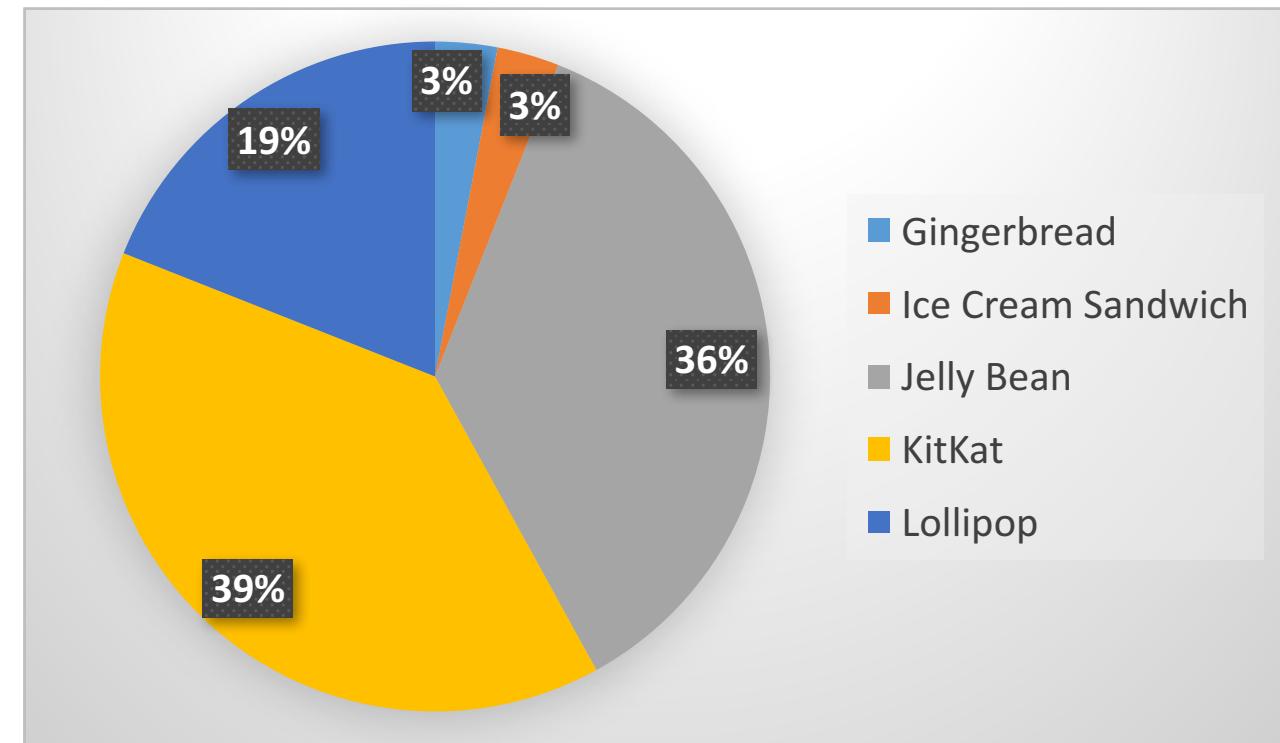
Lollipop was released in November 12, 2014, but
51.6% of the devices are still older than that!

Google stopped patching for Android older than 4.4,
but **21.5%** of the devices are still older than that!

Chinese Market Is Even Worse

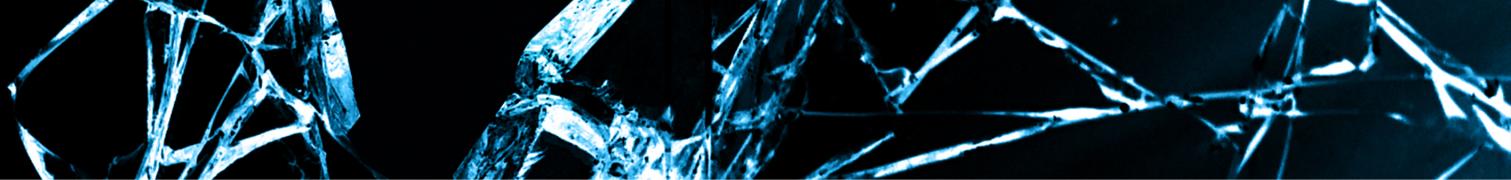
(Stats from devices with Baidu apps installed, July 2016)

Version	Codename	Rate
2.3.x	Gingerbread	3%
4.0.x	Ice Cream Sandwich	3%
4.1.x	Jelly Bean	36%
4.2.x		
4.3	KitKat	39%
4.4		
5	Lollipop	19%
5.1		



Lollipop was released in November 12, 2014, but
80% of the devices are still older than that!

42% of the devices are <4.4!



Why Are Android Kernel Vulnerabilities Long Lasting?

- The long patching chain delays the patch effective date
- Fragmentation makes it challenging to adapt the patches to all devices
- Capability mismatching between device vendors and security vendors

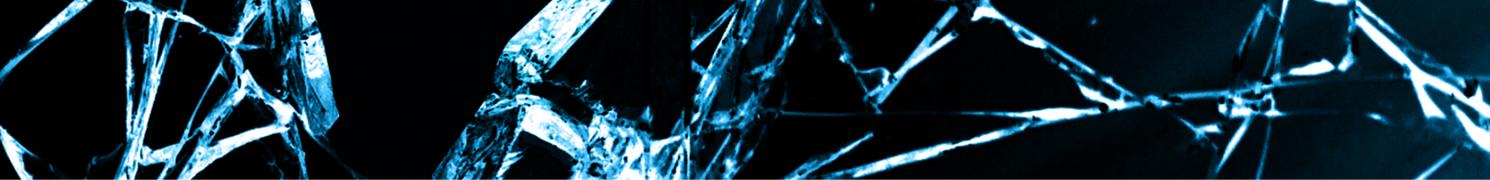
Phone Vendors:

- Privileged to apply the patches
- With source code, easy to adapt the patches
- Not enough resources to discover and patch vulnerabilities



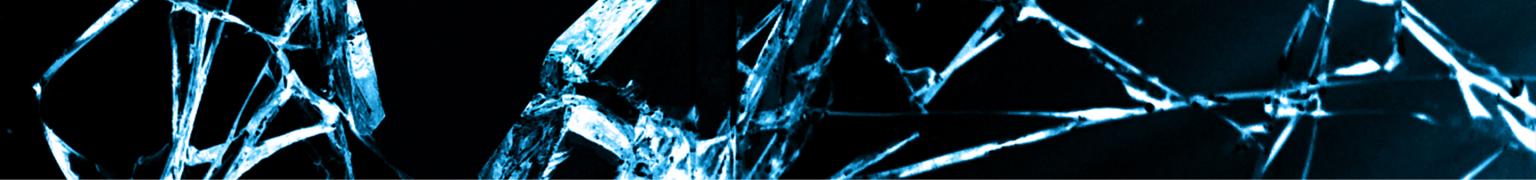
Security Vendors:

- Capable to discover and patch vulnerabilities
- Not privileged enough
- Without source code, difficult to adapt the patches

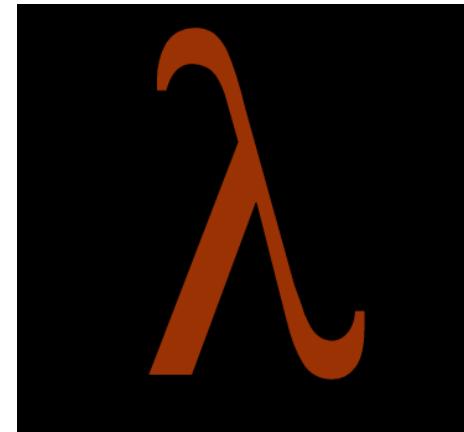


Agenda

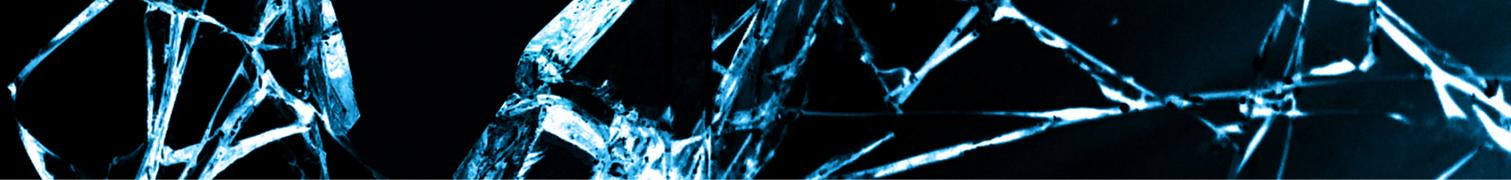
- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - LuaKpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem



CVE-2014-3153 (Towelroot)



- The `futex_requeue` function in `kernel/futex.c` in the Linux kernel through 3.14.5 does not ensure that calls have two different futex addresses, which allows local users to gain privileges.



CVE-2015-3636 (PingPong Root)

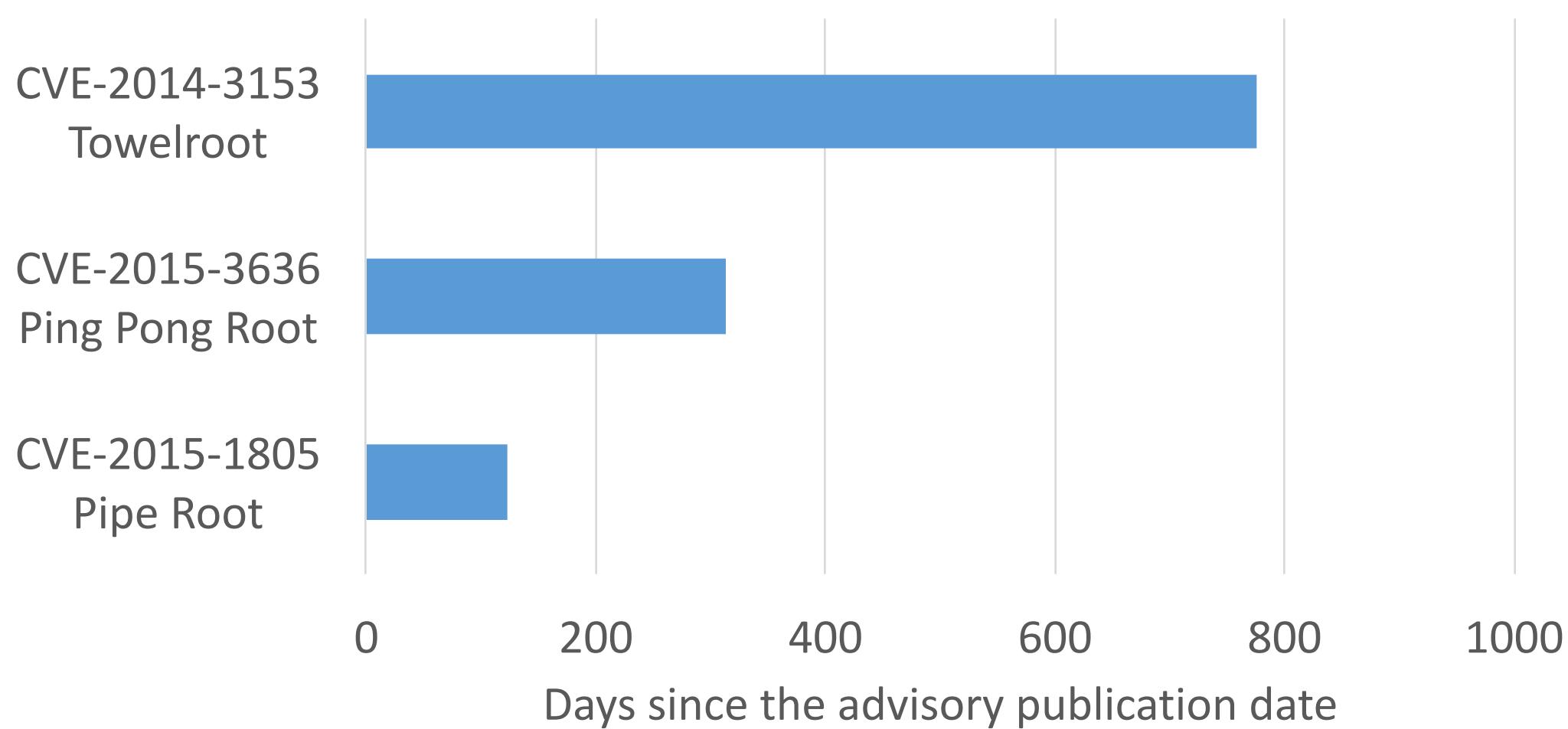


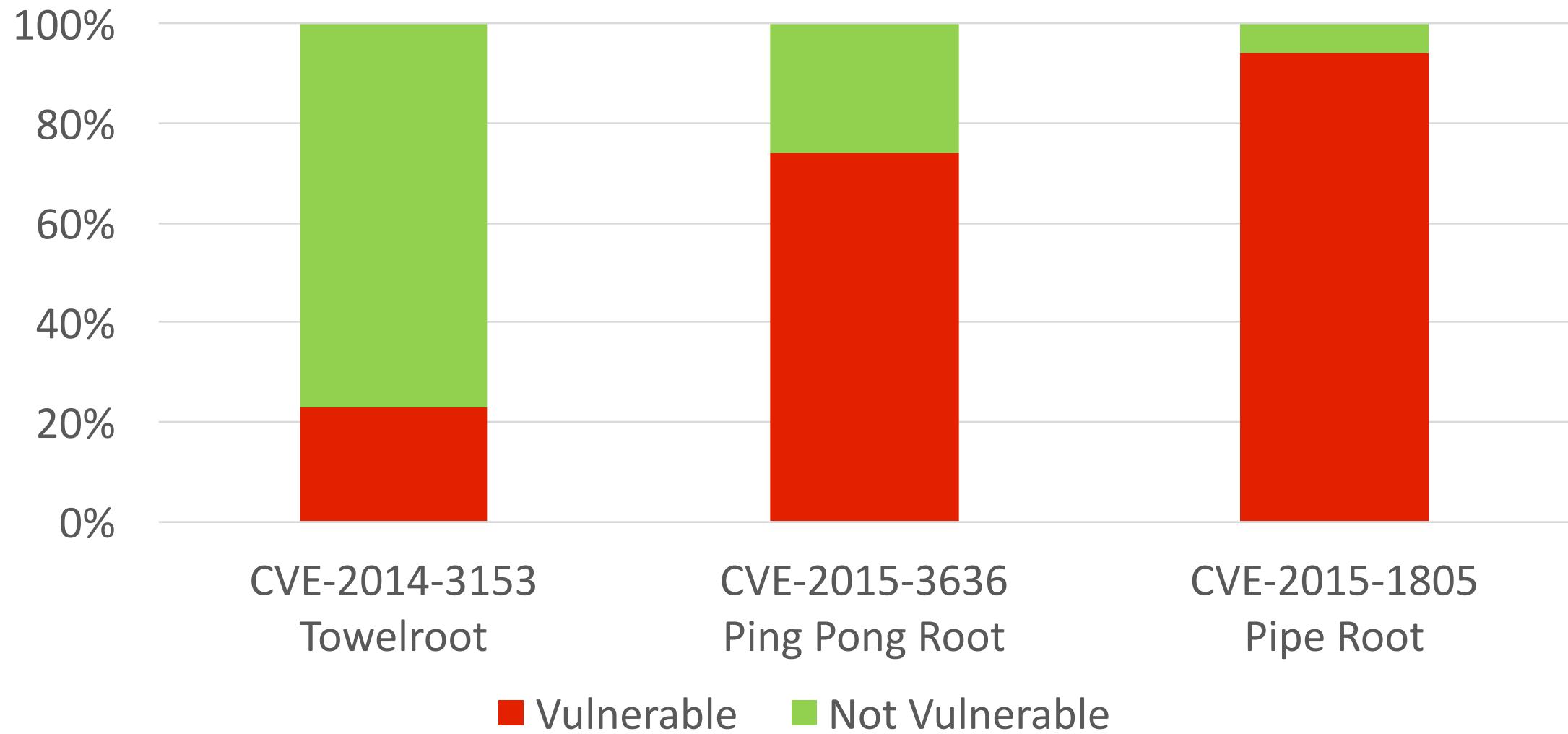
- The `ping_unhash` function in `net/ipv4/ping.c` in the Linux kernel before 4.0.3 does not initialize a certain list data structure during an unhash operation, which allows local users to gain privileges or cause a denial of service.

CVE-2015-1805 (used in KingRoot)



- The pipe_read and pipe_write implementations in kernel before 3.16 allows local users to cause a denial of service (system crash) or possibly gain privileges via a crafted application.
- A known issue in the upstream Linux kernel that was fixed in April 2014 but wasn't called out as a security fix and assigned CVE-2015-1805 until February 2, 2015.

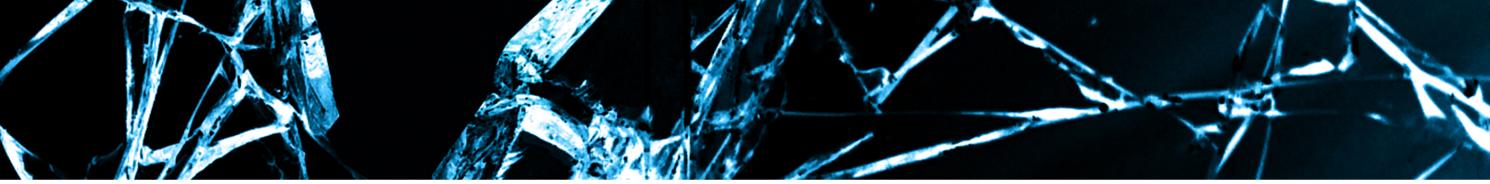




Vulnerability statistics collected from Chinese Android device in July 2016

How/Who to Secure Them???





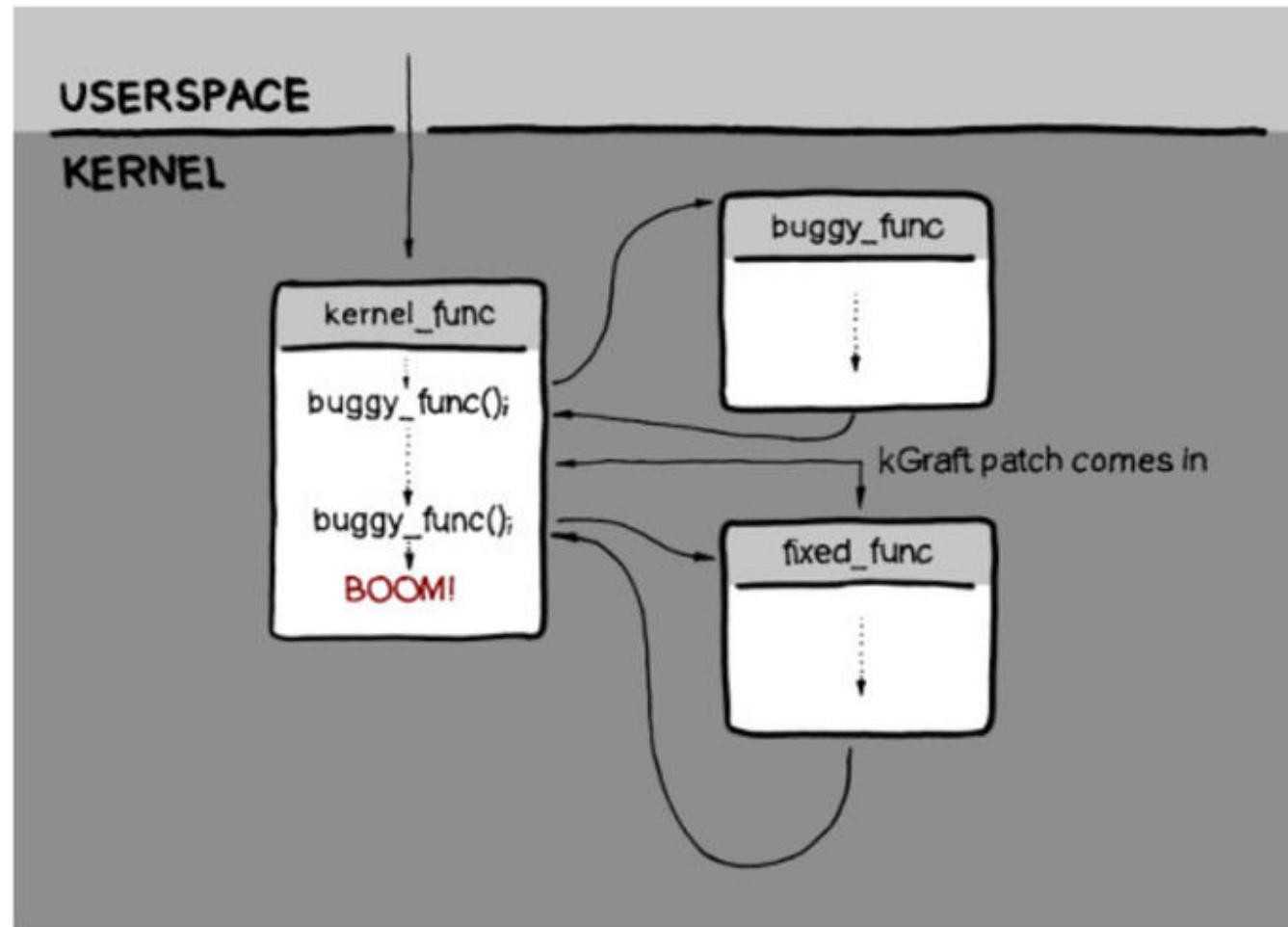
Agenda

- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - LuaKpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem

Kernel Live Patching

- kpatch
- kGraft
- kssplice
- Linux upstream's livepatch
-

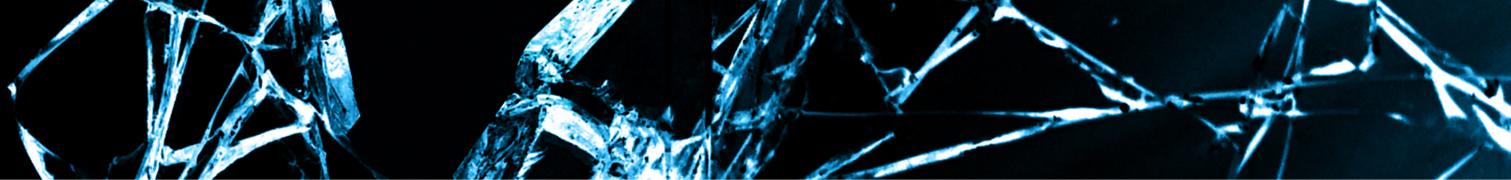
Kernel Live Patching



kGraft as an example

Kernel Live Patching

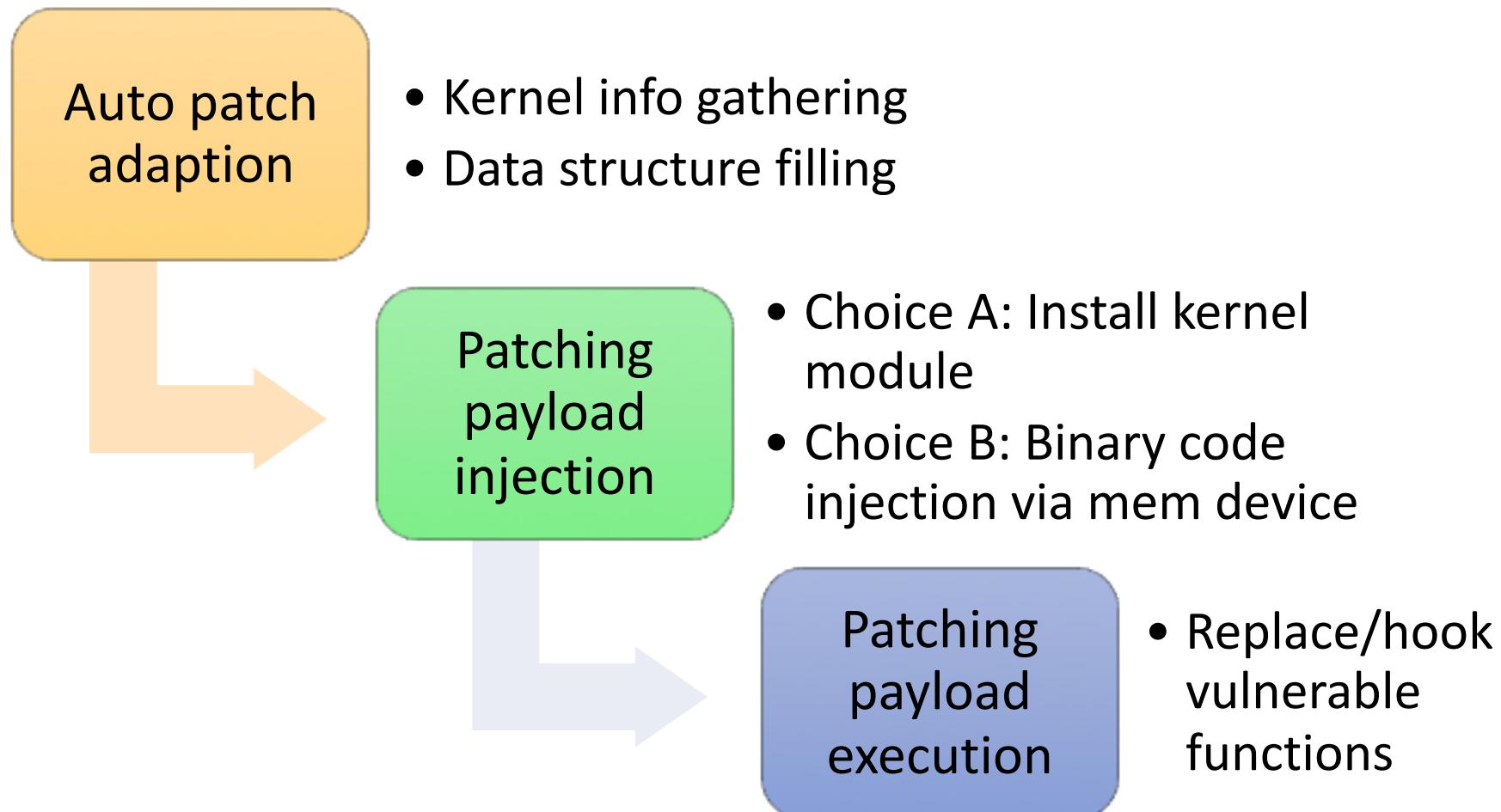
- Load new functions into memory
- Link new functions into kernel
 - Allows access to unexported kernel symbols
- Activeness safety check
 - Prevent old & new functions from running at same time
 - `stop_machine()` + stack backtrace checks
- Patch it!
 - Uses ftrace etc.



Challenges for Third Party

- Most existing work requires source code. Phone vendor is the only guy that can generate the live patches
- Unable to directly apply patches to other kernel builds

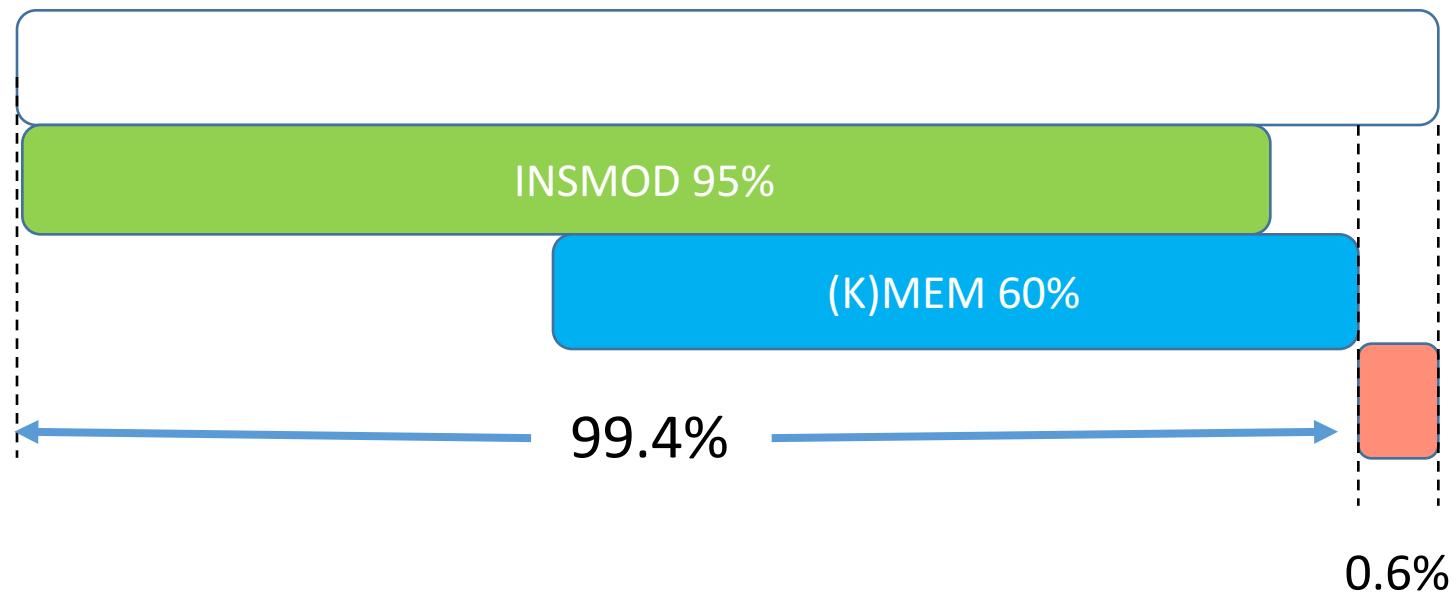
AdaptKpatch - Adaptive Live Patching

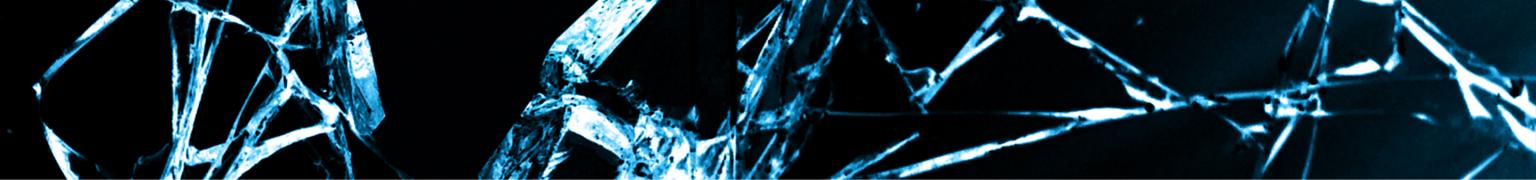


Kernel Info Collection

- Kernel version
 - /proc/version
 - vermagic
- Symbol addresses/CRC
 - /proc/kallsyms (/proc/sys/kernel/kptr_restrict)
- Other kernel modules
 - Symbol CRC/module init offset
- Boot image
 - decompress gzip/bzip/lzma/lzo/xz/lz4
 - some are raw code or even ELF file

Patch Injection Methods Coverage





Method A: Kernel Module Injection

Kernel checks that need to be resolved for adaption

- vermagic check
- symbol CRC check
- module structure check
- vendor's specific check
 - ❖ Samsung lkmauth

Bypass vermagic/symbol CRC

- Big enough vermagic buffer
- Copy kernel vermagic string to module
- Copy kernel symbol CRCs to module

```
include/linux/vermagic.h
#define VERMAGIC_STRING
    UTS_RELEASE " "
    MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT \
    MODULE_VERMAGIC_MODULE_UNLOAD MODULE_VERMAGIC_MODVERSIONS \
    MODULE_ARCH_VERMAGIC

#define VERMAGIC_STRING "AAAAAAAAAAAAAAAAAAAAAAAAAAAAA=YA!"
```

Bypass Samsung lkmauth

.text:C00C7718	EXPORT lkmauth
.text:C00C7718 8C 32 9F E5	LDR R3, =__stack_chk_guard
.text:C00C771C F0 4F 2D E9	STMD SP!, {R4-R11,LR}
.text:C00C7720 54 D0 4D E2	SUB SP, SP, #0x54
.text:C00C7724 84 42 9F E5	LDR R4, =0xC1254B04
.text:C00C7728 01 A0 A0 E1	MOV R10, R1
.text:C00C772C 00 90 A0 E1	MOV R9, R0
.text:C00C7730 7C 02 9F E5	LDR R0, =lkmauth_mutex
.text:C00C7734 00 30 93 E5	LDR R3, [R3]
.text:C00C7738 4C 30 8D E5	STR R3, [SP,#0x78+var_2C]
.text:C00C773C 16 FC 1E EB	BL mutex_lock
.text:C00C7740 0A 10 A0 E1	MOV R1, R10
.text:C00C7744 6C 02 9F E5	LDR R0, =0xC0CC09D3
.text:C00C7748 E6 CA 1E EB	BL printk
.text:C00C774C 2C 00 8D E2	ADD R0, SP, #0x78+var_4C
.text:C00C7750 64 12 9F E5	LDR R1, =aTima_lkm ; "tima_lkm"
.text:C00C7754 9A 8C 08 EB	BL strcpy
.	...
.	...
.text:C00C7874 44 11 98 E5	LDR R1, [R8,#0x144]
.text:C00C7878 00 00 51 E3	CMP R1, #0
.text:C00C787C 02 00 00 1A	BNE lkmauth_failed // BNE => NOP
.text:C00C7880 54 01 9F E5	LDR R0, =0xC0CC0C0B
.text:C00C7884 97 CA 1E EB	BL printk
.text:C00C7888 3C 00 00 EA	B lkmauth_pass

Method B: mem/kmem Injection

- Symbol addresses
 - vmalloc_exec
 - module_alloc
- Structured shellcode
- Allocate/reuse memory
- Write into memory
- Trigger the running

```
struct shell_code_binary {  
    unsigned long magic;  
    unsigned long version;  
    unsigned long header_size;  
    unsigned long shellcode_size;  
    unsigned long shellcode_entry;  
    unsigned long lookup_name_offset;  
    unsigned long mmap_ram_start_offset;  
    unsigned long mmap_ram_end_offset;  
    unsigned long vuln_count_offset;  
    unsigned long vuln_ids_offset;  
    unsigned long current_pid_offset;  
    unsigned long kmem_write_count;  
    unsigned long patch_count;  
    unsigned long* write_offset_array;  
    unsigned long* patch_ids_array;  
    unsigned long* patch_offset_array;  
    unsigned char* shellcode_body;  
};
```

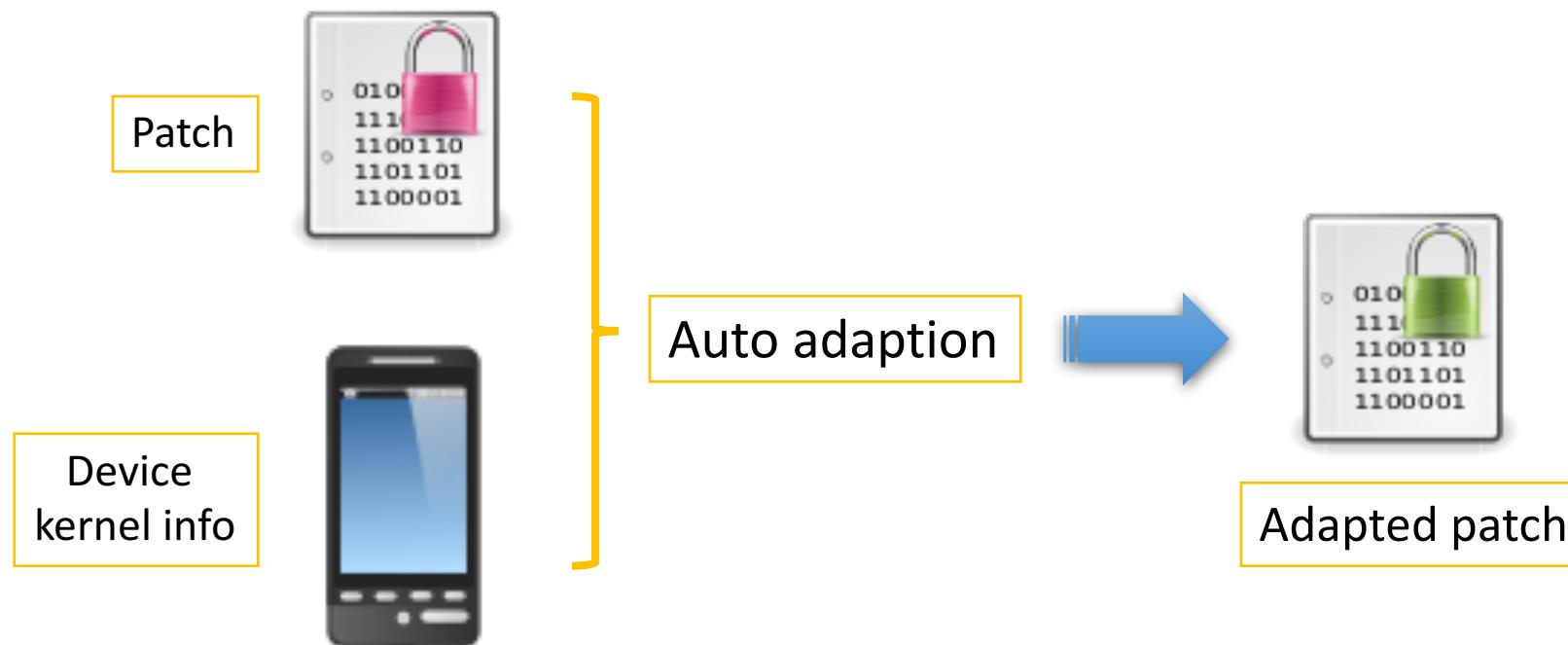
Patching Payload Execution

- Overwrite the function pointer
- Overwrite with patch code directly
- Inline hook

Same with other
live patching methods

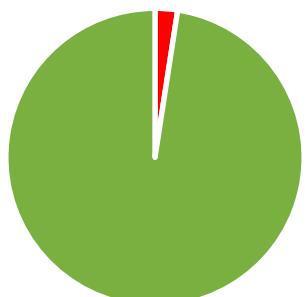
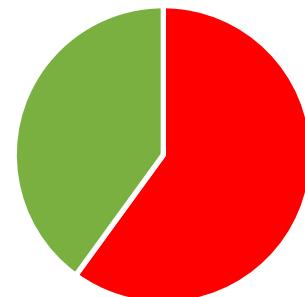
Adaption Challenges Solved

- Patch automatic adaption



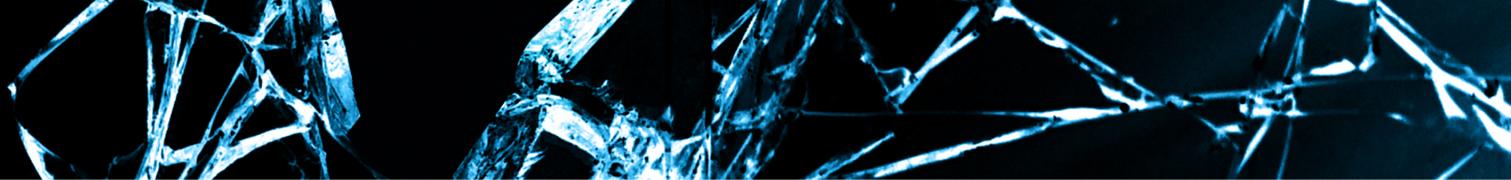
Challenges Solved

- ✓ Most existing work requires source code. Phone vendor is the only guy that can generate the live patches
- ✓ Unable to directly apply patches to other kernel builds



■ Vulnerable ■ Immutable

■ Vulnerable ■ Immutable



Successfully Evaluated CVEs

- mmap CVEs → Framaroot
- CVE-2014-3153 → Towelroot
- CVE-2015-0569
- CVE-2015-1805 → Pipe Root
- CVE-2015-3636 → Ping Pong Root
- CVE-2015-6640
- CVE-2016-0728
- CVE-2016-0805
- CVE-2016-0819
- CVE-2016-0844
-

Successfully Evaluated on Most Popular Phones



GT-I8552



GT-S7572



S4



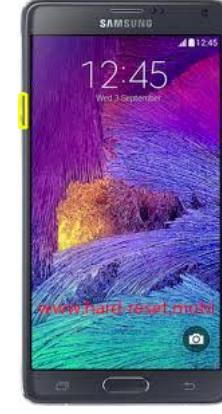
A7



SM-G5308W



Grand 2



Note 4



Successfully Evaluated on Most Popular Phones



C8813



P6-U06



Honor



U8825D



Successfully Evaluated on Most Popular Phones



M7



M8Sw



S720e



T528d

htc

Successfully Evaluated on Most Popular Phones



A630t



A788t



A938t



K30-T

lenovo

Successfully Evaluated on Most Popular Phones



Coolpad 酷派



SONY



HUAWEI

oppo
SMARTPHONE

vivo
Smart Phone

htc

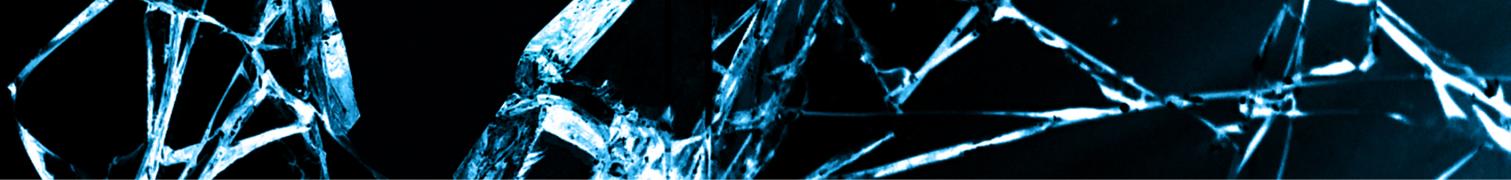
MEIZU 魅族

TCL

ONEPLUS

ZTE 中兴

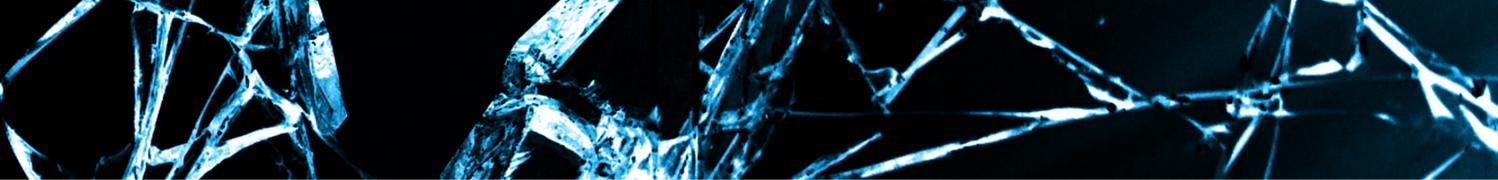
lenovo



Demo

Before Patch: **Ping Pong Root** succeed

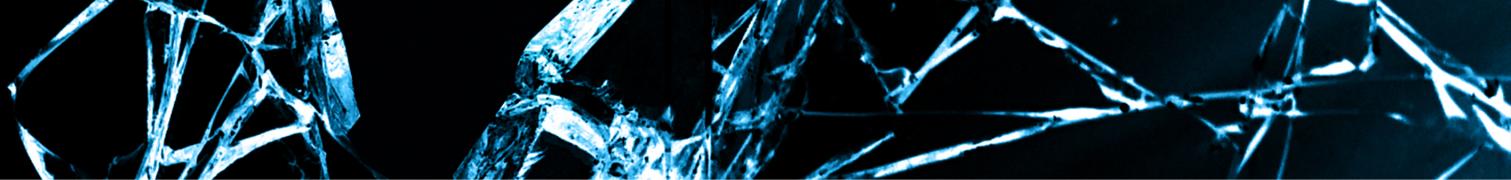
After Patch: **Ping Pong Root** fail



Recall the Two Problems

- The long patching chain
 - Solved by adaptive live patching
- Capability mismatching
 - To be solved by a joint-effort

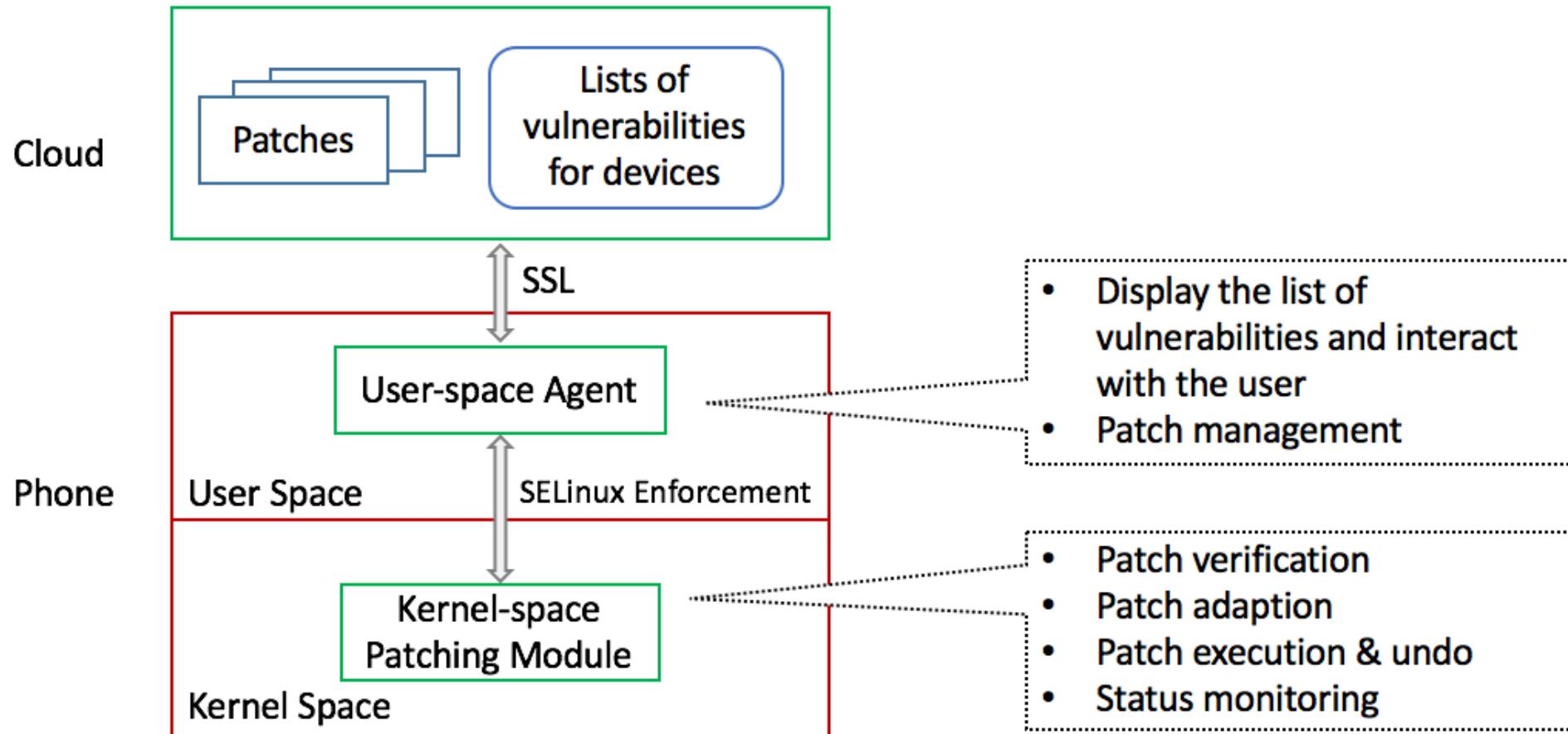




Exploit existing
vulnerabilities to gain root

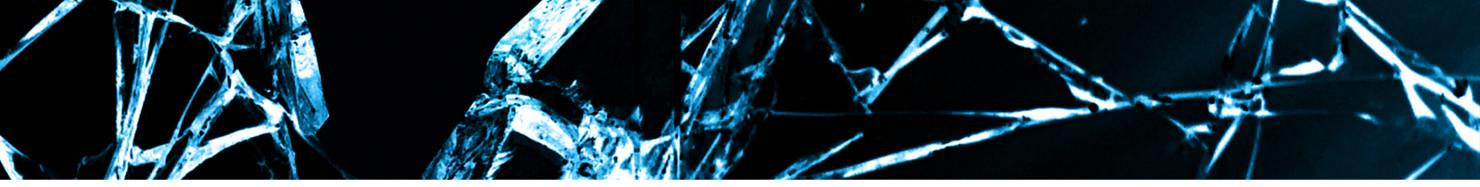


Vendor cooperation & pre-
embedded kernel agent



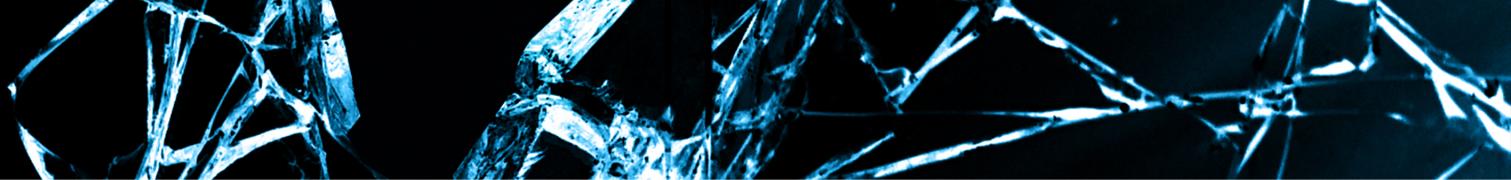
Multi-stage Vetting Mechanism





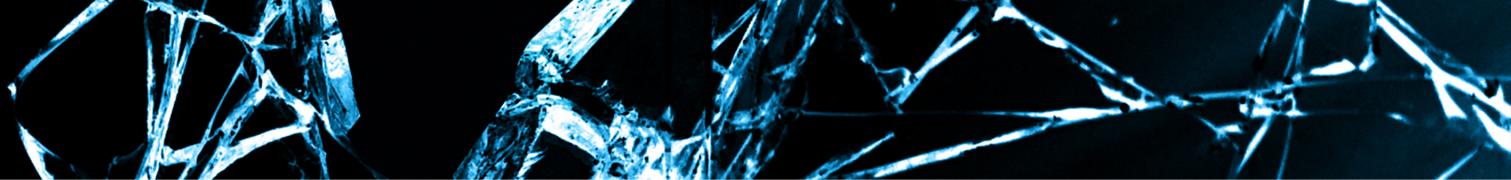
Agenda

- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - Luakpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem



We need a patching mechanism

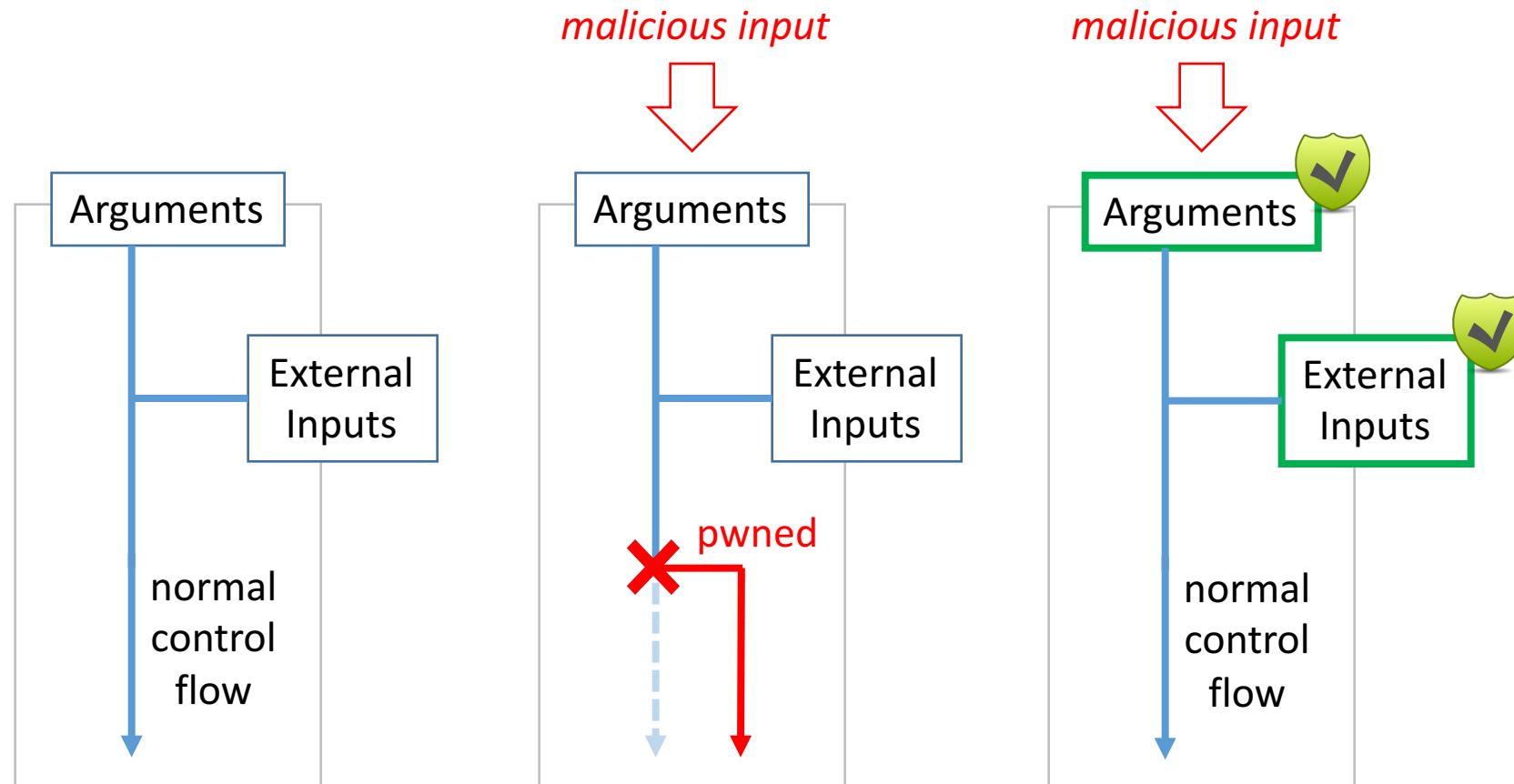
- powerful enough to block most threats;
- agile enough for quick patch generation;
- yet restrictive enough to confine possible damages caused by the patches.



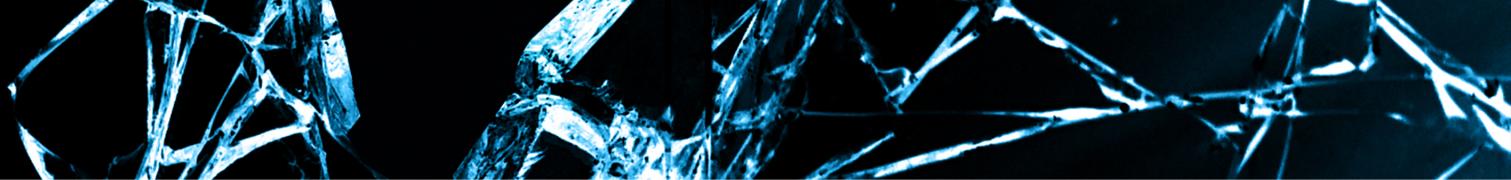
Our Solution -- LuaKpatch

Inserting a type-safe dynamic language engine (Lua)
into the kernel to execute patches

- Easy to update
- Naturally jailed in the language VM
- No need to worry about memory overflow etc. of the patches



By hooking the data input entries and validating the input, we can block most of the kernel exploits.

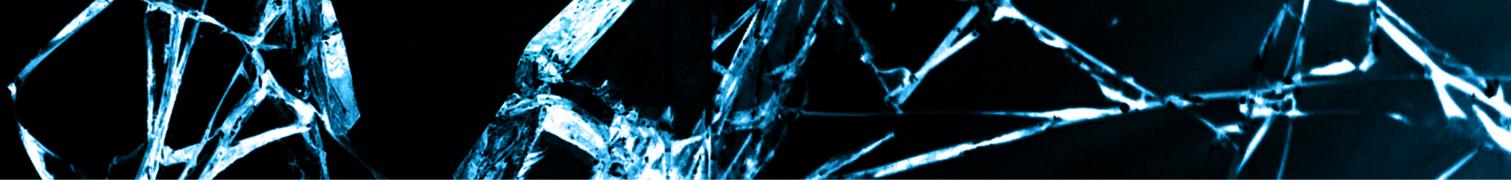


So we have the following restrictions

- 1) The patch can hook a target function's entry;
- 2) In combination with 1), within the target function, the patch can hook the invoking point or returning point of functions that return a status code (e.g., `copy_from_user`);
- 3) The patch can read anything that can be read (registers, stacks, heaps, code, etc., as long as it does not trigger faults), but cannot modify original kernel memory (no write, and no data can be sent out);
- 4) After judging whether the input is malicious or not, the patch can return specific error codes.

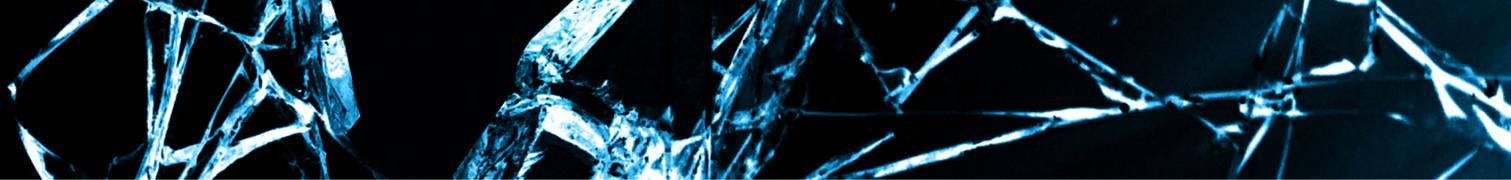
```
1: fun(...) {
2:     // entry of A can be hooked
3:     bool result;
4:     struct *s;
5:
6:     // foo is allowed to be hooked
7:     result = foo(...);
8:     if (result == E_INVALID)
9:         return;
10:
11:    // bar cannot be hooked
12:    s = bar(...);
13:    if (s)
14:        s->fun();
15: }
```

A running example to illustrate which functions can be hooked and which cannot



Implementation of LuaKpatch

- Many practices followed from the *lunatik-ng* project.
- Line-of-Code (LoC) is ~11K. 600 LoC are the core patching logic.
- Compiled as a 800KB kernel module.
- Capability interfaces:
 - Symbol searching
 - Hooking
 - Typed reading
 - Thread info fetching



```
1  function kpatcher(patchID, sp, cpsr, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r14)
2      if patchID == 1 then
3          uaddr1 = r0
4          uaddr2 = r2
5
6          if uaddr1 == uaddr2 then
7              return ERROR
8          else
9              return 0
10         end
11     end
12   end
13
14   fun = kpatch.search_symbol('futex_requeue')
15   kpatch.hook(1, fun)
```

Sample Lua patch to fix one of the vulnerable conditions of CVE-2014-3153, known as “Towelroot”

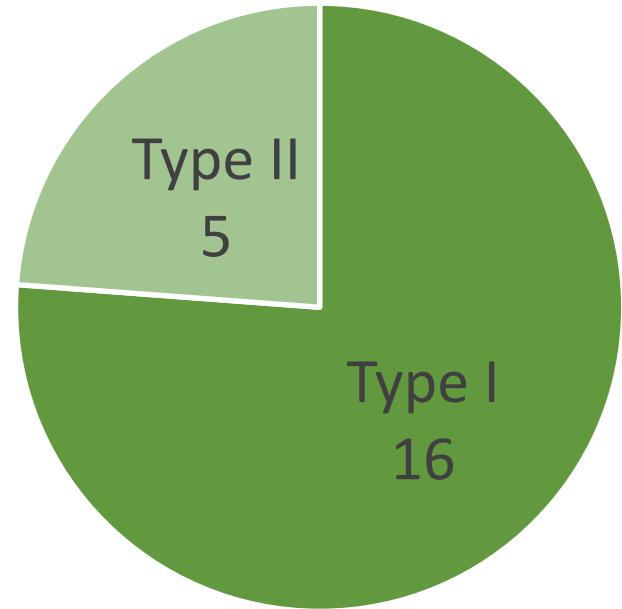
Efficacy Evaluation

CVE-2012-4220	CVE-2013-6123	CVE-2015-3636
CVE-2012-4221	CVE-2013-6282	CVE-2015-6619
CVE-2012-4222	CVE-2014-3153	CVE-2015-6640
CVE-2013-1763	CVE-2014-4321	CVE-2016-0728
CVE-2013-2094	CVE-2014-4322	CVE-2016-0774
CVE-2013-2596	CVE-2015-0569	CVE-2016-0802
CVE-2013-2597	CVE-2015-1805	CVE-2016-2468

CVEs verified to be protectable by LuKpatch.

Most are Type I vulnerabilities (those that can be patched by simply hooking the entry of the vulnerable functions), but the highlighted/colored ones are Type II vulnerabilities (those that also need to hook the invocations that return status code).

Efficacy Evaluation



*All 21 CVEs can be patched by LuaKpatch. 16 are Type I, and 5 are Type II.
So 76% of them can be easily fixed by hooking and checking input at the function entry.*

Example I (CVE-2013-1763)

```
diff --git a/net/core/sock_diag.c b/net/core/sock_diag.c
index 602cd63..750f44f 100644
--- a/net/core/sock_diag.c
+++ b/net/core/sock_diag.c
@@ -121,6 +121,9 @@ static int __sock_diag_rcv_msg(struct sk_buff *skb, struct nlmsghdr *nlh)
     if (nlmsg_len(nlh) < sizeof(*req))
         return -EINVAL;

+    if (req->sdiag_family >= AF_MAX)
+        return -EINVAL;
+
     hndl = sock_diag_lock_handler(req->sdiag_family);
     if (hndl == NULL)
         err = -ENOENT;
```

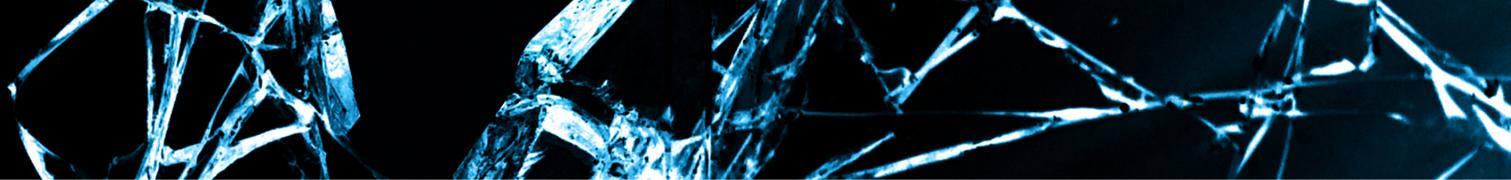
LuaKpatch can patch it by hooking the entry of the `__sock_diag_rcv_msg` function, getting the `nlh` argument, obtaining `req` from `nlh`, and then checking whether the condition `req->sdiag_family >= AF_MAX` is satisfied. If this is true, it is an exploit condition and the patch should return an error.

Example II (CVE-2013-6123)

```
-----  
if (copy_from_user(&u_isp_event,  
    (void __user *)ioctl_ptr->ioctl_ptr,  
    sizeof(struct msm_isp_event_ctrl))) {  
    pr_err("%s Copy from user failed for cmd %d",  
        __func__, cmd);  
    rc = -EINVAL;  
    return rc;  
}  
-----
```

```
diff --git a/drivers/media/video/msm/server/msm_cam_server.c b/drivers/media/video/  
index 5fc8e83..6e49082 100644  
--- a/drivers/media/video/msm/server/msm_cam_server.c  
+++ b/drivers/media/video/msm/server/msm_cam_server.c  
@@ -1390,6 +1390,15 @@ static long msm_ioctl_server(struct file *file, void *fh,  
}   
  
    mutex_lock(&g_server_dev.server_queue_lock);  
+  
+  
+  
+  
+  
+  
+  
+  
+  
+  
    if(u_isp_event.isp_data.ctrl.queue_idx < 0 ||  
        u_isp_event.isp_data.ctrl.queue_idx >= MAX_NUM_ACTIVE_CAMERA) {  
        pr_err("%s: Invalid index %d\n", __func__,  
               u_isp_event.isp_data.ctrl.queue_idx);  
        rc = -EINVAL;  
        return rc;  
    }  
  
    if (!g_server_dev.server_queue  
        [u_isp_event.isp_data.ctrl.queue_idx].queue_active) {  
        pr_err("%s: Invalid queue\n", __func__);  
    }
```

LuaKpatch can patch it by hooking the returning point of the `copy_from_user` invoked by `msm_ioctl_server` to check the exploit condition.

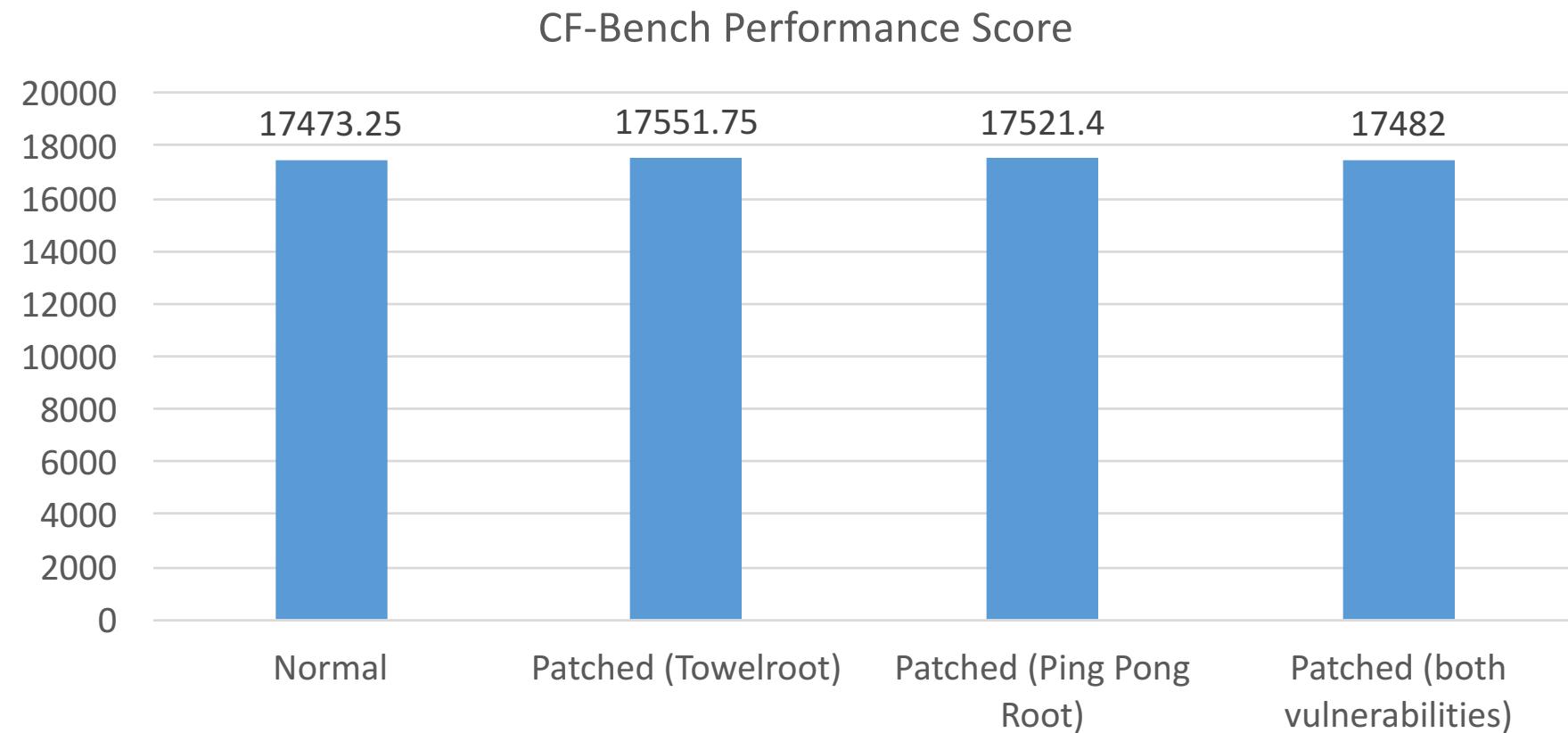


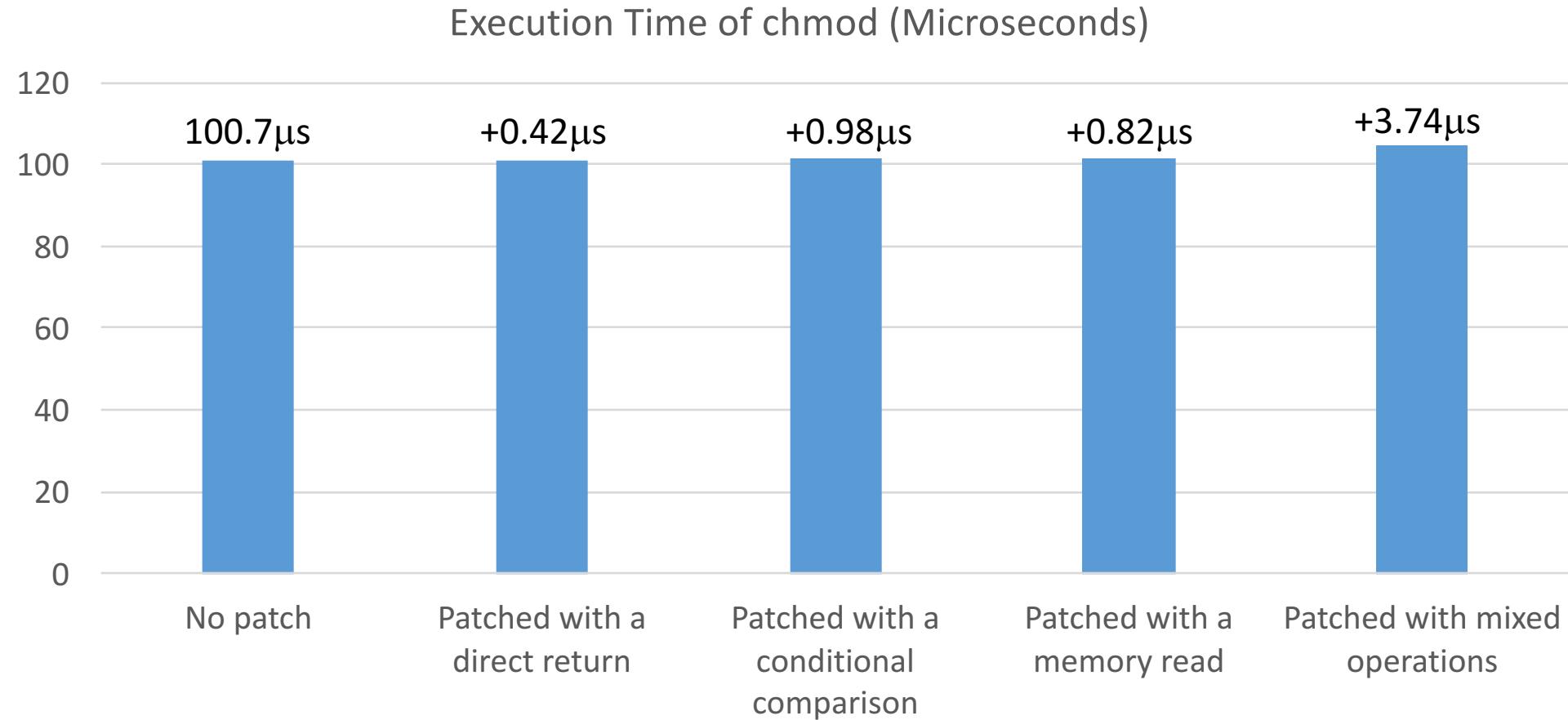
Demo

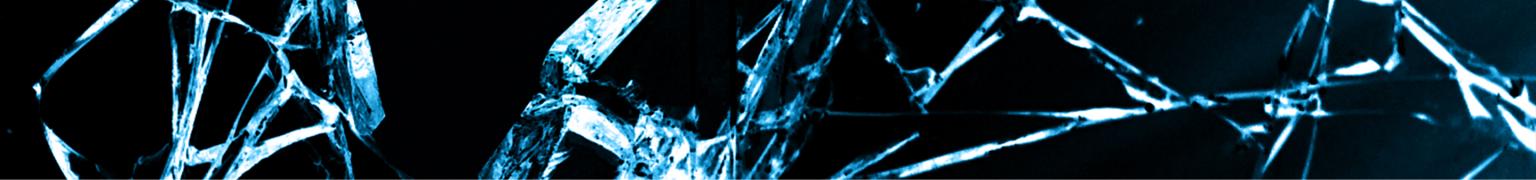
Before Patch: Vulnerable to **Towelroot** and **Ping Pong Root**

After Patch: Immune to **Towelroot** and **Ping Pong Root**

Performance Evaluation



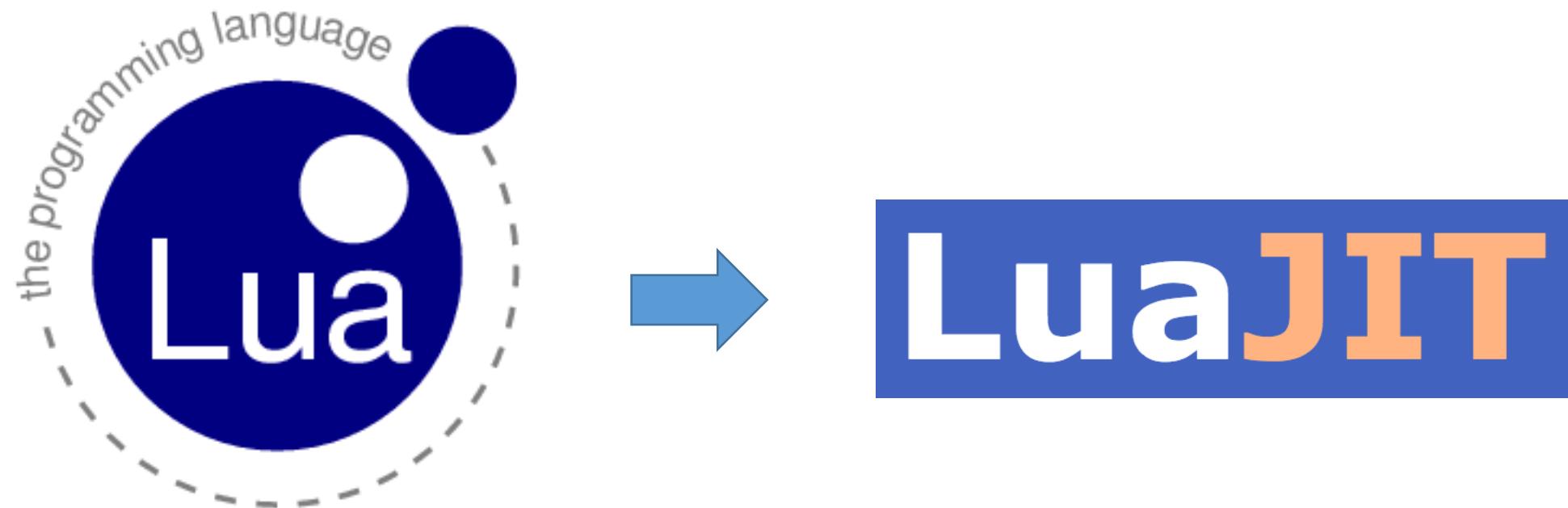




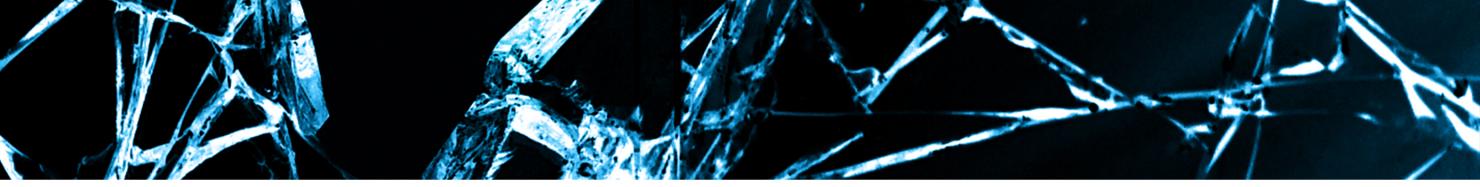
LuaKpatch validation check adds an overhead under 4 microseconds, only **4%** of a chmod system call.

Because system calls are not invoked all the time, the impact to the overall system performance should be even less.

- When a user normally browses Internet using Chrome on Nexus 5 + Android 4.4, `gettimeofday` was the mostly-called system call, triggered for ~110,000 times. The overall performance overhead can be estimated as $5\mu\text{s} * 110,000 / 1\text{min} \approx 0.9\%$, which is quite small.



As an ongoing work, we are migrating Luakpatch to LuaJIT, which should further improve the performance.

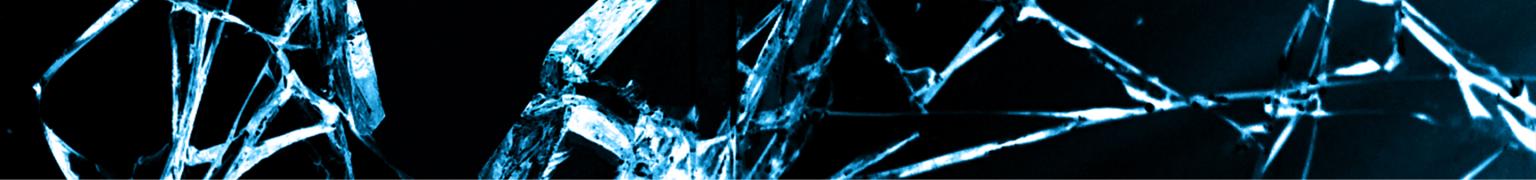


Agenda

- The Problem
 - Android Kernel Vulnerability Landscape
 - Why Are They Long-lasting?
 - Case Studies
- The Solution
 - AdaptKpatch: Adaptive Kernel Live Patching
 - LuaKpatch: More Flexibility, Yet More Constraint
- The Future
 - Establishing the Ecosystem

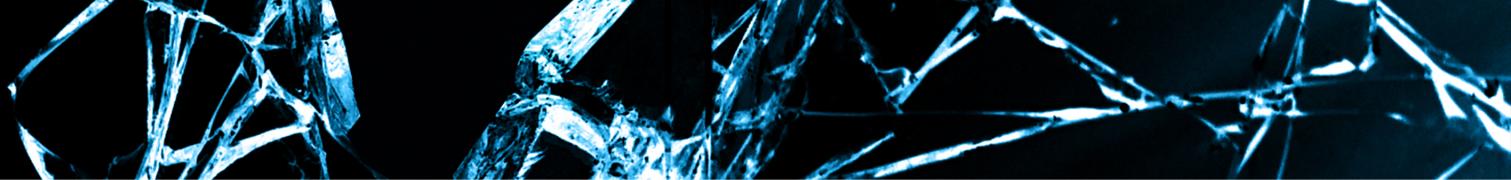


The patching circle in the open collaborative patching ecosystem



Let's fight the bad together!

- The number and the complexity of kernel vulnerabilities keep increasing, so more joint effort makes it easier to battle against them.
- In the AdaptKpatch scheme, patches can be vetted and cross-validated by qualified alliance members.
- Last but most importantly, all vendors can join together to develop a patching standard instead of implementing different variants. If different hot patching mechanisms exist, it introduces another layer of fragmentation.



Thanks!

Yulong Zhang, Yue Chen, Chenfu Bao, Liangzhao Xia,
Longri Zheng, Yongqiang Lu, Lenx Wei

Baidu X-Lab

August 2016