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

Yulong Zhang, Yue Chen, Chenfu Bao, Liangzhao Xia, Longri Zhen, Yongqiang Lu, and Tao Wei
Baidu X-Lab

1. Background

Android kernel vulnerabilities are dangerous. Attackers can exploit kernel vulnerabilities to root Android devices and achieve malicious goals. If the kernel gets compromised, all security mechanisms relying on kernel integrity (e.g. app sandboxing, payment/fingerprint framework, etc.) will be broken. TrustZone, the last line of defense for Android devices, will also be threatened as the compromised kernel enables the attacker to inject malicious payloads into TrustZone.

So, it is ideal to get rid of kernel vulnerabilities. However, more and more kernel vulnerabilities have been unveiled each month (shown in Figure 1). On one hand, this is a good trend indicating more and more effort has been spent on securing the Android kernel; on the other hand, kernel vulnerabilities that have been disclosed but remain unfixed have become the largest threat for Android users. Having been in the spotlight for weeks or even months, they usually have public and stable exploits, and thus become underground businesses' favorite. For example, from the recent popular malware incidents with global impact [1-4], one can always find a root toolkit inside. Such root exploits are either from public Proof-of-Concept (PoC), or copied from one-click root apps. Consequences of these malware include frauds, credential leakage, fingerprint leakage, losing money from banking/payment/financial accounts, or even remote controls.

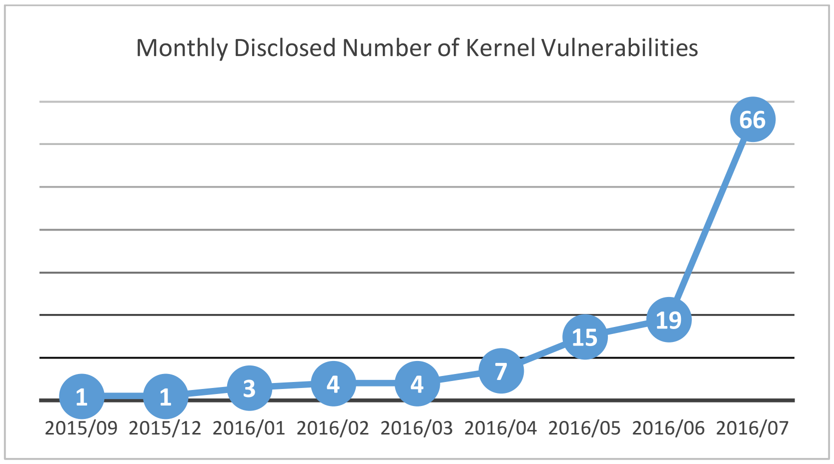


Figure 1: Number of kernel vulnerabilities disclosed in monthly Android Security Bulletin

People commonly claim that “using Android is easier to be attacked”, or “Android is less secure than iOS”. Actually, if we simply compare the kernel vulnerabilities disclosed in each iOS release (Table 1), it appears that iOS’ kernel vulnerability disclosure frequency is not less than Android’s. So the root cause of the problem is not only that Android has frequent vulnerability disclosures, but also because these vulnerabilities remain unfixed over a long time.

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

Table 1: iOS kernel vulnerabilities disclosed in each release, compared to that of Android. Note that the number of vulnerabilities may not be directly comparable due to different threat levels. But the comparison can still provide a rough sense that iOS is not born as “bullet-proof”; it is more secure just because it can get timely patches in a large scale.

2. Why Are Android Kernel Vulnerabilities Long-lasting?

Why Android cannot get these kernel vulnerabilities patched in a timely manner like iOS? There are mainly three causes:

1) The long patching chain delays the patch effective date;
2) Fragmentation makes it challenging to adapt the patches to all devices;
3) Capability mismatching between device vendors and security vendors.

These causes are discussed in detail below.

2.1 The Long Patching Chain

As shown in Figure 2, it usually takes months or even years for a patch to pass all stages and land on users’ phones, not to mention that some patches may be discarded in one of these stages.

For example, in an academic study [5] , the authors claim that from reverse-engineering a famous root app, they found at least 10 kernel exploits that have never been reported to vendors. In this case, the vulnerabilities have been identified and made public (since everyone can reverse-engineer that famous root app to understand the vulnerabilities, or even reuse the root exploit library directly), but the vendors did not know their existence, so there is no patch released through the official patching channel to patch them. This is the scheme where the patching chain stops at the very first stage.

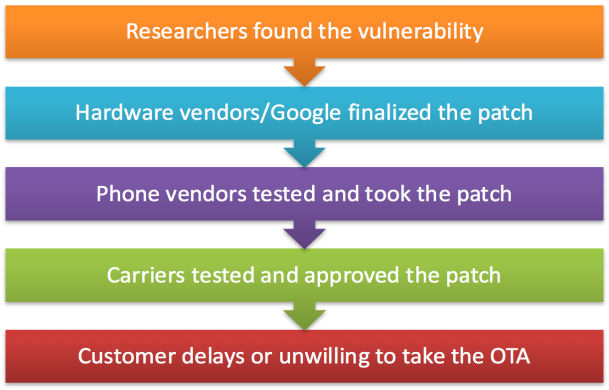

Researchers found the vulnerability
Hardware vendors/Google finalized the patch
Phone vendors tested and took the patch
Carriers tested and approved the patch
Customer delays or unwilling to take the OTA

Figure 2: The long patching circle of Android kernel vulnerabilities

As another example, Google Project Zero identified a bug [6] but somehow it was not applied to vendor's kernel branch. So according to this post, "the majority of Android kernels based on 3.4 or 3.10 are still affected despite the patch being available for 6 months". This is the scheme where the patching chain got delayed before reaching the third stage.

The first three steps could be skipped if the phone vendors are willing to allocate enormous resources and manpower to identify kernel vulnerabilities and release patches timely. However, the patching chain would still be delayed by the carriers [7] , who need to thoroughly test whether a proposed patch affects the stability and service quality of the carrier phones.

Finally, when an Over-The-Air (OTA) patch arrives at the devices, users may not be willing to take the patch immediately since most of the cold patches require phone rebooting to become effective. So the concern of interrupting user experience further delays the traditional patches' effective date.

2.2 Fragmentation

The Android ecosystem is open. There are countless vendors and each vendor usually has many phone models (Figure 3). The different patching status of various models causes fragmentation. It is tedious to manually port a patch to so many platforms, which further slows down the patching process.

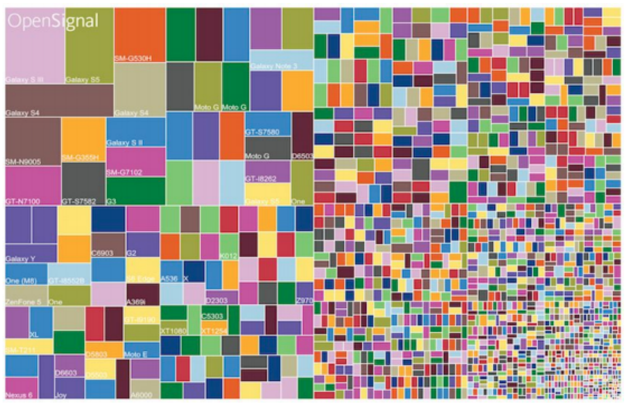


Figure 3: Android device fragmentation, cited from [8]

Google also provides official statistics of fragmented Android version distribution [9] . The following facts stand out:

1) Lollipop (Android 5.0) was released in November 2014, but as of July 21, 2016, **51.6%** of the devices are still older than that. This indicates that the whole Android ecosystem is slow to catch up with the latest Android release.
2) Google has stopped providing patches for Android older than 4.4, but as of July 21, 2016, **21.5%** of the devices are still older than that. This means that more than 1/5 of the devices will never receive timely patches.

Making the situation worse, the world's largest Android market, China, blocks network access to Google, so Chinese devices cannot obtain OTA patches directly from Google. Due to the same reason, Chinese devices cannot report their patching level to Google, thus Google's Android version statistics do not include Chinese devices – if they do, the statistics should look even more upsetting. We randomly collected the Android versions of nearly two millions of devices with Baidu apps installed, and observed the following facts:

1) Lollipop (Android 5.0) was released in November 2014, but as of July 21, 2016, **80%** of the Chinese devices are still older than that.
2) Google has stopped providing patches for Android older than 4.4, but as of July 21, 2016, **42%** of the Chinese devices are still older than that.

Therefore, the problem is not only to adapt patches to lots of platforms, but also to adapt patches for so many old devices. Most device vendors focus more on promoting new devices rather than providing support for devices more than 2 years old. So although there are still a considerable amount of users sticking to old devices, they will unfortunately not receive timely patches.

2.3 Capability Mismatching

Device vendors have privileges high enough to apply the patches on the phone. They can also adapt the patches based on the source code they possess – it is tedious, but still feasible. But as mentioned above, their first priority is not to patch the devices to bullet-proof. Moreover, exploits and defenses are not their area of expertise.

On the contrary, others have interest to provide timely patches for devices, like security vendors (for fame and profit) and developers of apps relying on Android security (for risk control). They have better understanding of the real-world threats too. However, since device vendors usually don't publish the exact up-to-date kernel source code for all devices, it is extremely difficult for third parties to adapt patches for all devices. It is also challenging for third parties to apply patches since they don't have enough privileges.

We call this dilemma as "Capability Mismatching" – those capable to apply the patch do not have highest interest or expertise to generate the patch, while those capable to generate the patch do not have capability to adapt and apply the patches. This is the third but still an important cause of the Android's vulnerabilities remaining unfixed for a long time.

Now that we have identified the three major causes, we can explain why Android appears to be less secure than iOS:

1) Apple controls the entire supply chain, from hardware to software, and have more rights speaking to carriers, so its patching chain is significantly shorter.
2) The number of iOS device variants is much smaller than Android's, and Apple has the source code for them, so it is much easier for Apple to generate and adapt patches for all variants.
3) From time to time, Apple refuses to sign old versions, so devices can only be upgraded and downgrading is not possible.

3. Case Studies of Long-lived Kernel Vulnerabilities

Here we use three widely exploited cross-platform kernel vulnerabilities for case studies. The first one is CVE-2014-3153, known as "*Towelroot*". The vulnerability is due to that the `futex_requeue` function in kernel through 3.14.5 does not ensure that calls have two different futex addresses, which allows local users to gain privileges. At its time being there was no Android Security Bulletin issued, so we conservatively count the first-known date as the time when this vulnerability was firstly patched in the upstream kernel: June 3, 2014 [10] .

The second one is CVE-2015-3636, known as "*Ping Pong Root*". The `ping_unhash` function in 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. The Android Security Advisory for this vulnerability was published on September 9, 2015 [11] .

The third one is CVE-2015-1805, where the `pipe_read` and `pipe_write` implementations in kernel before 3.16 allows local users to gain privileges via a crafted application. The upstream patching date is April 3, 2014 [12] . This vulnerability is very interesting because it was fixed in upstream but was not assigned a CVE number until February, 2015. However, no Android patch was provided for this vulnerability until early 2016 when researchers notified Google that the issue could be exploited and had been abused in the wild. The Android Security Advisory for this vulnerability was published on March 18, 2016 [13] .

Figure 4 shows the elapsed days from the advisory publication date for the three representative kernel vulnerabilities. The more days elapsed, the more victims are exploited due to the vulnerability. Note that even the shortest period (the one for CVE-2015-1805) has been longer than 120 days.

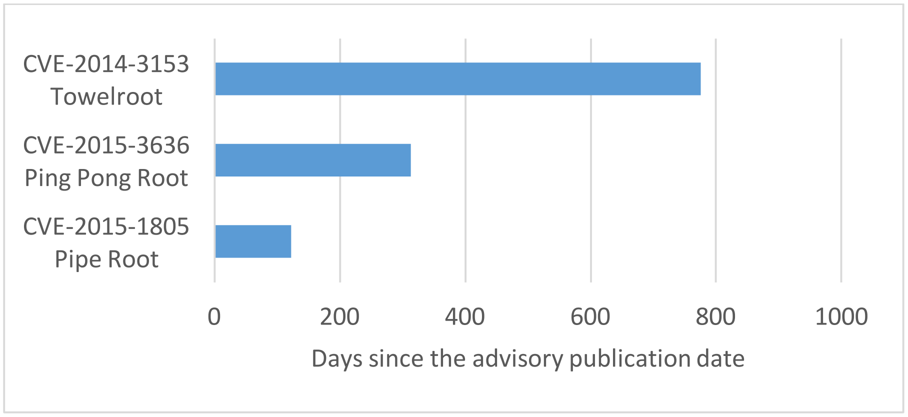


Figure 4: Days from advisory publication date to now (July 2016) for the three famous "one-to-root-all" kernel vulnerabilities

It is against our user-agreement to exploit the devices to detect whether a vulnerability exists or not. So we simply collected the kernel version and build timestamp from nearly two millions of devices in China with Baidu apps installed. We assume that if a kernel was built earlier than the advisory date, it is highly likely that it is unpatched for the corresponding vulnerability; otherwise it is assumed as patched. This conservative assumption should underestimate the rate of vulnerable devices because few device vendors can catch up with Google's Android Security Advisories. As shown in Figure 5, this conservative counting still yields a large portion of vulnerable devices. It is sad to see that so many users are still under root exploit attack even after months or years of the vulnerability's disclosure.

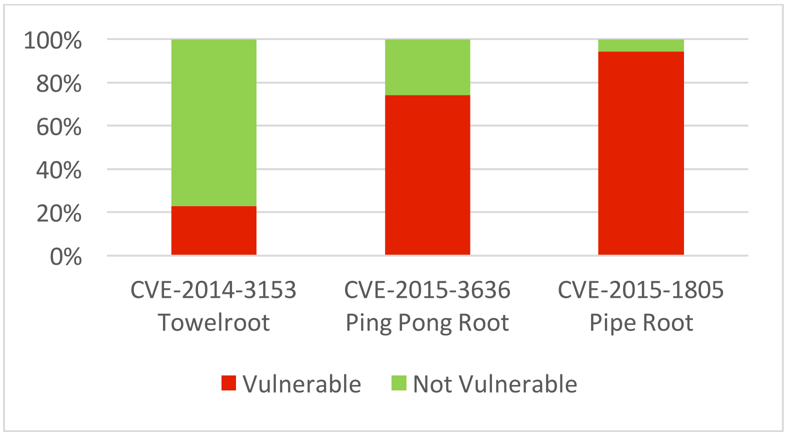


Figure 5: Vulnerability statistics collected from Chinese Android device

4. AdaptKpatch: Adaptive Kernel Live Patching

How can we solve the problem then? To shorten the long patching chain, both the academia and industry have proposed kernel live patching methods. Representative implementations include kpatch [14] , ksplice [15] , kGraft [16] , and Linux upstream's livepatch [17] . Details may vary for different solutions, but they usually share the following steps:

1) Load new code into memory, and make it accessible (exported) to existing code;
2) Perform safety check, to avoid the mixed context (old & new code invoked in the same call stack);

3) Switching execution from old code to the new code.

However, we still have two problems to address – the fragmentation issue and the capability mismatching issue. All the existing kernel live patching mechanisms require source code. The patching code is generated by comparing the difference of two binaries, one compiled from the old source code and the other from the patched source code. Once generated, this certain piece of patching code can only be applied to the certain kernel build; it cannot be installed to other platforms due to symbol dependency, etc.

So is there a way to automatically adjust binary patches to adapt to different device models with different Android kernel versions? The answer is yes. In the 7th HITB Conference [18] , we have already presented *AdaptKpatch* to perform automatic patch adaption to all Android devices, as shown in Figure 6.

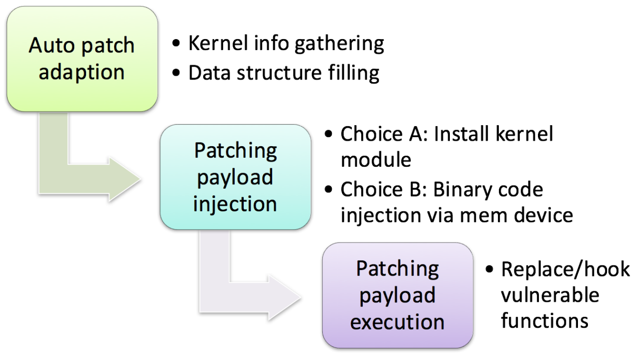

Auto patch adaption
• Kernel info gathering
• Data structure filling
Patching payload injection
• Choice A: Install kernel module
• Choice B: Binary code injection via mem device
Patching payload execution
• Replace/hook vulnerable functions

Figure 6: Adaptive kernel live patching procedures without phone vendors' cooperation [18]

To be brief, when our patching agent lands on a phone and detects kernel vulnerabilities that can be exploited to gain root, it will notify the user about the vulnerabilities and ask whether he/she wants to fix it. Once permitted, the patching agent roots the device and collects necessary information needed for adaption, including kernel versions, symbol addresses, symbol CRCs, etc. Next, depending on whether the device provides `insmod` or the `(k)mem` device, the agent either downloads a kernel module template or a binary code template for patching, and fills the collected info into the template to generate the adapted patch. This procedure solves the symbol dependencies of the patch and fulfills kernel checks, so the generated patch can be inserted into the kernel via `insmod` or `(k)mem` without a problem. Once the patch can be executed in the kernel mode, the rest of the patching procedure is the same as other existing solutions like kpatch/ksplice/kGraft etc. This enables third party vendors, who do not have access to the exact source code of the device kernel and drivers, to perform live patching.

Note that AdaptKpatch only performs hot patching – it does not modify the binaries on the file system, so it does not conflict with the static integrity protection mechanisms (such as the Secure Boot). On rebooting, all the patches are gone so the framework should be launched in the very early stage of the booting process and reload the patches. A boot flag is used to detect if last booting failed. In that case, AdaptKpatch will perform failover without loading the problematic hot patches.

This framework can be further improved if there exists a joint effort between the device vendor and the third party patch developer. In the above design, the patching agent needs to probe kernel vulnerabilities. However, it is difficult to precisely probe kernel bugs without disturbing user experience. So it would be better if the device vendor can provide a list of vulnerabilities corresponding to each kernel build. This should be easy for device vendors since they possess the source code. Moreover, it would be even better if the device vendor can integrate a patching module in the kernel. This saves the third party from rooting the device. Nevertheless, these improvements are not required but can highly boost the efficiency.

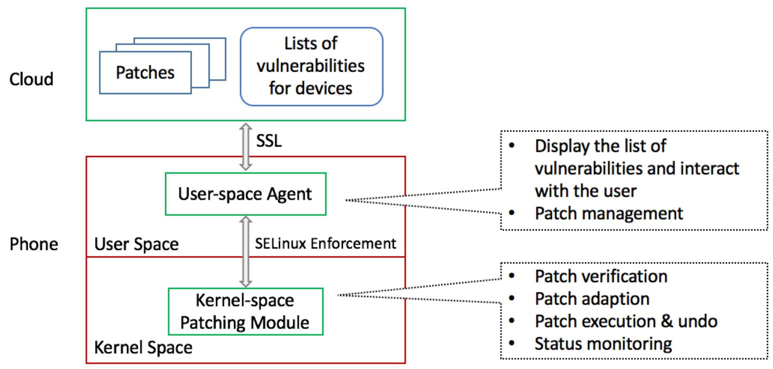


Figure 7: Adaptive kernel live patching framework with phone vendor's cooperation

The framework with device vendors' cooperation is shown in Figure 7. The user-space agent communicates to cloud for vulnerabilities of the current kernel build, and downloads patch templates accordingly. After prompting to the user and getting authorized, it sends the patches to the kernel patching module. The SELinux policy enforces that only the agent can talk to the patching module. The patches should be signed so that the kernel patching module can reject illegal code pieces. The patches should also be versioned so they can only be upgraded without being downgraded. After passing the verification, the kernel patching module then executes the patch and can later undo the patch. The kernel patching module should also monitor the system status for potential issues (and perform patch failover if there is an issue).

Now the device vendors can lay back and no longer need to worry about generating patches, and the motivated third parties can focus on providing patches without worrying about adaption. Everything seems perfect, except one concern – how can the device vendors trust the third party patches before signing and deploying them? To ensure the security of the patches, we need a multi-stage vetting mechanism:

1) Vendor qualification: the patch framework only accepts patches submitted from pre-qualified vendors with security expertise and good reputation.
2) Patch security vetting: the submitted patches should be reviewed and signed-off by other members in this alliance before deployment.

3) Reputation ranking: after being deployed on the devices, the patches are subject to user feedback. Any unstable or suspicious patches will be immediately removed and the corresponding vendor's reputation will be lowered. This is similar to the app store.

This alliance-based vetting mechanism can significantly reduce backdoors and new vulnerabilities introduced by the patches. But as long as the patches are in the native form (compiled from C and execute directly in kernel mode), it does not eliminate security issues. Especially when critical vulnerabilities are found and the phone vendors or the users want to deploy patches in a few days, it is still challenging to thoroughly review the binary patches. That is why we have an alternative: LuaKpatch.

5. LuaKpatch: More Flexibility, Yet More Constraint

5.1 Design Principles

If vendors have enough time (e.g., two weeks) to review the patches and run thorough tests against the patches, it should be fairly safe to deploy the patches using AdaptKpatch described above. However, if critical vulnerabilities arise and need to be hot-patched in two days, it is dangerous to load hasty patches into millions of devices. So we need a mechanism:

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

To fulfill the requirements, we can insert a type-safe dynamic language engine (such as Lua) into the kernel to execute patches. Programs written in dynamic languages are easy to update and are naturally "jailed" in the language virtual machine (VM). The type safety feature prevents the patches from introducing new vulnerabilities that can be attacked.

Then how should we expose the kernel to the patching engine to ensure minimum surface exposed but capable enough to do the patching work? After examining most of the recent kernel exploits, one can find that in order to trigger the vulnerability, the attacker needs to feed malicious data inputs into the target function. This is illustrated in Figure 8 – regardless of the attacker's tricks, he/she eventually needs to inject malicious input to hijack the control flow to achieve code execution. Therefore, we actually do not need to replace the whole function; if we can hook function entries or external data readings to validate input data, it is sufficient to detect most of the threats. If malicious input data are encountered, the validator returns an error code, so the afterward control flow will naturally fall into the error handling path instead of the original vulnerable path.

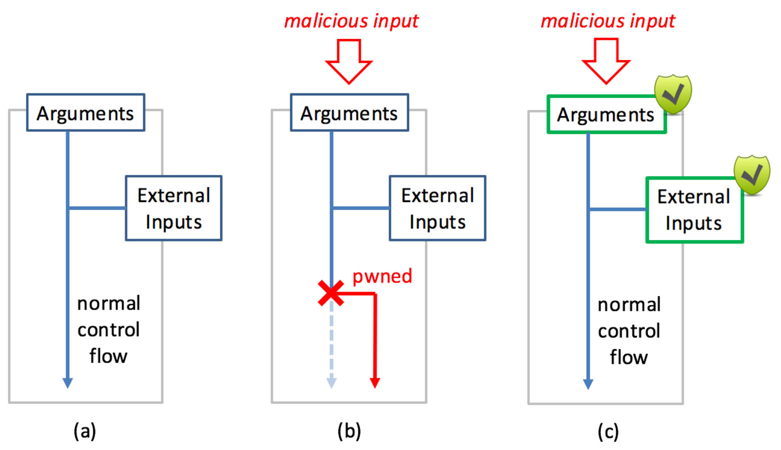


Figure 8: Kernel functions take input from either arguments or external data reading (e.g., `copy_from_user`) as shown in (a). Most Android kernel exploits inject malicious input into the target function and hijack the control flow to achieve arbitrary code execution, as shown in (b). So, by hooking the data input entries and validating the input as shown in (c), we can block most of the kernel exploits.

To be more specific, 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.

The first policy is straightforward. It forces the patch developer to explicitly claim the target function, and only checks the arguments passed into it. The patch reviewer can thus immediately understand which function is to be hooked and why. The third and fourth policies are also reasonable, which provide the capability to detect attacks but preventing modifications.

But what is the meaning of the second restriction? To make it intuitive, a running example is shown in Figure 9. The vulnerable function `fun` calls data reading functions `foo` and `bar`. The function `foo` reads data into a buffer and returns success or failure, while `bar` returns a pointer to a structure. After invoking `foo`, `fun` only uses a conditional statement[1] to handle the return value of `foo`, so `foo` satisfies the rule. On the contrary, after invoking `bar`, `fun` invokes a function pointed by a field in the structure returned by `bar`, so `bar` fails to satisfy

[1] Conditional checks include statements like `if/else`, `switch/case`, `do/while`, `for`, etc.

the restriction. As a result, we allow the patch to hook the invoking point and the returning point of `foo`[2], but forbid hooking those of `bar`.

```
fun(...) {
    // entry of A can be hooked
    bool result;
    struct *s;

    // foo is allowed to be hooked
    result = foo(...);
    if (result == E_INVALID)
        return;

    // bar cannot be hooked
    s = bar(...);
    if (s)
        s->fun();
}
```

Figure 9: A running example to illustrate which functions can be hooked and which cannot

We generate a whitelist of kernel functions whose return value is a status code based on general kernel source code. Only functions on this whitelist can be hooked within the target function. With this restriction, we eliminate return value based control flow manipulation. Another great benefit of this restriction is that when the validator detects a malicious input and returns an error code, the conditional check afterwards will naturally switch to the error handling branch. This not only avoids triggering the vulnerable code branch, but also avoids crashing the kernel.

We call vulnerabilities that can be fixed by simply hooking the vulnerable function entry as *Type I vulnerabilities*. Those that need to also hook invoked functions are classified as *Type II vulnerabilities*. In Section 5.3, we will show the numbers of these two different types of vulnerabilities.

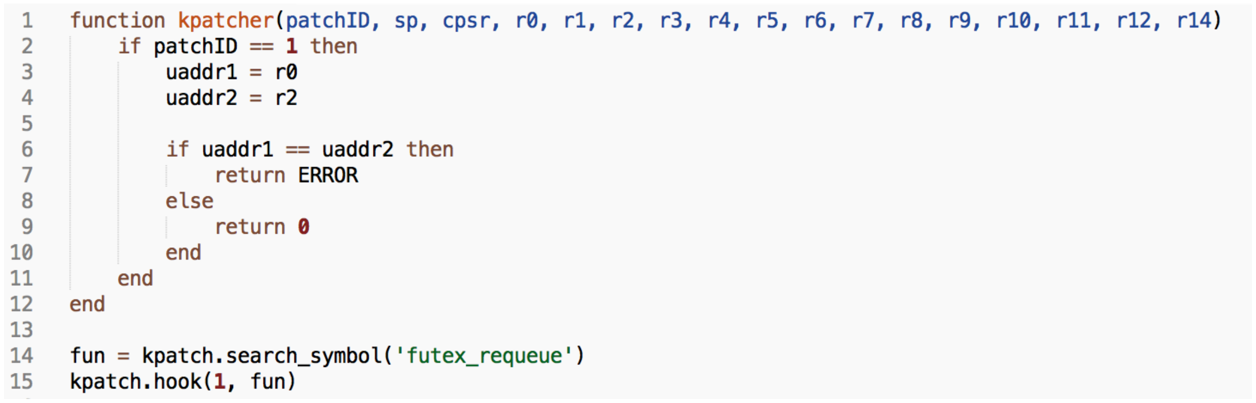
5.2 Implementation

We implement the memory-safe dynamic language engine based on Lua following the restriction rules described above, named as *LuaKpatch*. Lua certainly satisfies memory safety and has many dynamic flexibilities. It is tiny, fast, simple, easy to extend, with great error handling, and has the best integration with C. In order to integrate it into the Android kernel, we followed many practices from the *lunatik-ng* project [19] . For example, numbers in the original Lua implementation are in floating-point type. But Linux kernel does not support floating-point arithmetic, and patches barely rely on it. So numbers should be defined as integers. Moreover, unnecessary Lua libraries like file operations are stripped away to reduce the code base size.

[2] Note that the patch hooks `fun`'s calling instruction and the returning site, not `foo`'s entry instruction and returning instruction. This avoids all the invocations to `foo` get hooked, which is expensive.

The Line-of-Code (LoC) of LuaKpatch is roughly 11K. Among them, 10K are Lua engine's code, and the core patching logic only takes less than **600 LoC**. LuaKpatch is compiled as an **800KB** kernel module. It can be pre-shipped in the kernel if the phone vendor cooperates. Otherwise, it can be shipped as a standalone kernel module and can be loaded using the adaptive method introduced in AdaptKpatch. Once loaded into the kernel, it launches a Lua VM instance and uses a kernel work queue to process patch requests. We implement the following interface families for Lua to operate on:

1) Symbol searching: take a symbol string and return the address.
2) Hooking: take an address and hook on it. The hooking is achieved by inserting a trampoline. Once triggered, the trampoline saves the current context (registers, stack, etc.) and invokes the designated Lua patching handler.
3) Typed reading: take an address, validate if the address is readable, then return the typed value to Lua.
4) Thread info fetching: return the current thread information, such as thread id, etc.



```
 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)
```

Figure 10: Sample Lua patch to fix one of the vulnerable conditions of CVE-2014-3153, known as "Towelroot"

Based on these APIs, the patch can be as simple as shown in Figure 10. On Line 14, the vulnerable function `futex_requeue` is targeted, and gets hooked on Line 15. The hooking process inserts a trampoline at the entrance of this function, which saves the context (registers and stack content) and invokes the `kpatcher` handler. The handler recognizes the corresponding check via the designated patch ID, and performs the necessary checks based on the saved context. If vulnerable conditions are detected, the patch handler returns an error code; otherwise it returns zero. When the Lua patch handler returns, the execution turns back to the trampoline. In case of error, the trampoline returns the error code to the caller of `futex_requeue`; otherwise it resumes normal execution into `futex_requeue`.

Similar to AdaptKpatch, LuaKpatch only performs hot patching, so it does not conflict with the static integrity protection mechanisms like Secure Boot. On rebooting, the framework should be launched in the very early stage to reload the patches. Also, it needs to fall back to normal booting if hot patches cause problems.

5.3 Efficacy Evaluation

We collect recent well-known Android kernel vulnerabilities with public exploits or crashing PoC in Table 2. It turns out that all the vulnerable conditions of them can be captured by LuaKpatch. The percentages of Type I and Type II vulnerabilities are shown in Figure 11. The majority of the vulnerabilities can be fixed simply by hooking and performing input validation at the function entry.

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

Table 2: Android root vulnerabilities that have been verified to be protectable by LuaKpatch. Most of the vulnerabilities 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).

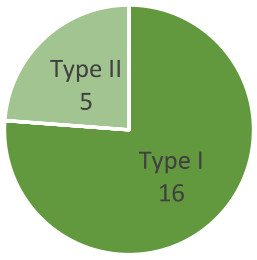


Figure 11: All 21 vulnerabilities that we collected can be patched by LuaKpatch. Among them, 16 are Type I vulnerabilities, and 5 are Type II vulnerabilities. So 76% of the vulnerabilities can be fixed by hooking and checking input at the function entry.

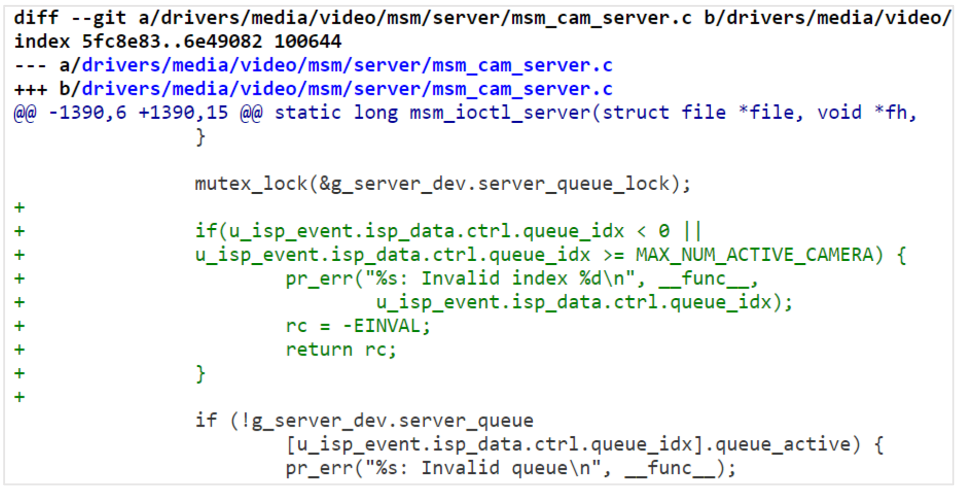
For example, for CVE-2013-1763, the source code patch is shown in Figure 12. LuaKpatch can patch it by hooking the entry of the `__sock_diag_rcv_msg` function, getting the `nlh` argument, obtaining `req` from `nlh`, and then checking whether the condition `req->sdiag_family >= AF_MAX` is satisfied. If this is true, it is an exploit condition and the patch should return an error.

```
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;
```

Figure 12: The source code patch for CVE-2013-1763

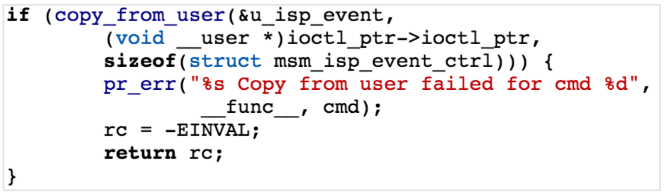
As another example, the source code patch for CVE-2013-6123 is shown in Figure 13. The checking condition relies on the content of `u_isp_event`, which is obtained from the user space (shown in Figure 14). So LuaKpatch cannot simply validate the exploit condition by hooking the entry of `msm_ioctl_server`. Luckily, `copy_from_user` returns status code, so LuaKpatch allows to hook `copy_from_user` invocations within `msm_ioctl_server`. The remaining checks are straightforward so we omit the elaboration.



```
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__);
```

Figure 13: The source patch for 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;
}
```

Figure 14: `u_isp_event` is obtained from copy_from_user

However, not all vulnerabilities can be perfectly solved. For example, for CVE-2015-3636 (known as "*PingPong Root*"), the official patch is to add `sk_nulls_node_init` into `ping_unhash`. LuaKpatch does not allow patches to achieve "*hook-anywhere*" and "*arbitrary-code-execution*" for security consideration. So, as an alternative the patches should insert an argument validation at the function entry of `ping_unhash`, checking if `sk->sk_nulls_node.pprev` equals to 0x200200 (`LIST_POISON2`). If this exploit condition is detected, the patch returns an error without executing into `ping_unhash`. Note that this introduces side effects, because the intended `ping_unhash` functionalities are dropped in case of exploits. Luckily, kernel exploits usually target at code paths that common tasks do not trigger (e.g., `ping_unhash` on Android is barely invoked), and this kind of side effects does not bring in serious consequences. We did an experiment by applying this patch on different devices (Nexus 5, Samsung S6, etc.) and verified that the PingPong Root can be successfully defended against. After two weeks' normal usage (Internet browsing, playing games, etc.) without rebooting, these devices are still running smoothly without a problem. After all, having some none-crashing side effects such as memory leakage is way better than being exploited. Even in the worst situation, we can fall back to the normal AdaptKpatch mechanism.

5.4 Performance Evaluation

LuaKpatch inserts extra validation checks into vulnerable kernel functions, but since the exploited functions are usually not frequently invoked (heavily executed code usually undergoes more thorough testing), the performance should not be impacted too much.

We evaluated the whole system performance overhead using CF-Bench [20] on Nexus 5 with Android 4.4. Four conditions were tested for comparison:

1) before getting any patches
2) with the Towelroot vulnerability patched by LuaKpatch
3) with the Ping Pong Root vulnerability patched by LuaKpatch
4) with both vulnerabilities patched

The average performance score was obtained from 20 measurements in each test. The result is shown in Figure 15 – there is no observable overhead incurred by LuaKpatch. This is as expected since the patched functions are barely called during Android normal execution.

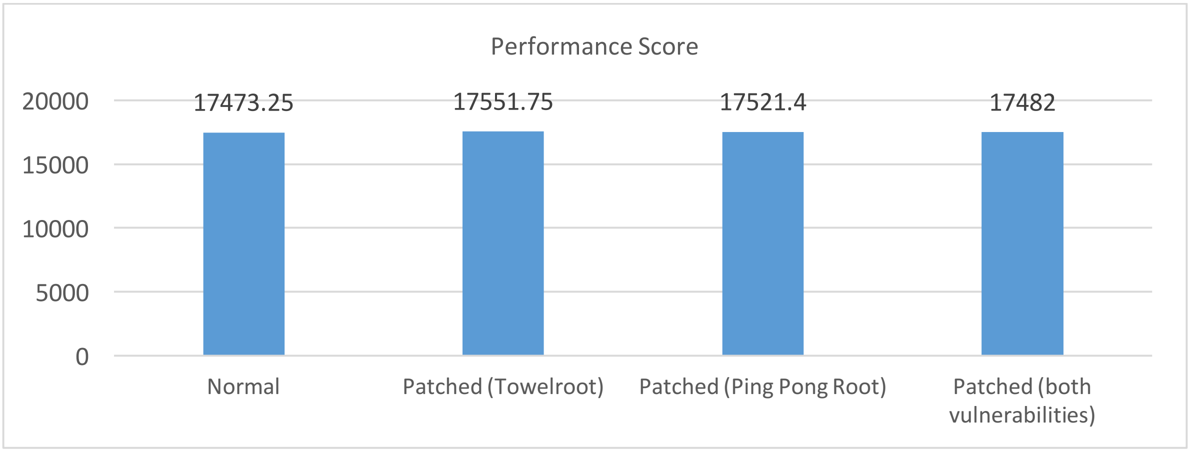


Figure 15: Performance scores obtained from CF-Bench. The scores have small deviations, which are within CF-Bench's normal fluctuation range.

Nevertheless, we are still interested in the overhead of LuaKpatch in the worst cases. More specifically, we want to measure the overhead of each LuaKpatch validation. To do so, we measured the execution time of `chmod` system call before and after patching it with a series of dummy patches. The system call is explicitly invoked for 1000 times for each measurement, and the average time is calculated from 20 measurements.

As shown in Figure 16, when no patch is applied, the system call itself takes about 100.7 microseconds. Then we applied four testing patches to measure the cost of LuaKpatch:

1) When applied with a patch that simply returns zero, the overhead is 0.42 microseconds. This reflects the cost of the trampoline execution time, namely the context saving/restoring plus the kernel-to-LuaKpatch/LuaKpatch-to-kernel transition time.
2) When applied with a patch that contains an "`if/elseif/else`" condition statement. This patch adds an overhead of 0.98 microseconds to the system call.
3) When applied with a patch which consists of a memory read, it adds an overhead of around 0.82 microseconds to the system call.
4) Lastly, to simulate a complicated situation, we wrote a patch with a mixture of assignments, memory reads and conditional statements. The overhead is about 3.74 microseconds. That is less than 4% of the normal `chmod` execution time.

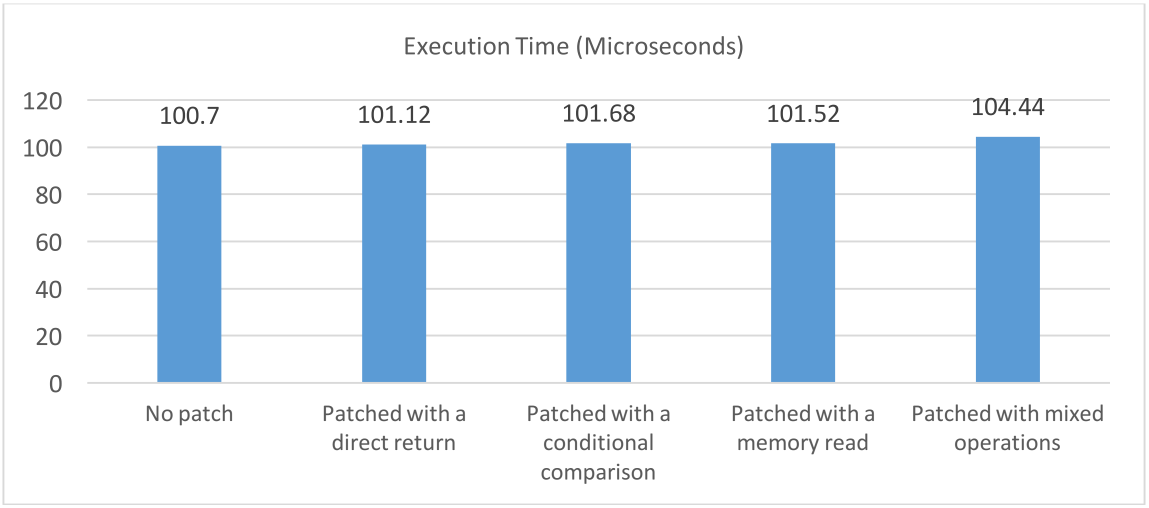


Figure 16: Execution time of `chmod` with/without the patches

From the above experiments, we can see that a common LuaKpatch validation check adds an overhead under 5 microseconds. Because system calls are not invoked all the time, the impact to the overall system performance should be even less. We counted all system call invocations on the Nexus 5 with Android 4.4 for 1 minute when a user normally browses Internet using Chrome. Among all the triggered system calls, `gettimeofday` was the mostly-called one, which got triggered for about 110,000 times. The overall performance overhead can be estimated as 5µs*110,000/1min ≈ 0.9%, which is quite small.

To summarize, LuaKpatch only introduces negligible performance overhead. As an ongoing work, we are migrating LuaKpatch to LuaJIT [22] , which should further improve the performance.

6. Conclusion

In this work, we discussed why Android kernel vulnerabilities can last so long. To tackle this problem, we introduced LuaKpatch and AdaptKpatch. Both frameworks can save developers from tedious patch porting work. Also, because they are open platforms, patches can be provided from various vendors. As a result, Android kernel patching circle can be greatly shortened.

As shown in Figure 17, with LuaKpatch, critical vulnerabilities can be patched in a few days. Before the cold patches are rolled out, or in case some vulnerabilities cannot be perfectly solved by LuaKpatch, AdaptKpatch can be utilized to fix the issues.

LuaKpatch is definitely safer than AdaptKpatch due to more restricted constraints. Strange as it may sound, both LuaKpatch and AdaptKpatch are much safer than the traditional cold patching. This is because cold patches are permanent modifications. If there is something wrong with the patches, millions of devices will be bricked. LuaKpatch and AdaptKpatch, however, only load patches lively on each reboot, and can easily fall back to normal booting if issues are encountered.

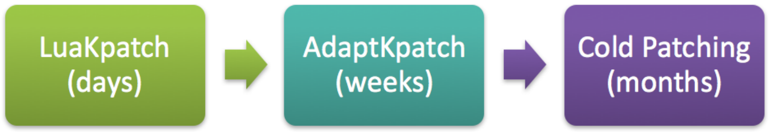


Figure 17: The patching circle in the open collaborative patching ecosystem. From left to right, it takes more time for patches to land on user devices to take effect. And from left to right, the safety level also decreases. Note that AdaptKpatch can perfectly fix more issues than LuaKpatch, so when LuaKpatch is not an ideal choice, AdaptKpatch can be a great alternative.

Based on LuaKpatch and AdaptKpatch, we call for the Android ecosystem to establish a patching alliance. This alliance consists of device vendors, security vendors, and other parties in the Android open source communities. This alliance will be of great significance, in that:

1) The number and the complexity of kernel vulnerabilities keep increasing, so more joint effort makes it easier to battle against them.
2) In the AdaptKpatch scheme, patches can be vetted and cross-validated by qualified alliance members.
3) 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.

Every party in the ecosystem should be well motivated – the device vendor will have more secure products thus more users and sales without overhead spent on security R&D. Security patch providers will gain reputation and profits (rewards and brand fame) on the other hand.

Finally, this framework can be easily extended and applied to general Linux platforms. We believe that improving the security of the whole ecosystem is not the dream of our own. We call for more and more parties to join in this effort to fight the evils together.

References

[1] http://www.cmcm.com/blog/en/security/2015-09-18/799.html

[2] https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html
[3] https://www.bluecoat.com/security-blog/2016-04-25/android-exploit-delivers-dogspectus-ransomware
[4] https://blog.checkpoint.com/wp-content/uploads/2016/07/HummingBad-Research-report_FINAL-62916.pdf
[5] Hang Zhang, Dongdong She, and Zhiyun Qian. Android Root and its Providers: A Double-Edged Sword. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)
[6] https://bugs.chromium.org/p/project-zero/issues/detail?id=734&can=1&sort=-id
[7] http://www.howtogeek.com/163958/why-do-carriers-delay-updates-for-android-but-not-iphone
[8] http://opensignal.com/reports/2015/08/android-fragmentation
[9] https://developer.android.com/about/dashboards/index.html
[10] https://github.com/torvalds/linux/commit/e9c243a5a6de0be8e584c604d353412584b592f8
[11] https://source.android.com/security/bulletin/2015-09-01.html
[12] https://github.com/torvalds/linux/commit/f0d1bec9d58d4c038d0ac958c9af82be6eb18045
[13] http://source.android.com/security/advisory/2016-03-18.html
[14] https://github.com/dynup/kpatch
[15] https://www.ksplice.com/doc/ksplice.pdf
[16] http://events.linuxfoundation.org/sites/events/files/slides/kGraft.pdf
[17] http://lxr.free-electrons.com/source/include/linux/livepatch.h
[18] https://conference.hitb.org/hitbsecconf2016ams/sessions/adaptive-android-kernel-live-patching
[19] https://github.com/lunatik-ng/lunatik-ng
[20] https://play.google.com/store/apps/details?id=eu.chainfire.cfbench&hl=en
[21] https://httpd.apache.org/docs/2.4/programs/ab.html
[22] http://luajit.org