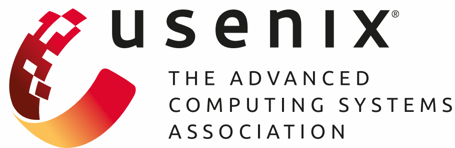

Automatic Hot Patch Generation for Android Kernels

Zhengzi Xu, *Nanyang Technological University;* Yulong Zhang, Longri Zheng, Liangzhao Xia, and Chenfu Bao, *Baidu X-Lab;* Zhi Wang, *Florida State University;* Yang Liu, *Nanyang Technological University*

https://www.usenix.org/conference/usenixsecurity20/presentation/xu

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Automatic Hot Patch Generation for Android Kernels

Zhengzi Xu
Nanyang Technological University

Yulong Zhang
Baidu X-Lab

Longri Zheng
Baidu X-Lab

Liangzhao Xia
Baidu X-Lab

Chenfu Bao
Baidu X-Lab

Zhi Wang
Florida State University

Yang Liu
Nanyang Technological University

Abstract

The rapid growth of the Android ecosystem has led to the fragmentation problem where a wide range of (customized) versions of Android OS exist in the market. This poses a severe security issue as it is very costly for Android vendors to fix vulnerabilities in their customized Android kernels in time. The recent development of the hot patching technique provides an ideal solution to solve this problem since it can be applied to a wide range of Android kernels without interrupting their normal functionalities. However, the current hot patches are written by human experts, which can be time-consuming and error-prone.

To this end, we first study the feasibility of automatic patch generation from 373 Android kernel CVEs ranging from 2012 to 2016. Then, we develop an automatic hot patch generation tool, named VULMET, which produces semantic preserving hot patches by learning from the official patches. The key idea of VULMET is to use the weakest precondition reasoning to transform the changes made by the official patches into the hot patch constraints. The experiments have shown that VULMET can generate correct hot patches for 55 real-world Android kernel CVEs. The hot patches do not affect the robustness of the kernels and have low performance overhead.

1 Introduction

Android platform has become the biggest mobile platform in the modern mobile device industry. The rapid growth of the Android ecosystem makes our lives convenient by bringing us thousands of new devices with various (customized) Android operating systems. However, most of these devices cannot receive timely updates. Table 1 gives the Android version distribution from 500 million devices as of October 2018[1]. The table shows that the recent release of Android Pie (9.0) in August 2018 reaches only very few devices after two months. However, from the August 2018's monthly release, the Android Security Bulletin [1] stopped to carry security patches for Android 6.x and below. As a result, based on the statistics in Table 1, 44.72% of Android devices will not receive any security patches unless vendors can upgrade the firmware themselves. Fig. 1 provides further detailed analysis of the Android patch level of the same 500 million devices. Only 20% of the devices can catch up with the 3-month-old security patch updates; only 60% of the devices can catch up with the 6-month-old security patch updates; and 20% of the devices only have security updates more than a year ago.

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%

The low upgrade rate has resulted in legacy Android systems with unpatched vulnerabilities. However, Android vendors are not motivated to fix those vulnerabilities. It is costly to apply changes to kernels, as it requires to go through tedious testing process to ensure that the changes do not break existing functionalities [18]. Therefore, the legacy systems will remain vulnerable for a very long period. Attackers can leverage the known vulnerabilities to attack easily.

To address this known vulnerability threat, tremendous efforts have been made to patch old Android systems. Among all the possible solutions, the hot patch technique provides a convenient way to fix the vulnerabilities without interrupting the normal functionalities of the program [47]. It greatly improves the user experience since it can ensure the system security without rebooting the devices. Based on the hot patch idea, Chen *et al.* have proposed an adaptive Android kernel live patching framework [14]. The framework hooks the vulnerable function and applies a pre-constructed hot patch to it.

[1]With user consent, we collected Android versions and patch levels from devices with the Baidu app installed.

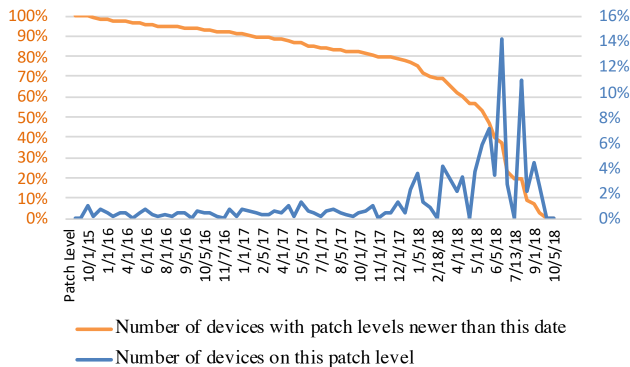

Figure 1: Android patch level distribution as of Oct 2018

The hot patch will block the malicious input to the function to ensure the security. Mulliner *et al.* [38] have built a framework, named PatchDroid, which can insert the hot patches into the binary for third-party unsupported Android kernels. It checks constraints during patching the vulnerabilities to ensure the stability of the system. These works try to build the framework to insert the hot patch into the legacy system. However, the hot patches are still needed to be provided to the system. The framework cannot generate the patches automatically.

Writing hot patches based on the officially released patches is challenging. At source code level, programmers can modify any parts of the vulnerable functions to fix the bug, while in binary, it is difficult to find exactly the locations to place the same modifications. To get a hot patch in the suitable place, security experts are required to understand the semantics of the official patch and write a corresponding hot patch. However, this is a time-consuming process and error-prone. It is not acceptable for the IT industry with the fast development cycle and limited security budgets. Therefore, there is a need to develop an automatic solution to correctly convert an official patch into a hot patch.

To this end, this paper proposes a solution to automate the hot patch generation process. We propose a proper definition of the problem and set the requirements and assumptions that involved. To have a complete understanding on the vulnerability patches, we first analyze most of the Android CVEs from the year 2012 to 2016 and categorize them based on their patching behaviors. With the insight from the analysis, we develop VULMET, which can automatically generate hot patches by extracting the semantics of the official patches using program analysis. VULMET will find a suitable place in the function, build and apply a semantic equivalent hot patch to fix the vulnerability. To test the effectiveness of VULMET, we have generated the hot patches for real-world Android CVEs. The hot patches can prevent the exploits with only little overhead on the system.

Overall, this paper has made the following contribution:

1. We formally define the process of automatic hot patch generation via learning from the semantics of the official patches. We elicit three requirements of the process and define its operation scopes.

2. We conduct an empirical study by collecting, summarizing, and categorizing different real-world Android kernel vulnerability patches based on their behaviors and distill four insights.

3. We propose an approach to automatically generate hot patches, and implement a tool, named VULMET, to simulate the hot patch generating process and test its performance using the vulnerabilities in the real-world legacy Android system. The experiments show that the generated hot patch can fix the vulnerabilities with low overheads.

The rest of the paper is organized as follows. In Section 2, we define the automatic hot patch generation problem with a real-world example. Next, in Section 3, we conduct a survey of Android vulnerability type and define the scope of the patch which can be used to generate the hot patch. Then, Section 4 presents the automatic hot patch generation framework. Section 5 evaluates of different aspects of VULMET with different experiments. Section 6 lists the related works in patch generation. At last, Section 7 concludes the paper.

2 Automatic Hot Patch Generation

In this section, we define of the automatic hot patch generation problem, state requirements and assumptions, and illustrate it with an example.

2.1 Problem Definition

We define the automatic hot patch generation as follow:

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

In this work, to achieve the goal of hot patch generation, we have conducted a vulnerability and patch survey to collect the vulnerable functions F with the official patches P. Then, we develop VULMET to automatically set up the metrics to measure whether a location is suitable to insert the hot patch and select the most suitable one as L'. After that, VULMET leverages the weakest precondition to transform the constraints of the original semantics into new constraints to form the hot patch P' at L'.

2.2 Requirements

To ensure the generated hot patches are practical, we have set the requirements to measure whether it is suitable to patch Android kernels.

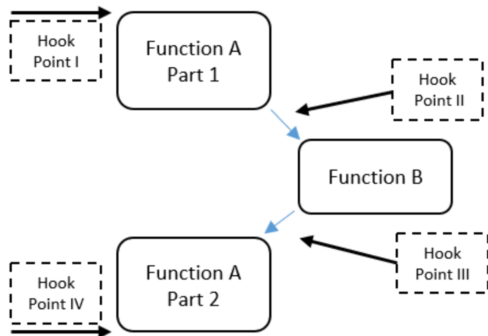

Figure 2: Example of the special case for Rule 1

Requirement 1: the generated hot patch should preserve the semantics as the corresponding official patches, which guarantees its correctness.
Requirement 2: the generated hot patch should not break the system, which ensures its robustness.
Requirement 3: the generated hot patch should incur low overhead, which ensures its efficiency.

2.3 Operation Scopes

To ensure the robustness of the patched program, we have defined three rules to limit the operations used in the hot patches.

Operation Rule 1. The patch can only be placed at the beginning or end of the functions or at the beginning or end of function calls.

In binary executables, the function level information is limited. For a given source code statement, it is difficult to locate a particular line of binary instruction. In addition, the instruction location changes in different versions of the functions. However, no matter what changes have been made inside a function, its boundary remains the same. With the help of IDA PRO [10], the function beginning and end place can be pinpointed. These places are stable even if the contents of the function have been slightly changed. Therefore, to ensure the hot patch is practical in Android kernels, we only allow the patch to be placed at the beginning or end of a function or the call of the function.

`Operation Rule 1` has one special situation, which has been shown in Fig. 2. In the case where function A calls function B, the hot patch can be applied to the beginning or end of function A (Hook Point I and IV) and the call to function B (Hook Point II and III). By hooking at the beginning or end of the call to function B (Hook Point II and III), we can achieve the equivalent semantics as if we are hooking in the middle of function A. Therefore, the hot patch still obey `Operation Rule 1`, but looks like to have the ability to hook in the middle of the function.

Operation Rule 2. The patch can read the valid content of the memory but it is prohibited from modifying the contents.

```
1  int q6lsm_snd_model_buf_alloc(struct lsm_client *client,
2      size_t len){
3          struct cal_block_data *cal_block = NULL;
4          size_t pad_zero = 0, total_mem = 0;
5          ...
6          cal_block =
7          cal_utils_get_only_cal_block(lsm_common.cal_data);
8          if (cal_block == NULL)
9                  goto fail;
10         ...
11         if (!client->sound_model.data) {
12                 client->sound_model.size = len;
13                 pad_zero = (LSM_ALIGN_BOUNDARY -
14             (len % LSM_ALIGN_BOUNDARY));
15 +                if ((len > SIZE_MAX - pad_zero) ||
16 +                (len + pad_zero >
17 +            SIZE_MAX - cal_block->cal_data.size)) {
18                         ...
19                         goto fail;}
20         ...}
21 ...}
```

Figure 3: Official Patch for CVE-2015-8940

Modifying the memory contents directly may be dangerous. A careless writing operation may change the program control follow and tamper the important data, which will result in unexpected behaviors of the program. Therefore, to enforce the security, we restrict the patch operation to be only reading the content without writing to the memory.

Operation Rule 3. The patch can only fix vulnerability with small changes and within one function.

Patches are usually small to address a specific security problem. If the patch modifies most parts of the function, it is equivalent to implement a new function. Hot patch has the limitation to fix this kind of bugs without introducing other problems [47]. Moreover, a large cross-function patch is rare and may involve the redesigning of the program logic, which is not suitable for hot patching. Therefore, we limit the hot patch to fix vulnerability within one function. However, if a large official patch can be divided into small patches within one function, the small patches can be converted into hot patches separately.

2.4 Real-world Example

In this section, a real-world example is given to demonstrate the concept of converting the official patch into the hot patch.

Fig. 3 has shown the official source code patch for `CVE-2015-8940` [6] in Android Qualcomm msm kernel 3.10. This patch fixes the integer overflow bug in function `q6lsm_snd_model_buf_allo()` by adding a sanity check at line 15 to 17.

To convert it into a hot patch, we first follow `Operation Rule 1` to hook the beginning of the function `q6lsm_snd_model_buf_allo()` at line 1. Then we need to find a semantic equivalent patch as the official patch at this point. The official patch contains one sanity check of variable `len`, `SIZE_MAX`, `pad_zero`, and `cal_block->cal_data.size`. Not all of these variables' values are known at the begin-

Table 2: Variable Relationships

Patch Variable	Equivalent Value
len	len (same as function input)
SIZE_MAX	constant
pad_zero	constant - (len % constant)
cal_block->cal_data.size	cal_utils_get_only_cal_block()

ning of the function. To build an semantic equivalent patch, we need to use variables whose values are known to represent the same sanity check. Since we are hooking at the beginning of the function, we can get the value of the function input parameters, `client` and `len`. Then we need to use weakest precondition reasoning, a program analysis technique, to find out the relationships between the input parameters and the sanity check variables.

Table 2 shows the relationships between the variables. The detailed algorithm to determine the relationships automatically is presented in Section 4. With them, we can generate an equivalent sanity check at the beginning of the program by replacing the official patch variable with the variables. The generated equivalent sanity check looks like:

```
1  if ((len > constant1 - (constant2 - (len % constant2)))
2          || (len + ((constant2 - (len % constant2))
3          > constant1 - func_return_value))
4           {return 0;}
```

The generated patch will only read the contents of the function inputs without any writing operation so that the `Operation Rule 2` is satisfied. Moreover, since the patch only fixes the vulnerability in one function, `Operation Rule 3` is also satisfied. Therefore, the generated patch complies with the operations in the definition.

3 Patch Type Analysis

To generate the hot patch from an official patch, we need to make sure that the official patch fixes the vulnerability in certain ways, which are able to be converted to a hot patch. Therefore, we conduct an empirical study on the different types of Android vulnerabilities. In the study, we provide the vulnerability patch categorization and distribution results as well as the insights found from the observations. After that, we are able to discuss the type of vulnerability patch that VULMET is able to support.

To have a comprehensive understanding of different types of patches, we have manually analyzed the recent Android kernel CVE vulnerability patches. We make an effort to collect most of the Android kernel vulnerabilities, which are publicly disclosed by Google. As VULMET works on the legacy vulnerabilities, we choose CVEs from the year 2012 to 2016, which mainly reside in Linux major version 3. We also ignore the older vulnerability, since it has a low chance to affect the recent Android devices.

Table 3: Patch Type Categorization

Type	Sub Type
Sanity Testing	Precondition Validation Error Handling
Function Call	Ensuring Atomicity Freeing Resources Call User Define Functions
Change of Variable Values	Zeroing Memory Initialization Increase Buffer Size
Change of Data Types	n.a.
Redesign	n.a.
Others	n.a.

3.1 Patch Categorization

Since our work focuses on the patch generation, the patch category should reflect the modifications to the function code rather than the consequence of the vulnerabilities. There are many patch categorization works on classifying the patches based on the type of the vulnerability they are fixing [51] [25] [56] [48]. However, only few works focus on the patches themselves. [36] has proposed a categorization schema based on the patch modification, which fits our need well. Therefore, we adopt the idea of this work and combine some of their patch type groups to form our patch categorization schema. The different patch modification category is listed in Table 3. `Sanity Testing` checks a certain condition and makes the decision to change the program control flow. Based on the different variable values it checks, `Sanity Testing` can be further divided into two subgroups. `Precondition Validation` type tries to check the function input parameters, and `Error Handling` tries to check the return value of the function call to add the error handling logic to the program.

The `Function Calling` patch type fixes the vulnerability via calling the functions. Based on the different function it calls, it can be divided into three subtypes. `Ensuring Atomicity` adds in the calls to synchronization functions such as `lock()` and `unlock()` to ensure the atomic operations. `Freeing Resources` calls the `free()` function to remove the unused resources. `Call User Define Functions` includes other function calls to achieve different purposes.

The `Change of Variable Values` patch type requires the modification of the memory contents. `Zeroing Memory` sets the memory to 0 to prevent information leak. Some of the `Zeroing Memory` patches are implemented using function call to `memset()`. We regard this type as the `Zeroing Memory` not the `Function Call`. `Variable Initialization` sets a default value to the variable. `Buffer`

Table 4: Patch Type Allocation

Type	NO.	Percent	Example
Sanity Testing	157	42.1%	CVE-2014-3145
Function Calling	65	17.4%	CVE-2014-8709
Change of Variable Values	37	9.9%	CVE-2014-1739
Change of Data Types	9	2.4%	CVE-2016-2062
Redesign	65	17.4%	CVE-2016-8457
Others	40	10.7%	CVE-2014-9683

Table 5: VULMET support patch types

Type	Supported / Unsupported
Sanity Testing	Supported
Function Call	Partially Supported
Change of Variable Values	Unsupported
Change of Data Types	Unsupported
Redesign	Unsupported
Others	Partially Supported

`Size Increase` is a special case where the patch increases the buffer to avoid overflows.

The `Change of Data Types` is a unique type where the variable type is changed, for example, from *int* to *long int*. `Redesign` refers to the rewrite the function logic with a lot of different program changes. The `Others` patch type specifies some minor changes that cannot be put into the major categories.

We have collected 375 CVEs. Except for 2 cases, whose official patches cannot be found, we have summarized and categorized the 373 CVE patches into different groups based on the patch categorization schema. The allocation of different types of patches is presented in Table 4. According to the table, `Sanity Testing` the most commonly used patch pattern, which accounts for 42.1%. This kind of fix tries to read and check the value of the variable to make decisions. It meets the `Operation Rule 2`, which does not write memory contents. This type of patches are good candidates for generating hot patches.

3.2 Observations

We have obtained four interesting observations during the study of the vulnerability patches.

Observation 1: Vulnerability patch changes are generally small compared to other program updates. Most of the patches in the Android kernels are small in size with only a few lines of code changes. In the 373 CVEs, there are only 64 CVEs that either have more than 30 lines of modification or modify more than 5 functions in one patch. This observation is consistent to the work [40], which states that in Chrome and FireFox bug fixes, small patches account for the largest percentage amount all the security-related patches. This observation suggests that hot patch is a possible solution to fix a large number of vulnerabilities in Android since it favors small changes.

Observation 2: Large vulnerability patches often consist of several small individual patches. Moreover, for the larger vulnerability patches, they often consist of many small individual fixes. In the 64 large patches, there are 50 patches that are the combination of several small changes in different functions. For example, the patch for vulnerability `CVE-2016-8457` [8] is considered as big, since it has more than 50 lines of code changes. However, they can be divided into several small fixes in different places of the functions. The reason is that there is a vulnerability pattern, which appears multiple times inside the function. Therefore, the similar fixes need to be inserted into the function for every occurrence of the same patterns, which results in a large fix when aggregated. This observation indicates that we are able to analyze some of the large and complex vulnerability fixes using the divide and conquer approach. By appropriately dividing the large patch, one can get smaller patches, which can be converted into hot patches individually.

Observation 3: The patch pattern may be different regardless of the vulnerability types. After summarizing the different type of patches, we have compared them with the different type of vulnerabilities. We find that there is no evidence to show that the patch type and vulnerability type have strong co-relations. In general, the patches for same type vulnerability may be written in different ways; and the same type of code change can fix different types of vulnerabilities. Therefore, the patch type should be summarized differently from the vulnerability type, which shows that our way of patch classification is reasonable.

Observation 4: Some patches consist of both non-security upgrade and vulnerability patch. There are some patches, which have non-security upgrade apart from having vulnerability fixes. The reason for mixing the two kinds of patches in the same commit may be that the programmer does not want to disclose the vulnerability directly to the public. By mixing them with some function upgrades, it makes them hard to be detected by the attackers. For example, in the fix of `CVE-2016-8457`, there is a piece of the code does the normal function update jobs without fixing the vulnerability. This observation explained the reason why some of the patches are large patches with a mix of many types of code changes. In fact, the real security patch may be small, but when being added in some other updates, it becomes large and difficult to be analyzed.

3.3 VULMET Work Scope

Based on the vulnerability patch study and the VULMET operation scope in Section 2.3, we have defined the scope of the

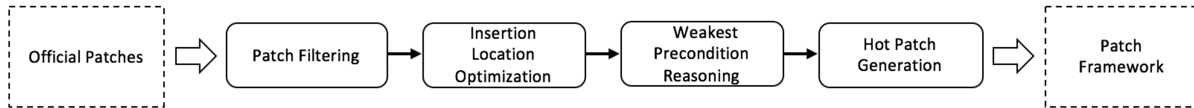

Figure 4: Framework Overview of VULMET

vulnerability types which VULMET is able to handle. Table 5 has shown the patch types, which can be supported by VULMET. First, VULMET will support the `Sanity Testing` since it only checks (reads) the value in the function and makes decisions, which satisfy all the Operation Rules in Section 2.3. Second, for the `Function Call` type, VULMET is able to go into the callee function and analyze the changes. If the changes do not involve the memory write operation, VULMET can support the patch. Thus, VULMET partially supports the `Function Call` type. Third, for the type of `Change of Variable Values` and `Change of Data Types`, since they both need to write the value to the memory which is against `Operation Rule 2`, they are not supported by VULMET. Forth, VULMET does not support the type `Redesign` since it greatly changes the original function semantics and violates `Operation Rule 3`. Last, in type `Others`, we have manually gone through each case. There are some cases, which do not contain writing operations to the memory. VULMET can generate hot patches from these cases. For example, in `CVE-2018-17182` [9], the patch removes the entire vulnerable function. In our patch categorization, it belongs to the `Others` type. VULMET is able to generate an equivalent semantic patch by skipping the function. The detailed discussion is shown in Section 5.1.1. However, there are also cases that involve the change of memory content. Therefore, VULMET can partially support this type.

4 Methodology

In this section, we present the detailed algorithms for automatic hot patch generation.

4.1 Overview

Fig. 4 shows the overview of VULMET. When a patch has been officially released, suitable patches will be selected for the hot patch generation. For a patch candidate, there are different locations inside the vulnerable function that the hot patch can be inserted. VULMET will choose the best location to insert the patch by calculating the side effect for each place. After that, it will leverage on the weakest precondition analysis to find the semantic equivalent constraints of the official patches. Those constraints will be converted into the hot patch, which can be applied to the binary programs.

4.2 Patch Filtering

The first step of VULMET is to determine whether an official patch can be converted into a hot patch. As stated in Section 2, the hot patch operation is limited to enforce the program security. Therefore, only the official patches, whose operation semantics comply with the requirements, can be used to generate the hot patch. To achieve it, VULMET will extract the official patch by diffing the vulnerable code and patched code. Then, for each statement in the patch, it will be classified as the normal operation and the prohibited operation. The prohibited operation includes the assignment of variable or pointer values and the call to memory modifying functions. If the official patch does not contain prohibited operations, VULMET will select it as a candidate to generate the hot patch. Otherwise, the patch is filtered out.

4.3 Insertion Location Optimization

4.3.1 Motivation and Problem Definition

According to Rule 1 at Section 2.3, hooking function at the beginning or the end is the requirement to ensure patch's practicality. Therefore, VULMET can only hook the target vulnerable function and the functions (i.e. callee function) which are called by the target function. Each of the hooking place is considered as a possible location to apply the hot patch. Among the several places inside the target function, VULMET is designed to find the best one. Some of patch points may not contain enough information on the variable values to calculate the semantic equivalent constraints. Some of them will have unexpected effects since the function may be executed until it reaches the patch point. To find the best point, those different aspects need to be taken into consideration.

To illustrate the problem, we reuse the example at Fig. 3 in Section 2.4. In this example, previously, we assume the patch point is at line 1. In fact, there are two more points that can also apply the hot patch. They are line 1 (the beginning of function `q6lsm_snd_model_buf_allo()`) and line 7 (the call to function `cal_utils_get_only_cal_block()`). Both of the two points will have enough information to calculate the relationship between the function parameters and the variables used in the official patch. Therefore, in either of the two points, VULMET can generate a semantic equivalent hot patch to fix the vulnerability.

However, patching the function at different locations will result in different side effects, which may harm the normal executions. In this case, if we insert the patch in the call

to function `cal_utils_get_only_cal_block()` at line 7 and the patch kills the execution, some instruction from line 1 to line 7 has already be executed (Note: at line 5 codes are omitted for simplicity). There may be some program changes such as memory allocation. However, if the function is killed in the middle, it may not finish the proper clean up process, such as freeing the allocated memory. This may introduce new program flaws and make the patched function unsafe. Instead, if the hot patch is applied at line 1 and kills the function, then the instructions with side effects will not be executed. Therefore, patching at line 1 is relatively safer than patching at line 7. VULMET is designed to select the best point among the candidates.

We define this problem as an insertion location optimization problem. The goal is to find an insertion point, which has adequate information to calculate the semantic equivalent constraints and has the least side effects on the program. The reason for choosing the point which incurs least side effects is that patching at this point will have the most similar semantics to the original patches. It is inevitable that, in some cases, the side effects will result in the function working differently than the original target function. In this case, VULMET chooses to sacrifice the normal functionalities to make sure the patch can block the vulnerabilities since the first priority is to protect the system. Therefore, by choosing the point with least side effects, VULMET tries to patch the vulnerabilities while keeping as many normal functionalities as possible.

4.3.2 Demonstration Example

The workflow of the algorithm is as the following. First, all the possible insertion points are listed. (In the running example of Fig. 3, the beginning of function `q6lsm_snd_model_buf_allo()` and the call to function `cal_utils_get_only_cal_block()`.) Since the hot patch works on the binary level, there may be inlined functions, which have been merged into their caller functions. Those inlined functions will not be considered as a proper insertion point. The detailed method to handle the inlined function will be given in Section 4.4.3. (After the compilation, the functions at line 1 and line 7 are not inlined in the resulted binary.)

Second, the algorithm will try to build two program paths. The first one (path I) starts from the function beginning and ends at the patch insertion point. The second path (path II) starts from the insertion point and ends the official patch location. To build the two paths, VULMET will remove the branches in the code and flatten the loops by unrolling them once. The resulted path is a sequential program slice. (In Fig. 3, the paths for insertion point at line 1 is path I: 1-1 and path II: 1-15 and the path for insert point at line 7 is path I: 1-7 and path II: 7-15.)

Third, to ensure, at the insertion point, there is enough information to build the semantic equivalent constraints, the algorithm will try to back-propagate the variables in the official patches through the path II. If all the variables can be traced back through the path, the insertion point will contain adequate information to build the hot patch. (As discussed in Section 2.4, the two insertion points have enough information.)

Fourth, the algorithm will check whether there is any side effect introduced if the patch is applied. If the patch insertion point is at the beginning of the vulnerable function, there will be no side effect generated. Otherwise, VULMET will examine the path I to obtain the statement which can lead to side effects. 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. The algorithm will choose the insertion point, whose path to the official patch has least side effects. (Since line 1 of the function in Fig. 3 is the beginning of the vulnerable function, patching at it has no side effects on the function. Line 1 will be selected as the optimal patch insertion point.)

Table 6: Relationship between the Semantics Calculation and the Weakest Precondition Reasoning

Semantics Calculation	**Precondition Reasoning**
Official Patch Semantics	Postconditions
Instructions and Statements	Predicate Transformers
↓	↓
Hot Patch Constraints	Weakest Preconditions

4.4 Weakest Precondition Reasoning

After selecting the patch insertion point, the next step is to produce the hot patch at that point by calculating the semantic equivalence of the official patch. In VULMET, this process is reformed into a weakest precondition reasoning task. In programming, a precondition is a statement that should be true before the function is called. While, a postcondition is a statement that will be true if the function finishes and all the preconditions are met. Table 6 demonstrates the relationship between the semantics calculation and the weakest precondition reasoning. Given an official patch, its semantics can be converted into one or more postconditions. The statements in the vulnerable functions will define the transformers in solving the weakest precondition. The process of getting the hot patch constraints is equivalent as calculating the weakest preconditions. The resulted weakest preconditions are the semantic equivalent hot patch of the official patch.

4.4.1 Determined Statement Transformation

To solve the weakest precondition problem, VULMET takes an input postcondition P and a statement s in the original

vulnerable function. It solves the condition via the calculation of the predicate transformers [17]. Then, it outputs the weakest precondition of s with respect to P, which is denoted by $wp(s,P)$. The rules of the calculations for the determined statement transformation are listed:

$$wp(skip, P) \Leftrightarrow P \quad (1)$$

$$wp(x := e, P) \Leftrightarrow P[x \longmapsto e] \quad (2)$$

$$wp(s_1 : s_2, P) \Leftrightarrow wp(s_1, wp(s_2, P)) \quad (3)$$

$$wp(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end}, P) \Leftrightarrow (b \wedge wp(s_1, P)) \vee (\neg b \wedge (s_2, P)) \quad (4)$$

`Rule 1:` When the statement has no effects on the postcondition P, the statement is skipped. The precondition is the same as the postcondition. `Rule 2:` When there is an assignment statement, the corresponding variable x inside the postcondition is transformed into e. The resulted precondition will be expressed in term of e. `Rule 3:` If the statements are sequential, the weakest precondition is calculated backward. The precondition of the second statement will be the postcondition for the first statement. `Rule 4:` If there is a branch statement, the precondition will be depending on the branch conditions. The branch conditions will be aggregated as part of the precondition.

The four rules will specify the determined statement transformation to get the weakest precondition. All the values in the transformation will be calculated precisely. Therefore, this process guarantees the equivalence between the post- and preconditions so that the generated hot patch will be semantically equal to the official patch.

4.4.2 Demonstration Example

The basic workflow of weakest precondition reasoning is demonstrated with a real-world example. For the simplicity, the demo is shown with C language, whereas the actual reasoning is based on LLVM. Fig 5 has shown the official patch for `CVE-2014-9873` [5]. The official patch tries to add a sanity check for variable `write_len` at line 11 and 12. To generate the hot patch, the patch semantic will be converted into a weakest precondition reasoning problem.

The postcondition P is `write_len <= 0`, the statements are the instructions from line 4 to 10. The output will be the precondition in term of the function input parameters, which is the same as the hot patch semantic. The problem is solved with the determined statement transformation. First, by `Rule 3`, the algorithm works backward. Therefore, the algorithm will start at line 10. Second, by `Rule 2`, the value of `write_len` is replaced by the equation on the right-hand side in line 10. The resulted precondition is `(int)(*(uint16_t *)(buf+2)) - cmd_code_len <= 0`. Third, by `Rule 3`, in line 8 and 9, the branch condition will be aggregated into the precondition to determine the value of the variable `cmd_code_len`. Forth, again by `Rule 2` at line 7, the value

```
1  void extract_dci_pkt_rsp(struct diag_smd_info
2      *smd_info, unsigned char *buf)
3  {
4      int cmd_code_len = 1;
5      int write_len = NULL;
6      uint8_t recv_pkt_cmd_code = 0;
7      recv_pkt_cmd_code = *(uint8_t *)(buf+4);
8      if (recv_pkt_cmd_code != DCI_PKT_RSP_CODE)
9          cmd_code_len = 4;
10     write_len = (int)(*(uint16_t *)(buf+2)) - cmd_code_len;
11 +   if (write_len <= 0)
12 +       return;
13     ...
14 }
```

Figure 5: Example: CVE-2014-9873

Table 7: Variable Reasoning

Post-condition	Precondition
write_len >=0	*(buf + 2) - cmd_code_len >= 0 (1)
(1)	*(buf + 2) - 4 >= 0 and recv_pkt_cmd_code != DCI_PKT_RSP_CODE *(buf + 2) - cmd_code_len >= 0 and recv_pkt_cmd_code == DCI_PKT_RSP_CODE (2)
(2)	*(buf + 2) - 4 >= 0 and *(buf + 4) != DCI_PKT_RSP_CODE buf + 2 - cmd_code_len >= 0 and *(buf + 4) == DCI_PKT_RSP_CODE (3)
(3)	*(buf + 2) - 4 >= 0 and *(buf + 4) != DCI_PKT_RSP_CODE *(buf + 2) - 1 >= 0 and *(buf + 4) == DCI_PKT_RSP_CODE (4)

of variable `recv_pkt_cmd_code` is changed into the value of `buf`. Line 4 to 6 only contain the assignment statements with constant values at the right-hand side. There, the postcondition will be transformed into precondition by replacing the variable values with their corresponding constants.

Table 7 has summarized the steps of the transformation from postcondition to the precondition. The original semantics will be changed into the precondition by the transformer rules. The final precondition `Equation (4)` in Table 7 will be the hot patch semantics.

4.4.3 Function Calls

For the non-determined statements, such as function calls and loops, VULMET uses algorithms to summarize the semantics. The detail explanation for handling the function calls and the loops will be given in the following sections.

Handling function call is a major task in program analysis. In this work, by `Operation Rule 1` in Sec. 2.3, a function call can be regarded as a hooking point, whose input parameters and return value can be obtained. Therefore, VULMET will use function calls to extract variable values for the hot

patch generation. However, there are some cases where the functions are not suitable to be used as the hooking points. Therefore, VULMET need to handle those cases to generate accurate patches.

Inlined function The first case is where the function is inlined during the compilation process. The inlining process will merge the binary instructions of the function into its caller's instructions. The start of the inlined function will be in the middle of another function. Therefore, it is difficult to find a precise location to hook those functions. VULMET handle the inlined function in a different way.

Before the function analysis start, VULMET will perform a check to figure out the inlined function in the target program. Then, it will import the contents of the inlined functions into their caller functions. The framework will treat the inlined function as a part of the target function's code when analyzing it. In general, the inlined function has two attributes. First, it only contains a small piece of code to perform simple tasks. Second, it hardly ever calls other functions. The two attributes make the function easy to be inlined. Also, they allow VULMET to import the code to do the analysis.

Value modification function There are function calls in the middle of the original function. The callee function may modify the values, which are used in the calculation of the weakest precondition. In order to have an accurate result, VULMET needs to analyze the callee functions to understand how the values are changed inside them. After that, VULMET can use the modification as the determined statement transformation to calculate the precondition. VULMET uses SVF [49,50], a tool that provides inter-function analysis to determine whether a particular variable has been changed inside the function. VULMET will skip all the irrelevant functions without any value changes. Next, for the functions with value changes, VULMET will go inside the callee function and calculate the changes made by the function. The changed semantics are summarized and used to represent the functions. After that, VULMET will start to perform the weakest precondition reasoning to get the hot patches.

Algorithm 1 describes the workflow for the function handling process. The functions in the algorithm refer to the callee functions inside the target vulnerable function. First, all the functions on the analysis path will be input into the algorithm. Next, VULMET will try to look up the function label in binary to check whether it has been inlined. If it is inline, VULMET will import the function into its caller for analysis. If the function is not inlined, it will continue to determine whether the point is the ideal insertion point. If the point is selected as the insertion point, it will extract the function input variable information and continue to weakest precondition solving. If the function is not the insertion point, it will check whether the function modifies the variable with the help of SVF. If the function modifies the relevant variable, VULMET needs to go deep into the function and performs further analysis to summarize the changes. If the function modifies an irrelevant variable or does not modify any variable, VULMET will skip it. If the callee function calls another function, which results in nested function calls, VULMET will treat the function as complex and skip the analysis.

Algorithm 1 Function Handling

```
1:  function HANDLE_FUNC(func f)
2:      Lookup f in binary
3:      if f's name is found (not inlined) then
4:          Check f for insertion point
5:          if f is insertion point candidate then
6:              Add f to insertion point analysis process
7:          else
8:              Check whether f modifies relevant variables
9:              if f modifies relevant variables then
10:                 Analyze the code in f
11:             else if f dose not modify then
12:                 Skip f and return
13:             else if f is too complex then
14:                 Skip f with red flag
15:             end if
16:         end if
17:     else
18:         Import the source code of the inlined function f
19:     end if
20: end function
```

4.4.4 Loops

Loops are another major problem in program MODanalysis. Since in static analyze, it is difficult to determine the exact number of iterations that the loop will be executed and the exact output values. Some works, such as [53], propose loop summarization algorithms, which could yield approximation results for some types of loops. However, since hot patches need to be precise to completely fix the vulnerabilities, the approximation in loops may greatly affect the accuracy of the patches.

Since loops are in different types, VULMET develops different strategies to handle different loops. The first type of loop is the one that contains the official patch. In this type, the patch semantics are repeated several times according to the loop iterations. VULMET will extract the loop iteration conditions and perform the weakest precondition solving on them. Then, it will construct a semantic equivalent loop at the insertion point. The hot patch semantics will be included inside the constructed loop. The second type of loop is the one that appears in the middle of the analysis path. To handle this type of loop, VULMET needs to first determine whether the loop modifies any of the relevant value used for weakest precondition solving. If no relevant value is changed, the loop can be skipped. Otherwise, VULMET leverages the idea of [53] to perform the loop summarization. It will generate the ranges

of the values which have been changed inside the loop. Then, VULMET takes the conservative way to choose the largest range of the value to form the hot patch semantics so that the generated patch can fix the vulnerability with the possibility of affecting the normal functionalities. Last, if the loop is too complex with new function calls or multi-level nested loops inside, we choose to skip the loop without any analysis.

4.5 Binary Hot Patch Generation

The last step is to generate the hot patch based on the precondition constraints. VULMET uses an empty function as the template and set the function to have the same number and type input parameter as the original target function. Then it inserts all the constraints to it and compiles the function into binary executables which can be hot patched to the kernels.

The major challenge is to determine the actual address of the variables used in the patches. Since VULMET hooks the function at the beginning or end, the address of the input parameters and the return value can be determined. For the address of variables inside structures, VULMET will look up the relative address from the source code. The relative address will be added to the base address, which is obtained from hooking, to give the exact address of the variables.

VULMET supports the hot patch for real-world Android platform with architectures ARM 32 bits and 64 bits. To suit for various architectures, VULMET is designed to output the weakest preconditions of the patches. These can be used to generate the binary instruction of different architectures to support different platforms.

The generated hot patch includes a binary executable with the patching logic and a file to record the hooking point(s). To apply the hot patch, one can use the standard hot patching procedure to load the hot patch into the memory and build a trampoline at the hooking point to direct the control flow of the program to the loaded patch. After the execution of the patch, it will either pass the control back to the function or return the function to prevent the vulnerability.

5 Evaluation

We have evaluated VULMET for the correctness, robustness, and efficiency of its generated hot patches. Correctness quantifies the patches' ability to fix the vulnerability, robustness quantifies the patches' ability to maintain the stableness of the program, and efficiency quantifies how much overhead the patches introduce. We have designed experiments to test the effectiveness of the patches in the three aspects. In the experiments, all the patches are tested on the Android Open Source Project (AOSP) platform Google Nexus 5X with Android kernel version 7.1.1 r31 bullhead build.

Table 8: Prevention of CVE exploit attacks

CVE NO.	Before Patch	After Patch
CVE-2014-3153	System crash	Safe
CVE-2016-4470	System crash	Safe
CVE-2014-4943	System crash	Safe
CVE-2018-17182	System crash	Not exploited

5.1 Correctness Evaluation

In this section, we evaluate the correctness of the generated hot patches. The experiment consists of three parts. First, we test the patches with real-world CVE exploits. Second, for the vulnerabilities whose exploit is not available, we manually verify the correctness of the patches. Third, we manually write hot patches and compare the generated hot patches against them to check whether the generated patches fix the vulnerable in the same way as human experts.

5.1.1 Experiment 1: Patches against Exploits

We assess the correctness of the generated hot patch against real-world exploits. We manually collect exploits for the Android CVEs and use them to attack the system patched by VULMET. To the best of our knowledge, we have found 3 working exploits for the vulnerabilities with the hot patches. In addition, we have also tested the hot patch for the recent critical vulnerability, `CVE-2018-17182`. Table 8 lists the four exploits and shows the program running results before and after the application of the hot patches. The result suggests that all the patches have successfully prevented the attacks from the exploits. For `CVE-2014-3153`, `CVE-2016-4470`, and `CVE-2014-4943`, the hot patches have fixed the vulnerability completely. For `CVE-2018-17182`, the hot patch can successfully prevent the exploit but cannot stop the system from crashing. It is because that the patch can only partially fix the vulnerability. In the following, we discuss the patch correctness in detail with code examples.

CVE-2014-3153 is a privilege escalation vulnerability in function `futex_requeue()` function. As shown in Fig. 6, the official patch fixes the vulnerability in three different location of the functions.

For the first patch in Fig. 6(a), VULMET extracts the semantics of checking the equivalent of variable `uaddr1` and `uaddr1` at line 2. Then, it converts the semantics into the hot patch at the beginning of the function `futex_requeue()`. The two variables used in the official patch are also the function input parameters. VULMET checks the analysis path to ensure there are no changes on the two variables. Therefore, the semantic will remain the same as the official patch. In addition, since the official patch is inside another sanity check (shown in (a) at Line 1), VULMET will also keep the semantics

```
1  if (requeue_pi) {
2  +    if (uaddr1 == uaddr2)
3  +        return -EINVAL;
4  ...
5  }
```

(a)

```
1  +if (requeue_pi
2  +        && match_futex(&key1, &key2)) {
3  +    ret = -EINVAL;
4  +    goto out_put_keys;
5  +}
```

(b)

```
1  +if (match_futex(&q.key, &key2)) {
2  +    ret = -EINVAL;
3  +        goto out_put_keys;
4  +}
```

(c)

Figure 6: Official Patch: CVE-2014-3153

when constructing the hot patch to keep as much original semantics as possible. The generated semantic is as follow:

```
hook futex_requeue
check requeue_pi:
if not 0:
    check uaddr1, uaddr2:
    if uaddr1 == uaddr2:
        return the function
```

For the second patch in Fig. 6(b), there is a function call inside the official patch, which has been inlined at the compiled binary. VULMET imports the code for the inlined function and extracts its semantics. Then the semantic is combined with the original official patch semantic, which is listed below.

```
check:
requeue_pi && key1 && key2
  && key1->both.word == key2->both.word
  && key1->both.ptr == key2->both.ptr
  && key1->both.offset == key2->both.offset
```

Then, VULMET tries to solve the conditions of all the variables appear in the semantics. `requeue_pi` is one of the input parameters so that its semantics remain the same. For the union pointers `key1` and `key2`, VULMET looks for a good patch insertion points, where the value of `key1` and `key2` is same as the value in the official patches. After the analysis, VULMET finds a non-inlined function call `hash_futex()` after the sanity checks. At that point, the value of `key1` and `key2`

```
1  static int pppol2tp_setsockopt(.....)
2  {
3  ...
4      if (level != SOL_PPPOL2TP)
5  -        return udp_prot.setsockopt(sk,
6  -            level, optname, optval, optlen);
7  +        return -EINVAL;
8  ...}
```

Figure 7: Official Patch: CVE-2014-4943

can be extracted. Thus, the tool will generate the hot patch by creating the patch at the point to get the value of `key1` and `key2` and checks them to make decisions.

For the third patch in Fig. 6(c), VULMET follows the same steps as the second patch, since the semantics of both of their official patches are the same.

CVE-2014-4943 is another function that has a known exploit [4]. It is a privilege escalation vulnerability located at function `pppol2tp_setsockopt()` and `pppol2tp_getsockopt()`. Therefore, the official patch fixes the vulnerability in two different functions. However, both of fixes follow the same way to fix the vulnerabilities. VULMET will generate the hot patch for each of the individual fix in the same steps.

Fig. 7 shows the official patch of `CVE-2014-4943` for function `pppol2tp_setsockopt()`. It tries to replace the value of the return statement. Instead of calling a function, the new return statement just returns a constant value. To generate a hot patch, VULMET will first look at the sanity check that contains the return statement. It builds a similar check statement at the beginning of the function to check the value of the variable since it is a function input parameter, whose value can be obtained via hooking. After that, if the condition is met, VULMET just returns the function. It will produce a hot patch with the same semantics as the original patch. The hot patch generation for the function `pppol2tp_getsockopt()` follows the same steps.

CVE-2016-4470 is a denial of service bug inside `key_reject_and_link()` in Linux kernel [7]. As in Fig. 8, the official fix adds in a sanity check to test the value of the variable `link_ret` at line 9 as shown in Fig. 8. The value is an indicator of whether the function `__key_link_begin()` is successfully executed. If it fails to run, the variable `edit` will not be initialized and the bug will be triggered. VULMET generates the hot patch by first selecting a good insertion point. After analyzing different possible places, VULMET has chosen to hook where the function call `__key_link_begin()` at line 6 has finished. It checks the return value of the function. If it is 0 (error), it will return the caller function to avoid further execution. Although there are some instructions between the insertion point and the official patching point (Line 8 has

```
int key_reject_and_link(.....){
...
    if (keyring) {
        if (keyring->restrict_link)
            return -EPERM;
        link_ret = __key_link_begin(keyring,
            &key->index_key, &edit);}
    ...
+   if (keyring && link_ret == 0)
        __key_link_end(keyring,
            &key->index_key, edit);
    ...
}
```

Figure 8: Official Patch: CVE-2016-4470

omitted some instructions), the program analysis results suggest they will not affect the value of the variable `link_ret`. Therefore, the hot patch provides the same semantics as the official patch to fix the vulnerability.

CVE-2018-17182 is a cache invalidation bug in the Linux kernel [9] [20]. The logic of the error handling function `vmacache_flush_all()` inside the kernel is incorrect, which results in potential exploit even when a strong sandbox is present.

The official patch fixes the vulnerability in two parts. First, it changes the sequence number from 32 bit to 64 bit, so that it avoids the overflow bug to trigger the error handling function. Second, it removes the buggy error handling function. There are two different semantics in the official patch. For changing the bit of the sequence number, VULMET is not able to generate an equivalent semantic of it, since modifying the memory contents is prohibited by the security requirements. However, VULMET can fix the second part since removing a function has an equivalent semantic as returning the function at the beginning. Thus, VULMET can generate a patch for part of the official patch. After applying the hot patch to the function, attackers can still trigger the overflow bug which may crash the program, but they are not able to exploit further to get the dangling pointer at the error handling function. The program is protected since the program will stop before the vulnerability is reached. The hot patch has partially fixed the vulnerability with a possible crash due to the remaining overflow bug. The fix semantic is listed below.

```
hook function vmacache_flush_all()
kill the function once called
```

5.1.2 Experiment 2: Manual Verification

Since the exploits are not always available for every CVE, it is difficult to conduct experiments on every patch against real-world attacks. Therefore, for the patches without exploits, we manually audit them to check whether the generated patches have fixed the patch or not. In total, VULMET has generated hot patches for 59 different CVEs. Excluding the 4 CVEs, which have known exploits, there are 55 to be manually verified. We believe that 59 vulnerabilities are sufficient to test the performance of VULMET since we are working on the real-world Linux kernel vulnerabilities. Table 9 has given the overall results for the manual verification.

Table 9: Manual Analysis on Patch Correctness

	Correct Patch	Incorrect Patch
Number	55	4

The results have suggested that VULMET has successfully generated correct patches for 55 out of 59 vulnerabilities. We have examined the four failed cases to understand the error made by VULMET. There are three patches which are considered as incorrect because the patches contain part of operations that need to modify the memory. Since the majority parts of these patches are sanity checks, when selecting the patch generation candidates, VULMET regards them as good ones. During the analysis, it will neglect the minor memory writing operations. However, the memory writing operations in the patches are the keys to fix the vulnerabilities. Therefore, VULMET will have difficulties to generate correct patches. In order to fix this issue, VULMET needs to enhanced its semantic analysis to detect the memory writing operation.

Another failed case is the one discussed in the previous section, `CVE-2018-17182`. In this case, only part of the semantics can be converted to the hot patch. Therefore, VULMET only gives an incomplete patch which can only prevent the exploits but not fixing the problems. From the failed cases, we know that to have a precise semantics of the original patches is one of the keys for generating the correct hot patches.

5.1.3 Experiment 3: Comparison with Human Written Patches

In this section, we would like to compare the generated patches with the human written ones. We manage to hire security researchers to understand the official patches and manually write hot patches for comparison. We have compared all the 55 correctly generated hot patches against the human written ones. Table 10 has summarized the comparison results between the human-written patches and the auto-generated patches. In addition, since human audition may be biased, we have also listed all the hot patch semantics online at [11].

The results show that most of the generated patches work in the same way as the human written ones. This is because both of the VULMET and the human follow the same way of understanding the semantics of the official patches. In the following, we will discuss the similarities and differences

Table 10: Comparison with Human Written Patches

	Similar Patch	Dissimilar Patch
Total number	54	1
CVE examples	CVE-2014-3145	CVE-2016-4470

between generated patches and human written patches.

Similar Patch: CVE-2014-4656 It is an integer overflows vulnerability in `snd_ctl_add()` function. As listed below, the official fix tries to check the input parameter `kcontrol`'s id index to see whether it is larger than the MAX value minus the `kcontrol`'s count. The official patch has put the fix at the beginning of the function. Since the generated patch also aims to fix the problem at the same point, there is no need for VULMET to do further semantic transformations. The generated patch is similar to the official one and so is the human written hot patch.

```
if (id.index > UINT_MAX - kcontrol->count)
```

Dissimilar Patch: CVE-2016-4470 The patch is discussed in the previous section with Fig. 8. For this case, the human-written patch is different from the generated one. The human-written patch tries to hook at the callee function after the official sanity check. It checks the variable value of `edit`. This value is an indicator of whether the function `__key_link_begin()` has been successfully executed. If the variable `edit` is found uninitialized, the function will be killed since `link_ret` will not be properly assigned.

These differences are introduced because the experts can understand the root cause of the vulnerability and apply the patch to fix the problem directly. Whereas, VULMET depends on the semantics of the official patches and follows a backward analysis path to transform them to hot patch semantics. However, both of the two patches can fix the vulnerability. Therefore, although with a slight difference in semantics, the generated hot patch can successfully patch the vulnerability as the human expert.

5.2 Robustness Evaluation

Since the hot patches modify the original programs, they may break other functionalities, which may lead to unexpected system faults. Therefore, it is important to ensure that the system robustness is not affected after applying the patches. In this experiment, we evaluate the robustness of the patched programs by testing patched kernels with Android benchmarks.

To build the testing environment, we choose the Android bullhead to build with Linux kernel version 3.10 and roll back commits to producing a kernel with many unpatched vulnerabilities. In this particular kernel, VULMET manages to convert 21 vulnerability patches into hot patches. Then we apply these patches to the kernel and run the AnTuTu benchmark [2] and the CF-bench [3] on the patched program to monitor any of the abnormal behaviors, such as crashes and hangs. Table 11 has summarized the results for the experiment. For demonstration purpose, we select 5 CVEs as the example and list the final results with all the 21 patches.

Table 11: Patch Robustness Analysis

CVE NO.	Kernel Ver.	Build	State
CVE-2014-4656	3.10	bullhead	robust
CVE-2015-7515	3.10	bullhead	robust
CVE-2015-8543	3.10	bullhead	robust
CVE-2016-2468	3.10	bullhead	robust
CVE-2016-8399	3.10	bullhead	robust
Overall	-	-	21/21 robust

The results show that all the hot patches do not crash or hang the program. To further examine the patch robustness in the real-world situation, we have selected and installed top 100 Android applications from the Google App Store. We use scripts to open, load, and close the application on the patched system and monitor abnormal behaviors. The result shows that all the application can be properly executed, which suggests that the patches maintain good robustness in the real-world situation. In conclusion, the generated patch does not break the normal functionalities of the patched program.

5.3 Efficiency Evaluation

Since the hot patches inject code into the original functions, it is important to ensure that the additional code does not add much overhead to the programs. A less efficient hot patch may introduce performance bug to the system, which affects its normal usage. In this experiment, we evaluate the efficiency of the hot patches by measuring the overhead of the program after patching.

We test the system performance before and after the patching with AnTuTu benchmark on Google Nexus 5X device. We control the experiment settings to be the same to test one hot patch a time. Each of the experiment is repeated 10 times and the scores are averaged to avoid variations due to noises. Table 12 lists the performance of the kernels with 5 individual CVE patches as well as the overall performance with all the 21 patches applied.

Overall, the results suggest that the hot patches do not introduce noticeable overhead system-wise. For the CPU running time benchmark (3rd column), the patched kernel does not have significant differences with the original one. For example, the kernel with all the patches applied only adds 0.06s for the total running, which is less than 0.1% in overhead. For the memory running time benchmark (5th column), the overall run time for the patched system is even shorter than the original one. For the score benchmarks (2nd and 4th columns),

Table 12: Patch Overhead Analysis

CVE id	CPU Score	CPU Time	Mem Score	Mem Time
Original Kernel	20620.0	1:22.30	4428.3	1:24.52
CVE-2014-4656	20597.9	1:22.78	4576.7	1:23.37
CVE-2014-9789	20525.5	1:22.51	4398.1	1:25.12
CVE-2015-7515	20731.0	1:22.34	4548.3	1:23.88
CVE-2016-8399	20455.7	1:22.36	4368.8	1:24.66
CVE-2016-10233	20715.5	1:22.31	4542.6	1:24.32
Overall	20587.2	1:22.36	4506.1	1:23.98

all the results are within the reasonable ranges, which are either slightly higher or lower compared to the original kernel results. Therefore, the patches make low overhead on the system.

5.4 Threat to Validity and Future Works

In this section, we discuss the limitations of VULMET and propose potential future works to improve it. First, the assumption has been made that the hot patch cannot modify the memory content of the original program. Though it guarantees the stableness of the patched program, it also limits the workable type of the generated hot patches. There is a large percentage of vulnerabilities which cannot be fixed by VULMET using the existing hot patches. In the future, we would like to develop algorithms to analyze the semantics of the memory contents and propose safe memory modification operations. The major challenge is two folds. First, the function stack information needs to be kept after applying the patch changes. Since we are not creating the new function stacks, we need to make sure the newly added patches do not overflow the old stacks. Second, VULMET needs to be able to insert the changes in the middle of the functions. The write operation is different from the read operation. At the binary level, a memory write operation is often followed by some read operations, which have data dependency on the previous write operation. Therefore, it is better to change the value at the same place as the original patch. Thus, to locate the binary instruction in the middle of the function is important to implement the write operation in VULMET. After identifying the patches whose write modification is safe, VULMET can generate the hot patch to cover more vulnerabilities.

Second, VULMET relies on the precise summarization of the official patch semantics to generate correct hot patches. In the experiments, some generated hot patches are incomplete because the semantics are not fully extracted by VULMET. It needs to have formal semantic analysis capability to define the changes made by the original patches. With this, VULMET will have less chance to miss out the important semantics of the official patches so that the overall accuracy will be improved.

Third, there are some patches being too complex to be analyzed. It is difficult to find the precise semantics of the large patches. Therefore, current VULMET only works on patches with changes in one function. In the future, we plan to introduce root cause analysis to help to identify the main changes that can patch the vulnerabilities. VULMET can generate the hot patches based only on the main changes so that it does not need to recover the full semantics for the complex patches.

6 Related Works

6.1 Automatic Patch Generation

Automatic patch generation is a hot topic in security researches [37]. Many different approaches have been proposed to address this problem. The first approach attempts to summarize patch patterns and use them to generate new patches to fix similar vulnerabilities. For example, in 2005, [45] has proposed automatic patch generation algorithms for the buffer overflow vulnerabilities. By monitoring the program operations in a sandboxed environment during attacks, it generates patches that can work at the same environment. [23] proposes PAR, which generates security patches by learning from the human-written patches. They manually examine the human-written patches and develop the patch template. Then, they locate the faults by running the test case and apply corresponding templates to fix the bugs. [33] mines a large number of human fixes and applies mathematical reasoning model to search for templates to fix the bugs. [32] also summarizes patch templates from the human patches and apply them to fix Java vulnerabilities. Instead of writing the templates manually, the work uses the clustering method to categorize different patch patterns and summarize the pattern for each of the categories. DeepFix [19] learns the patch patterns using deep learning with multi-layered sequence-to-sequence neural network and fix vulnerability with the patterns.

The second approach tries to generate patches by testing different patch candidates with the testcases. The patch that can pass the test will be selected. Shieldgen [16] generates the patch for the unknown vulnerabilities via analyzing the zero-day attack instances. [24, 52] propose and improve GenProg, which automatically searches for patches using a genetic programming algorithm to evolve the variant to find the correct patches. They use mutation and crossover operators to change the original program and simulate the program evolution. During this evolution, different patch behaviors can be executed so that the best one can be selected to fix the bug. [44] also leverages on program evolution to automatically search for patches in the assembly code programs. They demonstrate that the patch generation at the binary level is as efficient

as at the source code level. [28–30, 42, 46] propose tools to generate patches and conduct an analysis of the effectiveness of the generation process. They define the operations that the patch can perform on the program and generate possible patch operations. They use heuristics and program analysis methods to rank the possible patch operations based on their possibility to fix the vulnerability. Then, they try different patches against the test cases to get the one which allows the test cases to pass. AutoPaG [26] also tries to generate patches for the out-of-bound read vulnerabilities in the Linux kernel. It can catch the violations and summarize the root causes during the runtime. The patch is then built to address these problems.

The third approach aims to analyze the cause of the vulnerability and build the patches to prevent that. Minthint [21] generates hints to help the programmers repair the bugs. Statistical correlation is used to find statements that are possible to appear at the patch location. SIFT [31] uses static program analysis to generate input filter for the integer overflow programs. [54] has proposed AppSealer, a tool which can automatically generate patches for known component hijacking vulnerabilities in Android applications. It uses the program analysis to identify the program slice which leads the vulnerable places and builds patches to block malicious program flows. [43] tries to generate filters for the web server to prevent malicious inputs. It helps the developer by automating the error-prone filter writing process. [27] studies real-world concurrency bugs and generates patches via analyzing the program flows. SearchRepair [22] has combined all three approaches. It generates Satisfiability Modulo Theories (SMT) constraints for defects, uses program analysis to locate bugs, and searches patches using test suits. [39] also uses SMT to solve the constraints to generate patches for buffer overflow bugs. [35] combines program analysis with data mining to generate patches with a low false positive rate. Directfix [34] tries to generate simplest source code patches using a semantics-based repair method so that the patches can be accepted by the developers.

Unlike these related works, our work has proposed a new approach by learning semantics from the official patches, which does not require the test cases. Since the generated patches have the same semantics as the official patches, they can fix the real vulnerabilities rather than merely pass the tests.

6.2 Hot Patching Framework

ClearView [41] is an automatic error patching framework at the binary level. It builds models for the normal execution of the program and detects abnormal executions, which are considered as errors. Once errors occur, it looks for the invariants and generates patches based on them. ClearView will perform the self-evaluation to determine whether the patches fix the errors. Bouncer [15] adopts attack detector DFI [12] to identify vulnerability exploits. Then it leverages both static and dynamic symbolic execution to generate the filters to drop the bad input before passing to the vulnerable program. Embroidery [55] is a hot-patching framework for outdated Android systems. It uses both static and dynamic analysis to build a binary rewriting engine to patch the vulnerabilities. Instaguard [13] is a hot-patching framework for Android aims to fix the vulnerabilities without adding code to the original programs. Instead, it uses the patch specification to generate rules to mitigate the vulnerabilities. Our work is complimentary for those works as our output hot patches can serve as the inputs for their patching frameworks.

7 Conclusions

In this work, we have defined the automatic hot patch generation problem. We studied the patch behaviors of the recent real-world Android vulnerabilities and proposed approaches to automatically generate hot patches, which can be applied directly to the Android kernels without affecting the user experiences. To demonstrate the capability of the approach, we have developed a tool, named VULMET, which can generate the semantic equivalent code changes by learning from the semantics of the official vulnerability patches via program analysis. The experiments demonstrated VULMET's capability to generate correct hot patches for fixing the real-world CVEs. The generated hot patches were tested to show that they can maintain the robustness of the program while keeping a very low overhead.

Acknowledgement

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001) administered by the National Cybersecurity R&D Directorate, and Alibaba-NTU JRI project (M4062640.J4A).

References

[1] Android security bulletin. https://source.android.com/security/bulletin.

[2] Antutu benchmark. http://www.antutu.com/en/.

[3] Cf-bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench&hl=en_SG.

[4] Cve-2014-4943 patch. https://git.kernel.org/pub/scm/linux/kernel/

git/torvalds/linux.git/commit/?id=3cf521f7dc87c031617fd47e4b7aa2593c2f3daf.

[5] Cve-2014-9873 patch. https://source.codeaurora.org/quic/la/kernel/msm/commit/?id=ef29ae1d40536fef7fb95e4d5bb5b6b57bdf9420.

[6] Cve-2015-8940 patch. https://source.codeaurora.org/quic/la/kernel/msm-3.10/commit/?id=e13ebd727d161db7003be6756e61283dce85fa3b.

[7] Cve-2016-4470 patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=38327424b40bcebe2de92d07312c89360ac9229a.

[8] Cve-2016-8457 patch. https://github.com/aosp-mirror/kernel_msm/commit/e5c1b001a822e8b38680655c400e7b3f67cc3323.

[9] Cve-2018-17182 patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7a9cdebdcc17e426fb5287e4a82db1dfe86339b2.

[10] Ida pro. https://www.hex-rays.com/products/ida/.

[11] List for hot patch semantics. https://sites.google.com/view/usenix-auto-patch-paper.

[12] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 147–160.

[13] Chen, Y., Li, Y., Lu, L., Lin, Y.-H., Vijayakumar, H., Wang, Z., and Ou, X. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)* (2018).

[14] Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., and Wei, T. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)* (2017).

[15] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing software by blocking bad input. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 117–130.

[16] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on* (2007), IEEE, pp. 252–266.

[17] Dijkstra, E. W., and Scholten, C. S. *Predicate calculus and program semantics*. Springer Science & Business Media, 2012.

[18] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., and Rajarajan, M. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials 17*, 2 (2015), 998–1022.

[19] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. In *AAAI* (2017), pp. 1345–1351.

[20] Horn, J. A cache invalidation bug in linux memory management.

[21] Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 266–276.

[22] Ke, Y., Stolee, K. T., Le Goues, C., and Brun, Y. Repairing programs with semantic code search (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (2015), IEEE, pp. 295–306.

[23] Kim, D., Nam, J., Song, J., and Kim, S. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 802–811.

[24] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (2012), IEEE, pp. 3–13.

[25] Li, F., and Paxson, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 2201–2215.

[26] Lin, Z., Jiang, X., Xu, D., Mao, B., and Xie, L. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), ACM, pp. 329–340.

[27] Liu, H., Chen, Y., and Lu, S. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering* (2016), ACM, pp. 715–726.

[28] Long, F., and Rinard, M. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, pp. 166–178.

[29] Long, F., and Rinard, M. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (2016), IEEE, pp. 702–713.

[30] Long, F., and Rinard, M. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices 51*, 1 (2016), 298–312.

[31] Long, F., Sidiroglou-Douskos, S., Kim, D., and Rinard, M. Sound input filter generation for integer overflow errors. *Acm sigplan notices 49*, 1 (2014), 439–452.

[32] Ma, S., Thung, F., Lo, D., Sun, C., and Deng, R. H. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security* (2017), Springer, pp. 229–246.

[33] Martinez, M., and Monperrus, M. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering 20*, 1 (2015), 176–205.

[34] Mechtaev, S., Yi, J., and Roychoudhury, A. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 448–458.

[35] Medeiros, I., Neves, N. F., and Correia, M. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web* (2014), ACM, pp. 63–74.

[36] Mokhov, S. A., Laverdiere, M.-A., and Benredjem, D. Taxonomy of linux kernel vulnerability solutions. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*. Springer, 2008, pp. 485–493.

[37] Monperrus, M. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR) 51*, 1 (2018), 17.

[38] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013), ACM, pp. 259–268.

[39] Muntean, P., Kommanapalli, V., Ibing, A., and Eckert, C. Automated generation of buffer overflow quick fixes using symbolic execution and smt. In *International Conference on Computer Safety, Reliability, and Security* (2014), Springer, pp. 441–456.

[40] Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., and Rajan, H. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (2013), IEEE Press, pp. 180–190.

[41] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 87–102.

[42] Qi, Z., Long, F., Achour, S., and Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ACM, pp. 24–36.

[43] Razmov, V., and Simon, D. R. Practical automated filter generation to explicitly enforce implicit input assumptions. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual* (2001), IEEE, pp. 347–357.

[44] Schulte, E., Forrest, S., and Weimer, W. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010), ACM, pp. 313–316.

[45] Sidiroglou, S., and Keromytis, A. D. Countering network worms through automatic patch generation. *IEEE Security & Privacy 3*, 6 (2005), 41–49.

[46] Sidiroglou-Douskos, S., Lahtinen, E., Long, F., and Rinard, M. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 43–54.

[47] Sotirov, A. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference, Las Vegas, Nevada* (2006).

[48] Soto, M., Thung, F., Wong, C.-P., Le Goues, C., and Lo, D. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), ACM, pp. 512–515.

[49] Sui, Y., and Xue, J. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction* (2016), ACM, pp. 265–266.

[50] Sui, Y., Ye, D., and Xue, J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering 40*, 2 (2014), 107–122.

[51] Tian, Y., Lawall, J., and Lo, D. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering* (2012), IEEE Press, pp. 386–396.

[52] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 364–374.

[53] Xie, X., Chen, B., Zou, L., Lin, S.-W., Liu, Y., and Li, X. Loopster: static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 84–94.

[54] Zhang, M., and Yin, H. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS* (2014).

[55] Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., and Gu, D. Embroidery: Patching vulnerable binary code of fragmentized android devices. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on* (2017), IEEE, pp. 47–57.

[56] Zhong, H., and Su, Z. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 913–923.