Automatic Hot Patch Generation for Android Kernels

USENIX Security 2020

Hot Patch

- A.k.a *live patching* or *dynamic software updating.*
- Application of patches to the binary without restarting it.
- Address problems related to unavailability of service provided by the program.
- Block the malicious input to the function to ensure the security.

Hot patch for Android

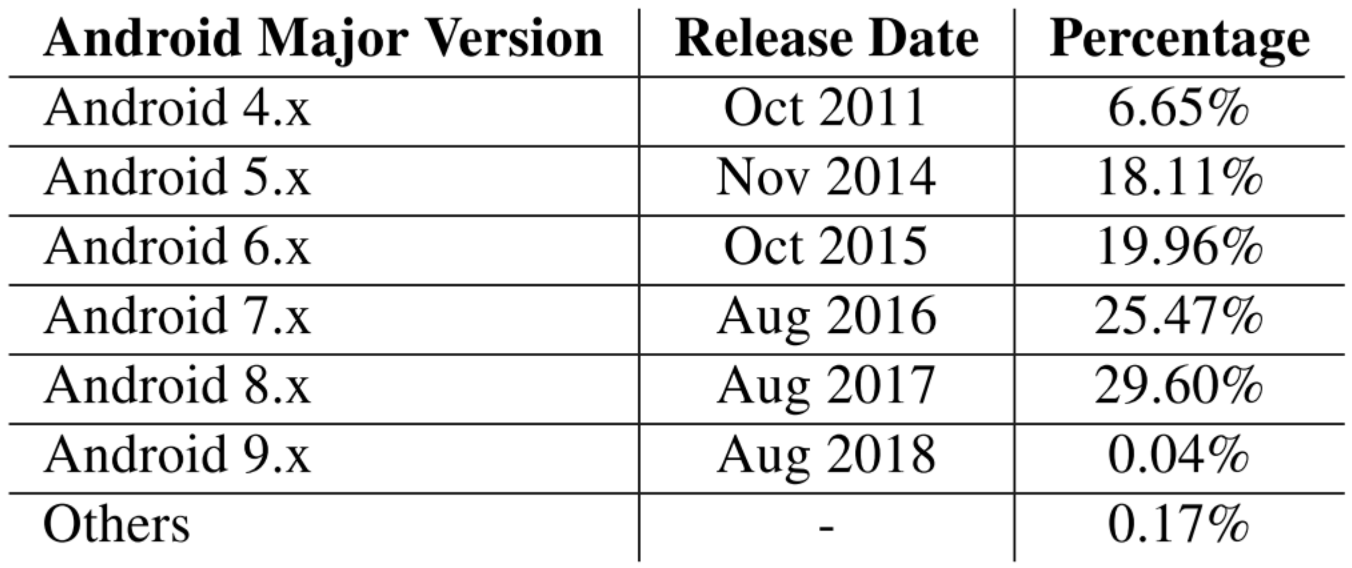
Motivation:

Android system has low upgrade rate (legacy systems).
Known vulnerabilities remain unfixed.
It is time consuming to develop patches manually.

Hot patch is a good way for such issues.

Previous work manually write hot patches with excessive human labor.

Table 1: Android version distribution (OCT 2018)



Android Major Version	Release Date	Percentage
Android 4.x	Oct 2011	6.65%
Android 5.x	Nov 2014	18.11%
Android 6.x	Oct 2015	19.96%
Android 7.x	Aug 2016	25.47%
Android 8.x	Aug 2017	29.60%
Android 9.x	Aug 2018	0.04%
Others	-	0.17%

Goal:
Input Fiter Generation

Given a vulnerable function F and its official patch P at location L, we would like to find a suitable location L_0 of F in binary form to insert an automatically generated hot patch P_0, which has the same semantics as P.

Why use filter rather than direct patch?

- In binary, it is easier to locate function beginning than other places.
- In different legacy systems, function beginning remains the same.

To ensure that the filter is safe, we limit the filter to only read the memory content without write operation.

Vulnerability Patch Survey:

Patch type classification on 375 CVEs of Android Kernel between year 2012 -2016.

Patch Type	NO.	Percent
Sanity Testing	157	42.10%
Function Calling	65	17.40%
Change of Variable Values	37	9.90%
Change of Data Types	9	2.40%
Redesign	65	17.40%
Others	40	10.70%

Observations

- Most patches do minor modification of the code, e.g., use less than 30 lines; most patches only modify one function.
  - 309/373
- Large patches can be divided into small ones.
  - 50/64

What patches are suitable for hot patches?

- Only include data read rather than write

- Other restrictions in this work
  - Only modify (insert patch) at the beginning or end of functions in the binary form.
  - One patch spans only one function with small changes, or several such patches.

- Therefore, this work focuses **sanity testing** and part of **function call** patches.

Approach – Where to insert patches?

- Function beginning or end with least “side effects”.
- Two slices:
  - Path1: Function beginning to candidate insertion location.
  - Path2: Candidate insertion location to official patch location.
- Investigate Path2 to check whether equivalent semantics can be preserved.
  - Backtrack variables used in official patches.
- Exam Path1 for less side effects introduced by the hot patch.
  - The side effects include the change of the global variables, the assignment of pointers, the allocation of a piece of memory without freeing it, as well as any of the calling to the system functions.

Approach – Patch construction

- Official patches and hot patches are in different locations.
- Construct the relationship between the variable values used in the official patch and the hot patch.
- Weakest precondition reasoning (Symbolic execution).
- Track the value relationship among code statements, e.g., if-else, loop, function calls.

```
F(x){
   // hot patch
   y = x + 1;
   if(y > 0) {
      // official patch
      validate(y+1);
   }
}
```

```
// hot patch
If(x + 1) {
    validate(x + 1 + 1);
}
```

```
F(x){
   // hot patch
   if(x = 10) {
      y = x + 1;
   }
   else{
      y = g(x) ;
   }
   if(y > 0) {
      // official patch
      validate(y+1);
   }
}
```

Function calls

- Irrelevant function calls – skip
  - Do not relate to the variables in the patches.
- Relevant function calls
  - Involve data write – skip
  - Nested function call – skip

- Relevant function calls without nested calls to other functions, and without data write.

Loops

- Wrap the official patch
    - Construct a corresponding loop to apply the hot patch.
- Irrelevant loops -- skip
    - Do not relate to the variables in the patches, or not in the path.
- Complex loops and multi-layers of loops are ignored.
- Relevant loops
    - Use loop summarization for the range of variables.
    - Possibly sacrifice normal functionality and pick the maximum range.

Approach -- How to apply to binary?

- Generate an additional function with the same parameter as the target function to be inserted.
- The addresses of variables used in the hot patches can be inferred because they are all about the parameters.
- Compile into a binary.
- Load the hot patch binary into the memory and build a trampoline at the hooking point to direct the control flow of the program to the loaded patch.

Evaluation:

- Accuracy:
  - Manually verified: 55 (out of 59) correct generated patches.
- Performance:
  - Less than 0.1% in overhead.
- Robustness:
  - 21 cases tested. All of them pass the testing.
- Other:
  - 54 (out of 55) generated patches have similar logic with human written patches.

Comments

- Writing logic
  - The section 2.3 Operation scopes reads like the limitation of the tool, why list it as “rules” of hot patch?
  - Rule 1 is due to the inherited challenge of binary analysis.
  - Rule 2 can be the research scope. Write operations are not proven to be infeasible. Place it after the patch analysis.
  - Rule 3 is just a limitation.
- What if the customized Android system is slightly different from the patch system? Why here only analyze the official patches? Can it be directly applied? Seems contradicted with the motivation. Evaluation does not provide necessary evidence.

Comments

- Technical contributions.
  - The problem is relatively easier than symbolic execution, e.g., function call, loops.
  - How hot patch binary is integrated is out of the scope of this work.
- Correctness evaluation.
  - Defending only one PoC exploit or manual check does not guarantee the correctness of the patch? Any formal method?
- Effects of inaccurate function call/loop handling.