NANYANG TECHNOLOGICAL UNIVERSITY

FINAL YEAR PROJECT - FINAL REPORT

Android Security: On-the-fly security policies update

Project ID: SCE14-0015

Supervisor: Asst Prof Alwen Fernanto Tiu

Student Name: Nguyen Thanh Nam

Matriculation Number: U1120222J

School of Computer Engineering

Nanyang Technological University - Bachelor of Computer Engineering

23 – 03 – 2015

Abstract

LogicDroid is a customized Android operating system which contains a security extension based on metric linear-time temporal logic (MTL) to capture privilege escalation attacks. By adding various hooks in Android OS, the call chains among applications and processes can be tracked by the monitor inside the kernel. The detection algorithm is determined by a security policy specification language. However, a single policy cannot capture all the attack scenarios, LogicDroid needs different policies to be able to handle new forms of attacks. The current implementation of LogicDroid only allows updating of policies by using the offline generated loadable kernel module.

Because of the complexity in changing the policy, this Final Year Project was created to simplify the process. The purpose of this project is to modify the structure of LogicDroid's security monitor so that modification of the policies can be done on-the-fly in a running instance of LogicDroid, without having to do offline compilation. This involves a redesign of the monitor to include a logic interpreter that can take as an input a security policy and updates its enforcement subroutines.

The project contains two parts: the first part is the implementation of intermediate interpreter to interpret the policy specification language to string data structure that can be read by the monitor. The second part is to enable a secure path from Application level to Linux kernel to allow updating the policy in a running instance of LogicDroid.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Nanyang Technological University, School of Computer Engineering and Asst. Prof Alwen Fernanto Tiu for granting me this opportunity and providing me with continuous guidance throughout the Final Year Project. Advices given by my project supervisor Asst. Prof Alwen Fernanto Tiu has been a great help during the project.

I also would like to extend my thanks to the Hendra Gunadi and Hoang Ngoc Hung for technical knowledge they have given me, helped me to understand the project.

Table of Contents

List of Figures

List of Tables

I. Introduction

In the last few years, Android has become the most popular mobile operating system for smartphones and tablets [5]. Android uses the layer architecture including 4 levels: application, framework, library, Linux kernel. At its core, Android relies on the user-based protection features provided by Linux kernel that runs each application as user in a separate process with its own set of data structures and prevents other processes from interfering with its resources and execution. This security mechanism is called Application Sandbox. At the install time, a unique user ID (UID) will be assigned to the application based on the permissions it declares in the Manifest file. These permissions will decide which groups this application belongs to and limits the resources that the application can access. The Application Sandbox is not only in the Linux kernel but also extended to the upper layers. Android uses inter-process communication (IPC) facility to provide communication between applications running in different processes. An additional security feature of Android is the permission mechanism. At runtime, when an application wants to call an API, Android will check whether the application has the permissions that is required to access this API

Despite all the security mechanisms mentioned above, Android still encounters some vulnerabilities, one of them is privilege escalation attack. Privilege escalation attack is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. There are two types of attacks that can lead to privilege escalation: the confused deputy attack and the collusion attack. In the confused deputy attack, an application does not have the permission to the certain services but still is able to access it by calling another process which exposes the interface to this functionality without any guards. The collusion attack requires two or more malicious apps to collaborate. This kind of attack does not exploit the system components but need the collaboration among applications to get the fully required permissions.

To detect the privilege escalation attack, a solution is to track the history of the call chains among the applications and processes. The main detection algorithm was introduced based on the following statement: If the application A calls application B followed by a call from application B to C within a certain reasonably time, then A is considered to call C indirectly. However, it is almost impossible to keep the call chains history of every application and process. The detection algorithms are based on the metric linear-time temporal logic (MTL) with simplified past time operators and recursive definitions [1] that does not need to store and analysis entire call chains history but only the calculation result of the last call event.

LogicDroid is a customized Android operating system that contains a security extension based on the mentioned above algorithm to capture the privilege escalation attacks. By adding various hooks in the Android OS, the call chains among applications and processes can be tracked by the monitor inside the kernel. The detection algorithm is determined by the security policy which is specified in MTL. However, a single policy cannot capture all the scenarios, the LogicDroid need the different policies to be able to handle new forms of attacks. The current implementation of LoigcDroid only allows updating policy by using the offline generated loadable kernel module.

Because of the complexity in changing the security policy in LogicDroid, this Final Year Project was created to simplify the process. The purpose of this project is to modify the structure of LogicDroid's security monitor so that modification of the policies can be done on-the-fly in a running instance of LogicDroid, without having to do offline compilation. This involves a redesign of the monitor to include a logic interpreter that can take as an input a security policy and updates its enforcement subroutines.

To summarize, my contributions are as follows:

1. The Implementation of intermediate interpreter to interpret the policy specification language to string data structure that can be read by the monitor.
2. Enable a secure path from Application level to Linux kernel to allow updating the policy in a running instance of LogicDroid.

The rest of the report is organized as follows: Section 2 talks more details about Android Security Mechanism and the general ideas of LogicDroid. Section 3 explains the implementation of the old and new version of LogicDroid. Section 4 discusses the testing and optimization. Section 5 talks about the application of LogicDroid to the real world problems. Section 6 is the conclusion and related discussion about the future works.

II. Overview

2.1 Android Architecture Overview

This section talks about the security features of the Android platform. *Figure* 01 describes the security components and various layers of the Android software stack. In the Android OS, only small amount of code running as root, the other layers above the Linux Kernel are restricted by the Application Sandbox.

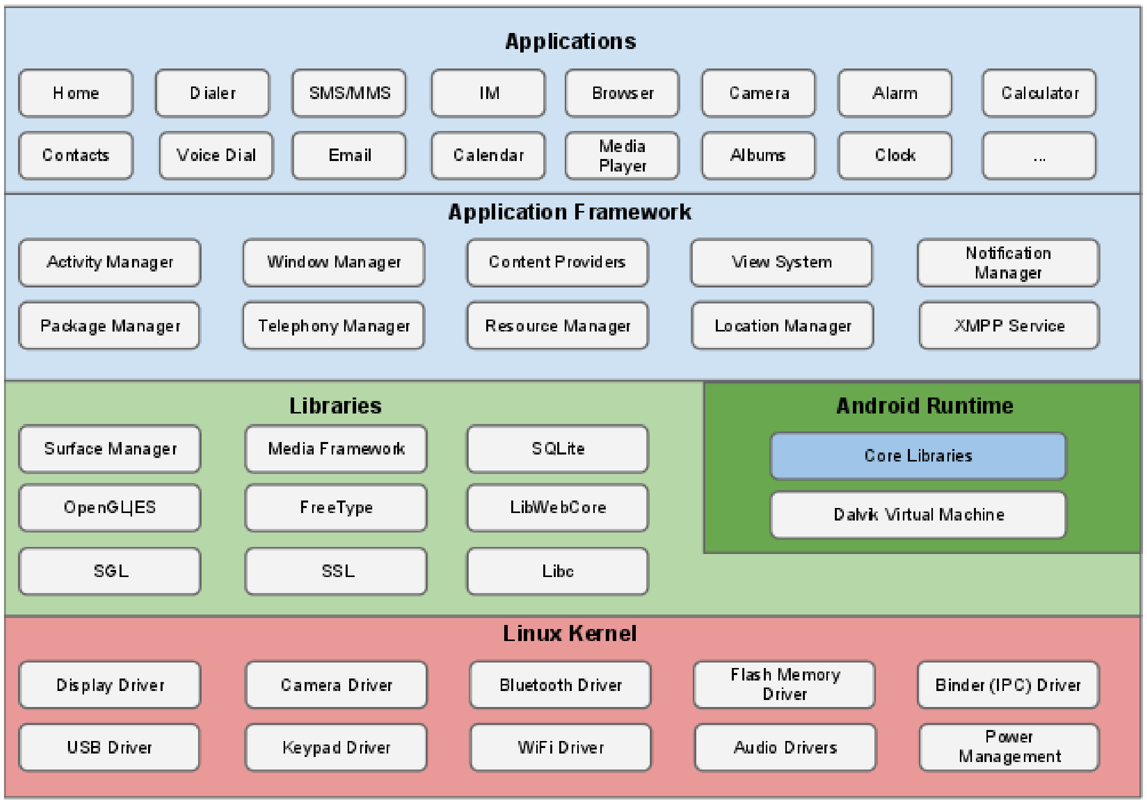


Figure 1: Android Software Stack

The main Android platform building blocks are:

- **Application Layer:** This layer is the topmost layer of the Android stack. It contains all the applications that the user can interact with.
- **Framework Layer:** This layer provides high-level services to application layer. It includes the programs manage the basic functions of Android such as Resource Manager, Telephony Manager, …

- **Library Layer:** This layer includes Android’s native libraries. Libraries carry a set of instructions to guide the device in handling different types of data. This layer is also an interface between Framework layer and Linux kernel.
- **Linux Kernel:** The Linux kernel provides the basic system functionalities. The main security enforcements are done inside Linux kernel.

At the operating system level, the Android platform provides the security of the Linux kernel, as well as a secure inter-process communication (IPC) facility to enable secure communication among applications running in different processes [4].

2.1.1 The Application Sandbox

The Android platform uses the Linux user-based protection mechanism to identify and isolate the application resources. The Android system assigns a unique user ID (UID) to each Android application and runs it as that user in a separate process [3].

This sets up a kernel-level Application Sandbox. The kernel enforces security mechanism between applications and the system at the process level through standard Linux facilities, such as UNIX access control system based on user and group IDs that are assigned to applications. By default, applications cannot interact with each other and applications have limited access to the operating system. If one application tries to do something malicious like reading another application's data, then the operating system protects against this because the application does not have the appropriate user privileges. The sandbox is simple, auditable, and based on UNIX user separation of processes and file permissions [4].

The Application Sandbox is not only applied in the kernel. This security model is also extended to native code and to operating system application. All the layers above the Linux kernel in Figure 01, including application layer, framework layer and library layer run within the Application Sandbox. Because of the Application Sandbox, the only way to communicate between applications and processes is using inter-process communication (IPC) facility in Android.

2.1.2 System Permission

In general, there are two kinds of permission: the permissions which are used to check at the runtime and the permission which are only used to assign the correct UID to application. At install time, Android gives each package a distinct Linux user ID based on the permission which are declared in *AndroidManifest.xml*. Some of the resources in Linux kernel are only available to the specific groups. If the application permissions are accepted by the user, the package manager will grant the specific UID which belongs to the corresponding group that has the authority to access the resources in Linux kernel.

For example: If an application wants to have access to the internet and be able to use GPS, it needs to declare the following permissions in AndroidManifest:

```
<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

At the installation time, when user accepts these permissions from the application, this application will be given the UID belongs to two groups. The first groups can access the Internet and the second group can access the GPS.

The second type of permission is used in runtime checking. When one application calls an API in Framework layer, Android will check for the permission which the API needs, if the application has the required permission, the security enforcement will be checked at process level as a user are trying to access some specific resources. The failure in one of these steps will result in a Security Exception being thrown back to the application.

2.2. Android Security Issues: Privilege Escalation Attack

Despite all the security mechanisms mentioned above, Android still has some vulnerabilities, one of them is privilege escalation attack. Privilege escalation attack is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. There are two types of attacks that can lead to privilege escalation: the confused deputy attack and the collusion attack.

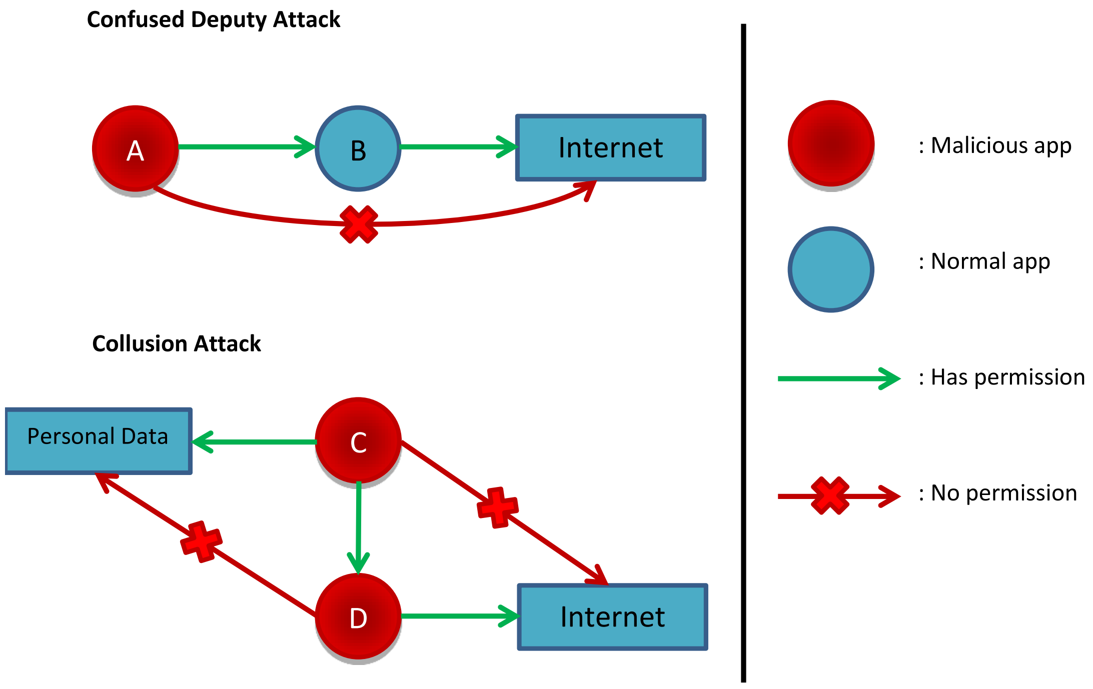


Figure 2: Confused deputy attack and Collusion attack

In the confused deputy attack, an application that does not have the permissions to the certain services but is able to access the services by calling another process that exposes the interface to this functionality without any guards. For example: Any Android application that does not have the Internet permission is still able to send data to Internet by calling the default browser app.

The collusion attack requires two or more malicious apps to collaborate [2]. This kind of attack does not use the system exploitation but need the collaboration between applications to get the fully required permissions. In the *Figure 02*, C has permission to Personal data, D has permission to access Internet, C and D can collaborate to send the personal data to Internet without notifying the user.

2.3 LogicDroid

LogicDroid is a customized Android Operating System which contains the implementation of a runtime security monitor. The security monitor enforces the security policy specified by a security policy specification language. The security policy language is based on metric linear-time temporal logic (MTL) [1]. MTL features temporal operators that are indexed by time intervals, allowing one to specify timing-dependent security policy. To capture privilege escalation, the extension of MTL with recursive definitions that are used to express call chains between apps. The main monitor sits inside the LogicDroid's kernel and the kernel allows inserting generated monitors modularly to change the policy.

The original project contains 4 main components:

- **Android OS**: Various hooks were added in Android OS that allows tracking the call chains among applications and processes. When the call chain event is detected, it will be sent to the Monitor for further evaluation.
- **Linux kernel**: The runtime Monitor was implemented inside the kernel to process the call events. If the event violates the security policy, the call will be blocked. Because the policy can be changed, the security monitor was implemented as loadable modules so that changing of policies can be done by loading different modules.
- **Policy Monitoring**: The Policy Monitoring was programmed in Java that takes the input from policy xml files and generates C source code for monitor module. This source code then will be cross-compiled with the Linux kernel to obtain the loadable module.
- **Trusted Group Manager App**: Some static relations are needed for the policy violation detection from kernel such as UID of system apps or UID of trusted apps which was set by the user. This app allows user to have the higher control to the security of LogicDroid.

2.3.1 Metric linear-time temporal logic with recursive definition

To detect the privilege escalation attack, a solution is to track the history of the call chains among the applications and processes. The main detection algorithm was introduced based on the following statement: If the application A calls application B followed by a call from application B to C within a certain reasonably time, then A is considered to call C indirectly. However, it is almost impossible to keep the call chains history of every application and process. By using metric linear-time temporal logic (MTL) with simplified past-time operators and recursive definition, the detection algorithm does not need to store and analysis entire call chains history but only the calculation result of the last call event. The equation of *trans* the reflexive transitive closure of call is showed below:

$$trans(x,y) \coloneqq call(x,y) \lor \exists z. (\diamondminus_n trans(x,z) \land call(z,y))$$

where call denotes the IPC event. To be practically enforceable in Android, the monitoring algorithm must be trace-length independent [1]. Therefore, some past-time metric operators such as the *Diamond Dot* and the *Since* operators are used and simplified so that the algorithms do not require to keep the history of all the events but only the last call event.

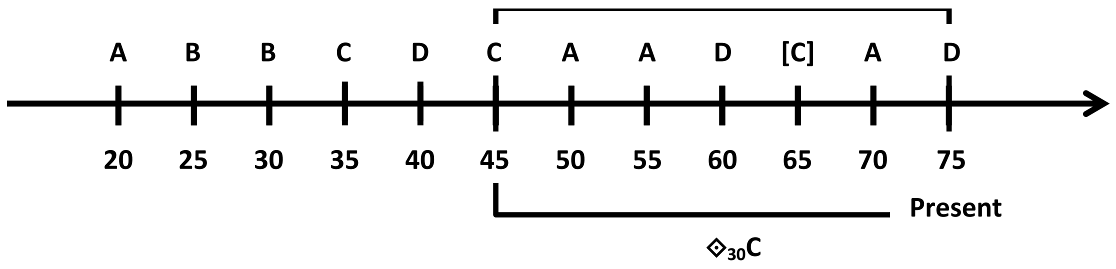
The following figures show the definitions of these past-time operator. These figures present the history of events (A, B, C, D) with the corresponding timestamp. The arrow illustrates the order of events based on timestamp.

1. Diamond Dot: $\diamondsuit_n \psi$ is true when intuitively ψ holds within n time units in the past. For example:

    a. $\diamondsuit_{10}$D is false because D did not hold within the time 65 to 75 (present)

    b. $\diamondsuit_{30}$B is false because B did not hold within the time 45 to 75 (present)

    c. $\diamondsuit_{30}$C is true because C held at the time 65 (within 30 time units to present)

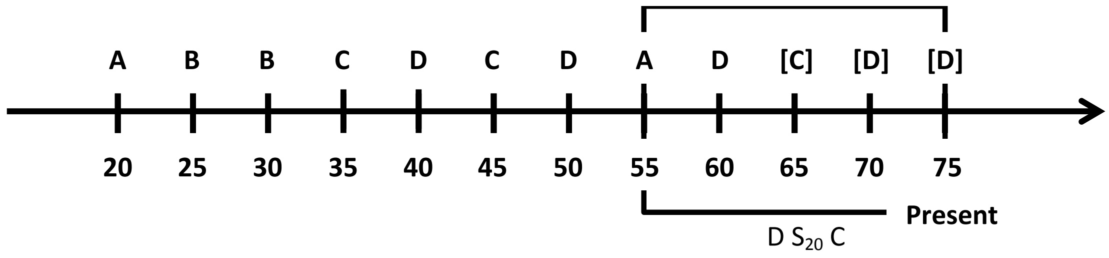


2. The *Since* operator: $\psi_1 \, S_n \, \psi_2$ (read ψ_1 since ψ_2) is true when ψ_2 holds or ψ_1 holds and ψ_2 held right before the chain of present ψ_1 events holds within n time units in the past. For example:

    a. A S_{10} D is true because the present event is D;

    b. D S_{20} A is false because the present event is not A and the event held before the chain of D events was C (not A) at 65.

    c. D S_{20} C is true because the C held right before the chain of D events holds and D is the present event.



2.3.2 The monitor sitting inside the Linux kernel

The LogicDroid implementation consists of two parts: the codes that generate a monitor given a policy specification, and the modifications of Android framework and its Linux kernel to hook the monitor to intercepts IPCs and access to Android resources.

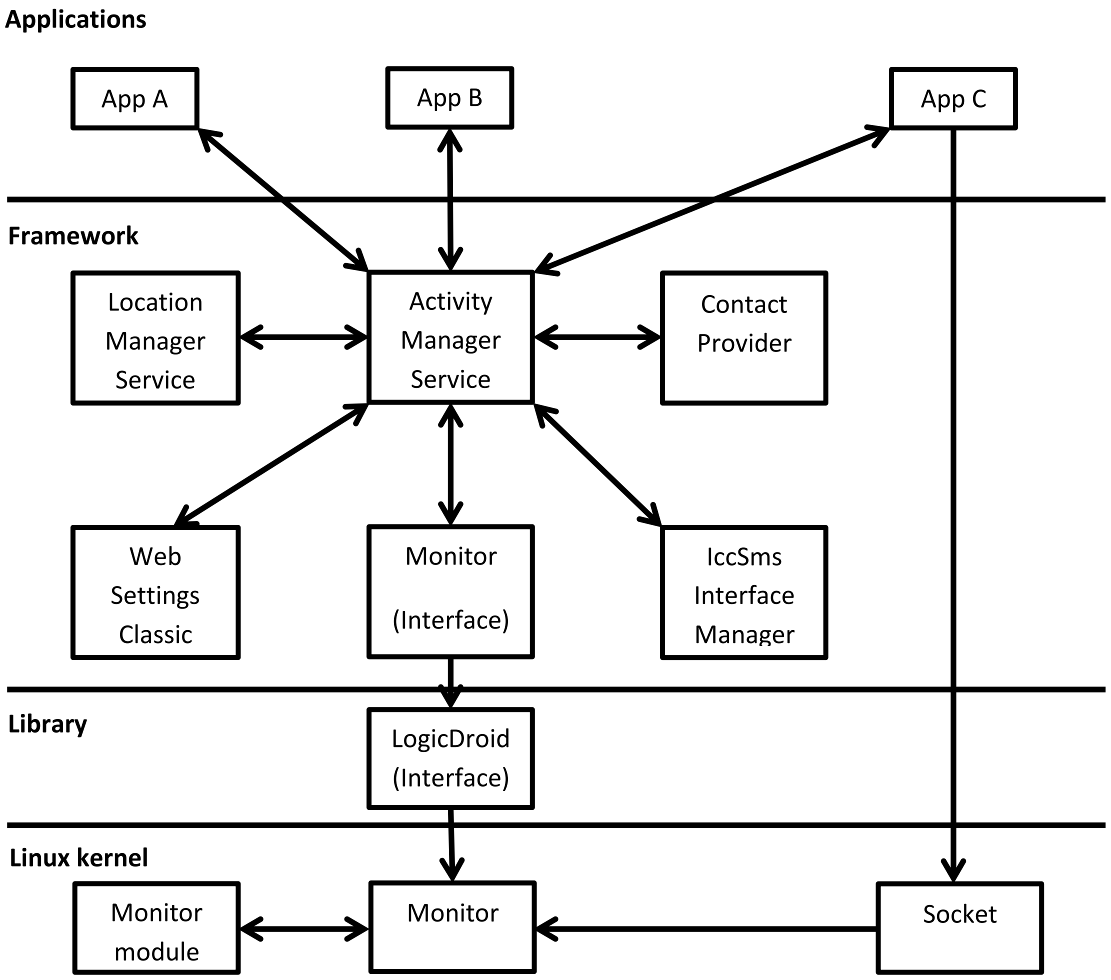


Figure 3: LogicDroid Architecture

The monitoring algorithm presented in previous section is implemented in the monitor inside Linux kernel. The reason for this is that there are some cases when a process requests kernel resources. If the monitor was in the upper layers of Android architecture, the call events from kernel could not be communicated up to the upper layer.

The programming language that was used to develop the reference monitor is the C language. Some kernel system calls are defined to allow the events to be passed down from upper layers along with the application's user id. The monitor will decide to whether block or allow the call to proceed depending on the policy that is implemented. Because the security checking mechanisms are implemented on top of the Android permission mechanism, so in the case where the reference monitor does nothing, the default Android permission mechanism would still be enforced.

2.3.3 The modification in Android OS

The modifications in Android are mainly done in the Linux kernel and framework layer, the library layer was modified to enable the communication between framework layer and the Monitor. Because of the Application Sandbox, all the communication among applications and processes can only be done via inter-process communication (IPC) mechanism. Therefore, to be able to detect call chains between applications or processes, we only need to supervise the IPC. By adding various hooks in the Android OS (mainly in the Activity Manager Service in framework layer), the call chain events can be detected.

The hooks are also placed in the Linux kernel. This is due to some of the applications or processes request to use the resources that might be inside the kernel. In this case, this kind of event cannot be detected in IPC because the process does not need to communicate with other process. To be able to detect this kind of event, the virtual UIDs are created for some specific resources and the hooks are placed just before resources are granted to the process. When a process makes the request, a virtual call event is created and sent to the monitor to simulate the call from UID of process to UID of the resource.

2.3.4 Updating the policy

The following graph shows the current design of Policy updating mechanism:

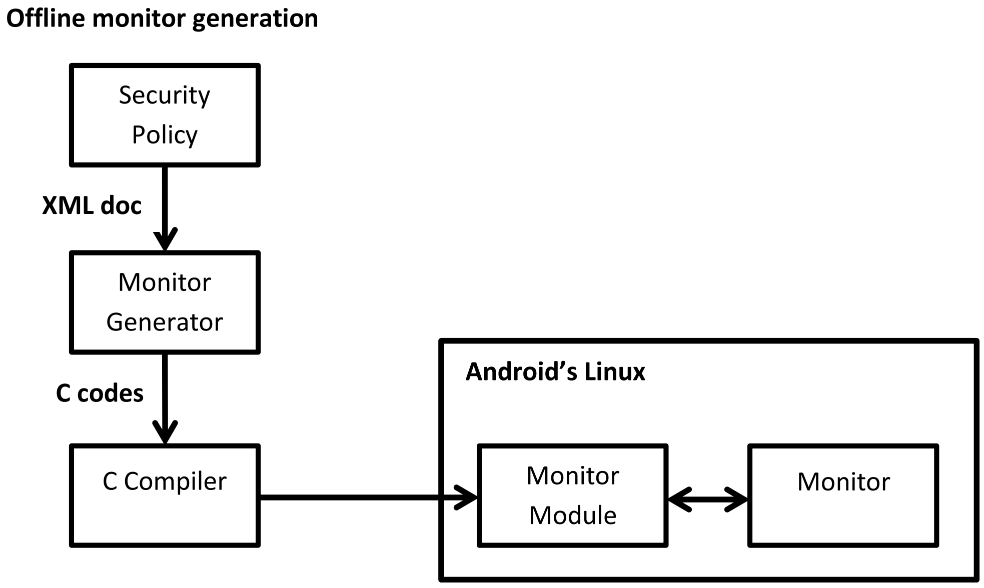


Figure 4: Policy Updating Mechanism

The current security policy to be enforced is hardcoded in the monitor for efficiency reasons. This, however, makes it slightly complicated to update the security policy in the monitor. It is necessary to update security policies since this might be required to counter new forms of attacks that may not be handled by the current policy. The current approach is to design the monitor as a loadable kernel module. Loadable kernel module is an object file that contains code extending the running kernel and can be removed and reinstalled in a live system. This module can be generated by an offline monitor generation process. The monitor generator takes as input a security policy (specified in RMTL, using XML as the representation language) and generates C codes that correspond to the kernel module of the policy. To be able to run in the specific android kernel, the module source code needs to be cross-compiled into the kernel object file. This file then will be put into LogicDroid and use the adb command *insmod* to install. This process is complicated due to the need of generating, recompiling the entire module source code and using the command to install the module. The whole process cannot be done on-the-fly in a running instance of Android.

2.4 LogicDroid: On-the-fly security policy update

Because of the complication in changing the policy as mentioned in the previous sections, this Final Year Project was created to simplify the process. The purpose of this project is to modify the structure of the monitor so that modification of the policies can be done on-the-fly in a running instance of Android, without having to do offline compilation. This will involve a redesign of the monitor to include a logic interpreter that can take as an input a security policy and updates its enforcement subroutines.

The main work packages need to be done in this project:

- Modify the Policy Monitoring class which generates the actual monitor module, to output an intermediate format, instead of C code. This intermediate format will then be able to read from the Monitor inside the kernel.
- Modify the security monitor to be able to parse and interpret the input from the upper layers of Android OS and eliminate the use of the loadable kernel module in changing the policy.
- Modify Android framework layer and library layer to create the secure path from application level to the Linux kernel to change the security policy.
- Develop an application that can take the XML file as input policy, interpret the policy file into string data and send this data to the Monitor inside the kernel.

The following graph shows the new design of Policy updating mechanism:

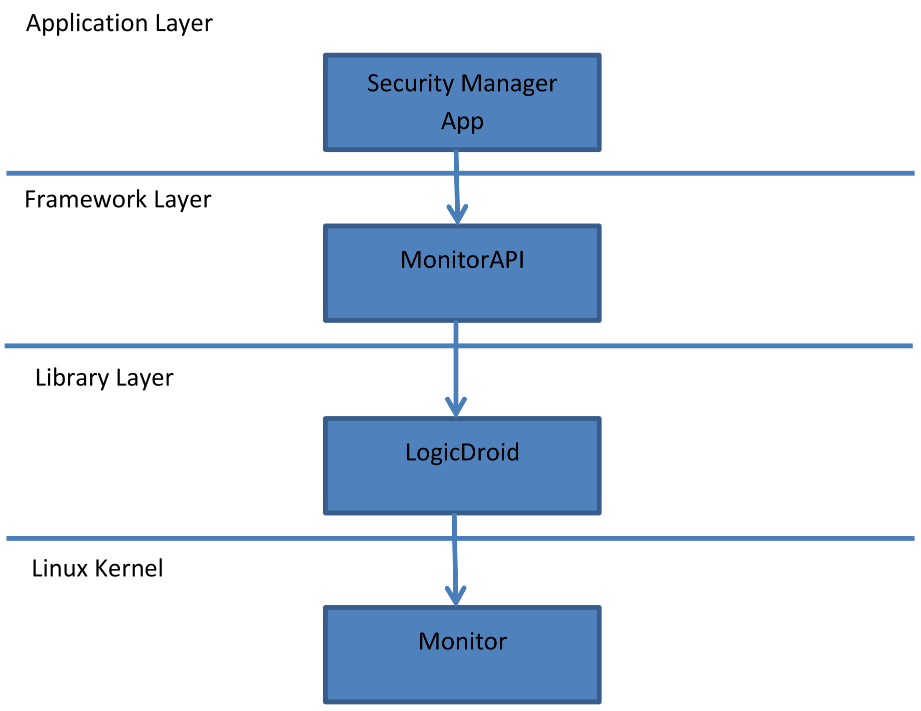


Figure 5: New design of policy updating

III. Implementation

3.1 LogicDroid source code

Before starting the project, I want to understand the mechanism of the Monitor inside the Linux kernel.

The Monitor communicates with the library layer by using the system calls. The *sys_LogicDroid_checkChain* and *sys_LogicDroid_modifyStaticVariable* system calls are implemented to send the call event and other data to the Monitor. However, the Monitor itself cannot process the call event, it requires the module to be able to execute the policy. This is because of the policy updating mechanism. If the policy is hardcoded in Monitor, when the policy is changed, the entire Linux kernel needs to be recompiled and replaced. Therefore, the loadable kernel module is used so that only the module needs to be recompiled. The main event processing functions are hardcoded in each module. The Monitor acts like an interface between Library layer in Android OS and the module.

When Android starts, the Monitor is initialized. All the call events which are passed to the Monitor are considered to not violate the policy if the module is not installed. The module registers itself by calling the function *LogicDroid_registerMonitor*. This function sends the address of main module functions to the monitor. The monitor uses the function pointer to store the addresses. After the module is installed, these function pointers are used to call the module functions to process the call events. *Figure 06* shows the interactions among Library layer, the Monitor and the module.

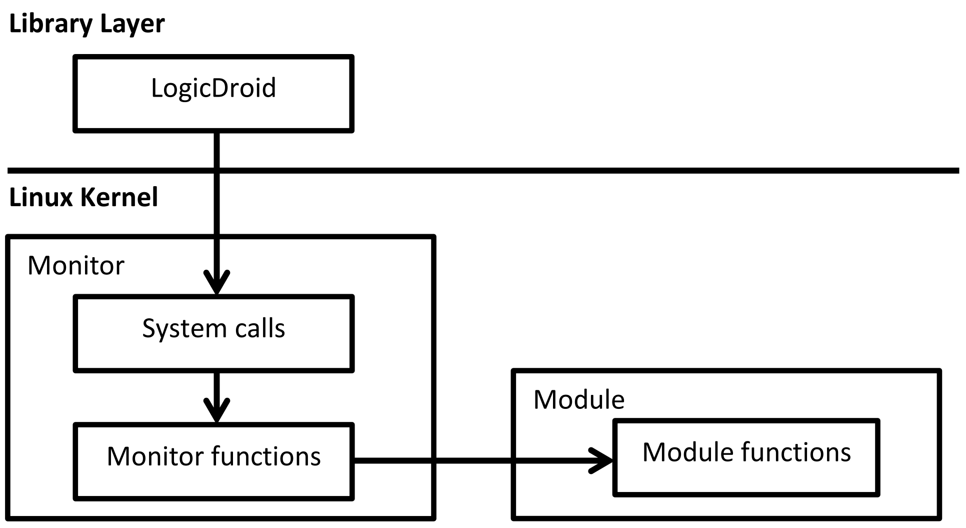


Figure 6: Interactions among Library Layer, Monitor and Module

The main functions which are used to process the data from Android OS:

- LogicDroid_checkEvent: This function uses the event data to execute the policy. If it returns 1, the call in the Android OS will be blocked.
- LogicDroid_renewMonitorVariable: The static relations are required to execute the policy. This function changes the value of static relation in the module.
- LogicDroid_initializeMonitor: This function initializes the UID of all applications and virtual UIDs that are set for Linux resources. When an application is installed or removed, the LogicDroid will call this function to reset the monitor.

When the module is uninstalled, it calls the function *LogicDroid_unregisterMonitor* to unregister itself. This function removes all the data that was set previously. From this point, all the call events in the Android OS are allowed to proceed.

To change the policy updating mechanism, I need to implement all the main functions in the Monitor so that the event and data processing will be done inside the Monitor. This implementation will be discussed in the next sections.

The module source code is generated by Policy Monitoring. Because my first implementation is the development of interpreter to interpret the policy specification language to string data structure, I will focus on how the Policy Monitoring works. The main functions of the Policy Monitoring are to get the input policy and use that policy to generate the module source code. The input policies are stored in XML format so that its structure can be easily read in Java by using the default Document Object Model (DOM) methods.

3.1.1 Policy Monitoring

Policy Monitoring is the module source code generator that takes an input policy encoded in XML format. It detaches the policy into some sub-formulas and determines the order between sub-formulas based on the order of operations. These sub-formulas are used to generate the event data processing functions in module source code.

Structure of Security Policy XML file

This section describes the basic structure of a security policy.

```
<policy>
   <policyId></policyid>
   <formulas>
      <formula>
      </formula>
   </formulas>
</policy>
```

The *<policyid>* tag stands for the id which is used to differentiate each policy. The main formula is defined in *<formula>* tag with its *<type>* attribute value is *main*. Inside the *<formulas>* tag, there can be many user-defined formulas but only one main formula. The user-defined formulas can be recursive. (The sample for a policy XML file can be found in the Appendix A).

The main formula contains a number of sub-formulas. Each of them is one operation which is illustrated by the different tags. There are 7 basic operations:

Operation Name	XML tag	Operation Name	XML tag
Exist	<exist>	For All	<forall>
And	<and>	Diamond Dot	<diamonddot>
Or	<or>	Since	<since>
Not	<not>		

Table 1: Table of logic operations

In each operation tag, there is a list of elements. The element can be an operation, a relation or a free variable.

Relation and free variable declaration

The relations in policy file are presented by the *<atom>* tag. There are two type atoms: The static atoms and dynamic atoms. Static atoms are the relations whose values do not depend on the event. The static relation value can be set by the user, allow the user to have higher controls on the policy violation detection. The static relation has to be declared in the security policy XML file. The following structure defines a static relation (*system*):

```
<atom>
   <atom_type>static</atom_type>
   <rel>system</rel>
   <var>x</var>
</atom>
```

The static relation *system* is presented in the monitor module source code as a boolean array with the size equals to the total number of application and process's UIDs and virtual UIDs. If the application is system app, its corresponding element in the array will be set to *true*. This is how the monitor can store the data of static relations.

The <*var*> tag indicates the free variable which is used in this *system* relation. Basically, the user can create the static relations with any name. However, to be able to change the value of these relations, the application has to know the relation name. Only some of the relations such as: *system* and *trusted* are hardcoded in Trust Manager App, allow user to decide whether the app or process is system app or trusted app. The other static relations with arbitrary name can be defined by user but it requires a special application to be able to change its value such as the relation: hasPermissionToSink.

Dynamic Atoms are the relations whose values are automatically updated when an event comes. An essential dynamic relation that every security policy has to contain is *call*. This relation indicates the call between apps or processes. The dynamic atoms also can be defined by the user such as *trans*. Unlike the static relations, the system knows when to change the value of dynamic relations. The following structure defines the call relation of an app to Internet:

```
<atom>
   <atom_type>dynamic</atom_type>
   <rel>call</rel>
   <var>x</var>
   <var>object:internet</var>
</atom>
```

The *call* relation normally has two free variables. However in this policy, the second free variable is set to a constant value of Internet UID. Therefore to get the value of this relation, we only need to know the value of the first free variable.

In the next step, Policy Monitoring tries to generate the main formula step by step. Policy Monitoring detaches the Policy into some sub-formulas, each sub-formula is one operation. These operations have the determined order and are more easily to generate.

The following figure shows how the main formula can be detached into the sub-formulas

```
// The main formula: exist_x((call(x, internet) and !(trusted(x)) and
!(system(x))))

// This formula can be detached into 4 sub-formulas

// [3] : !(trusted(x))

// [5] : !(system(x))

// [1] : (call(x, internet) and !(trusted(x)) and !(system(x)))

// [0] : exist_x((call(x, internet) and !(trusted(x)) and !(system(x))))
```

Figure 7: Main formula detachment

After detaching the policy, Policy Monitoring generates the sub-formulas in the event processing function. The policy will be calculated based on the order of the sub-formulas.

When modifying the monitor to enable on-the-fly update, the policy execution still needs to be based on the sub-formula calculation. Therefore, instead of passing the whole policy to the monitor, my approach is to detach the policy and use the sub-formula as the input data.

3.1.2 Monitor Module Source code

To store the policy execution data, the monitor module use a data structure definition called *History*. All the data such as the timestamp of the event, the time metric of operations, the value of relations and the policy calculation is able to access by reading the *History* structure.

History Structure

The following figure shows the how the *History* Structure is declared.

```
typedef struct tHistory {

    char ***propositions;

    char **atoms;

    int **time_tag;

    long timestamp;

} History;
```

Figure 8: History Structure Declaration

The *propositions* array is the data used to calculate the policy. The first index is the formula index. If the first index = 0, it indicates the main formula, if this index is greater than 0, it indicates the user-defined formula such as *trans*.

The *atoms* array is the array of relations. This array is used to make the data in *propositions* consistent and easier to modify. Some of elements in *propositions* point directly to *atoms* elements. Therefore, in order to modify the value of one relation, we only need to modify the value of *atoms* elements, the change will also apply for *propositions* elements.

The *time_tag* is an array of time metric if the policy has some operations require metric time such as diamond dot and since. The last structure variable is *timestamp* which records the timestamp of the call event that triggers the policy execution.

The *History* structures are initiated in *LogicDroid_module_initializeMonitor* by the function *History_Constructor*. This function allocates the memory to the *History* depending on the policy complexity. This function also points some of the *propositions* elements to *atoms* elements for the data consistency and set the value of *time_stamp*.

There are two *History* pointers: *next* and *prev*. The *next* pointer points to the current value of policy calculation while the *prev* pointer points to the previous value of policy calculation. Some of the logic operations require the uses of *prev* such as diamond dot and since. The final result of policy calculation is *next->propositions[0][0][0]*.

After calculating the policy, the value of *next* and *prev* will be swapped if the event did not violate the policy. The *prev* pointer will point to the current value of policy calculation and the *next History* will be reset. By using the *History_Reset* function, the value of entire propositions array will be reset to 0. Some of the dynamic relation elements are also be reset but the links between *propositions* and the *atoms* remain unchanged. This means the *History* structures still stay in the memory, only the values are reset.

The *History* structure is an important part in policy violation detection. However, the data and memory which is used to initialize the *History* depends on the policy, it was hardcoded in the module source code and generated by Policy Monitoring. Therefore, to implement the new policy updating mechanism, my intention is to keep the *History* structure and use the input data to initialize the *History*.

3.2 LogicDroid on-the-fly security policy updates

My first implementation is to develop an intermediate interpreter to interpret the policy specification language to string data structure that can be read by the monitor. This interpreter is actually the modified version of Policy Monitoring because the Policy Monitoring can generate the module source code, all the necessary data that used for the policy calculation can be easily archived by modifying the Policy Monitoring source code. In this section, I only want to focus on the structure of the input string. The input string structure will tell how the interpreter interprets the policy specification language and how the Monitor can process the data.

Basically, the Monitor inside the kernel can be able to implement to read complex policy structure. However, to reduce the computation time and the workload of the kernel, I decided to pass the most basic data to the Monitor. By doing this way, the Monitor does not need to process the data but can use it directly for the policy violation detection. The reason I choose the string format for input data is that the monitor only reads the input data once. It does not affect the performance of policy violation detection and is easier to implement.

The first step of implementation is to find out which part of module source code is independent with the policy. By reading and adding some lines in the source code to print out, I can determine the parts of the module that are not changed with different policies. The unchanged parts are the function declarations, data structure, variable declarations and constants. The major changed parts are the event data processing functions and memory allocation.

Therefore, I decided to separate the data which is used to pass to the Monitor into two tables: Dependency Table and Resources Allocation Table. The Dependency Table will be the main policy data which is used to calculate the policy in *LogicDroid_checkEvent* function. The Resources Allocation Table is how the memory is allocated and the how the arrays of *History* are related to each other.

3.2.1 Dependency Table Structure

As mentioned in the previous section 3.1.1, the Policy Monitoring generates the event data processing function in module source code by detaching the policy and hardcoding the sub-formulas one by one based on the order of operations. My approach is to consider sub-formulas as the data input for the event processing function in monitor. The Dependency Table will contain a list of Dependency Table Records, each record is the necessary data to calculate the sub-formulas.

The following graphs describe the differences and the similarities between the original source code in Policy Monitoring and the generated code in the module.

Source code:

```
...

formulaStr.append(indent[add_idx] + "for (" + var +" = 0; " + var +" < app_num && !next->propositions[" + policy_number + "][" + curr_idx + "][" + firstIndex + "]; " + var +"++){\n");

...

formulaStr.append(indent[add_idx] + "next->propositions[" + policy_number + "][" + curr_idx + "][" + firstIndex + "] = next->propositions[" + policy_number + "][" + curr_idx + "][" + firstIndex + "] || next->propositions[" + policy_number + "][" + subIndex + "][" + secondIndex + "];\n");

...
```

Figure 9: Policy Monitoring source code is used to generate Exist operation

Generated code:

```
// [0] : exist_x((call(x, internet) and !(trusted(x)) and !(system(x))))

for (x = 0; x < app_num && !next->propositions[0][0][0]; x++) {

    next->propositions[0][0][0] = next->propositions[0][0][0] || next-
>propositions[0][1][x];

}
```

Figure 10: Generated Exist operation in Monitor module

There are the main parts which are independent to the policies, the changed parts are actually the indices. The number of indices required to generate one sub-formula is fixed. Therefore the collection of indices can be used as the input data to the Monitor, it will be in the Dependency Table.

In comparison between *Figure 09* and *Figure 10*, we can easily recognize that the data needed to generate Exist operation are *var*, *policy_number*, *curr_idx*, *firstIndex*, *subIndex*, *secondIndex*. The *var* is the free variable, the *policy_number*, *curr_idx*, *subIndex* are the constants. The *firstIndex* and *secondIndex* are the expressions of free variable. In this case: *secondIndex* is *x*.

The similarities can be found in other operations, the number of needed data is limited. It requires the free variables, the fixed indices and the expressions of free variables (From this point to the end of the report, I use the word Variable Index for the expression of free variables to be consistent with the implemented source code). These data will be a part of Dependency Table Record.

Dependency Table Record Structure

Each of the sub-formula will be presented as one record in Dependency Table. The Dependency Table contains a list of Dependency Table Records. In the string format, it starts with the number of Dependency Table Records, following by the Dependency Table Record list. Each record has the necessary data for one operation function: The number of free variables, The Variable Indices and the fixed variables.

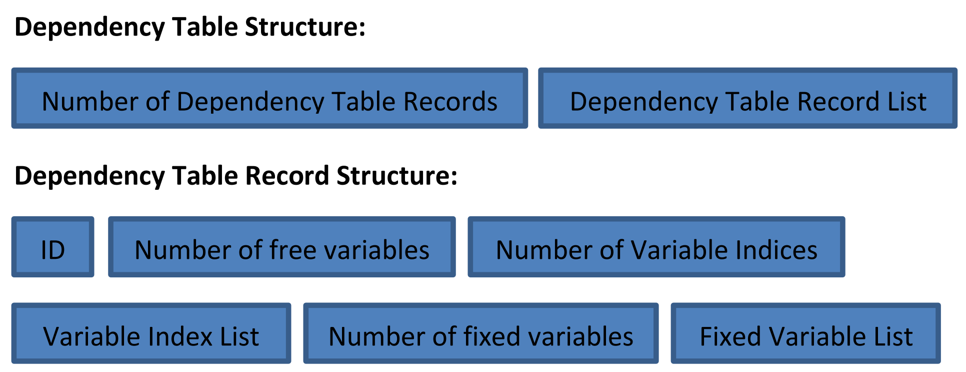

Dependency Table Structure:
Number of Dependency Table Records
Dependency Table Record List
Dependency Table Record Structure:
ID
Number of free variables
Number of Variable Indices
Variable Index List
Number of fixed variables
Fixed Variable List

Figure 11: The Dependency Table Record Structure

Notes:

- ID is the ID of the operation function that takes this Dependency Table Record as input data.
- The number of free variables is used to calculate the Variable Indices.
- The number of Variable Indices is used to read the Variable Index List. The detail for the Variable Index Structure is discussed in the next section.
- The number of fixed variables is used to read the Fixed Variable List.

The order of sub-formulas is also the order of Dependency Table Record. When the policy is calculated, the Dependency Table Record will be read one by one in this determined order.

Variable Index Presentation

The Variable Index is the expression of free variables. To present the Variable Index in string format, I use the number of Variables following by the Variable list. Each Variable in the Variable Index can be either free variable or a constant. For example the following policy:

$$\exists x. \left(\left(call(x, INTERNET)\right) \land \neg system(x) \land \neg trusted(x) \right)$$

The Variable Index which presents in the relation *call(x, internet)* is *x * app_num + 1* while the Variable Index which presents in the relations *call(x, y)* is *x * app_num + y*. Therefore, I decided to use the following structure to store the Variable in the Variable Index:

```
// This data structure is used to present the Variable
// (i.e: constant : 0,1,... - variable: x,y,z,...)
typedef struct tVariable{
    //type = 0 -> constant | 1 -> variable
    int type;
    int value;
} Variable;
```

If the Variable is constant, its type attribute is 0 and its value attribute equals to the constants. If the Variable is free variable, its type attribute is 1 and its value attribute equals to the free variable index. For example the following nested loops which have 3 free variables:

```
for (x = 0; x < app_num; x++)
    for (y = 0; y < app_num;  y++)
        for (z = 0; z < app_num; z++)
```

The value for Variable x is 0, for Variable y is 1 and for Variable z is 2. All x, y, z have the type attribute is 1. The Variable Index such as *x* app_num* app_num + 2 * app_num + z* can be presented as a list of 3 variables. The first one is *x* (type = 1; value = 0), the second one is *2* (type = 0; value = 2) and the last one is *z* (type = 1; value = 2). The string format is 3|1|0|0|2|1|2 (The first number is the length of Variable Index)

3.2.2 Operation Function

The policy is detached by Policy Monitoring into the sub-formulas with the determined order. To compute these sub-formulas, I created the operation functions. Each operation function corresponds to one logic operation as shown *Table 1*. The operation functions are programmed based on the Policy Monitoring source code. All the data these functions require is stored in Dependency Table Record. The Dependency Table is a public array, therefore to refer to the correct Dependency Table Record, the operation function need the index of the record. At the first step, the operation function tries to read the number of free variables and the fixed indices.

The figure below show how the Exist operation function is implemented:

```
// EXIST - ID = 0
int exist_function(History *next, History *prev, int recordIdx){
    int freeVarNo = dependencyTable[recordIdx].freeVarNo;
    int policyNo, currIdx, subIdx, mainLoopsNo, i, firstCalIdx;
    if (freeVarNo < 1)
        return 0;

    policyNo = dependencyTable[recordIdx].fixIdx[0];
    currIdx  = dependencyTable[recordIdx].fixIdx[1];
    subIdx   = dependencyTable[recordIdx].fixIdx[2];
    mainLoopsNo = power(freeVarNo);

    for (i = 0; i < mainLoopsNo; i ++){
        firstCalIdx = freeVarCal(freeVarNo, i, dependencyTable[recordIdx].varIdx[0]);
        next->propositions[policyNo][currIdx][firstCalIdx] = next-
>propositions[policyNo][currIdx][firstCalIdx] || next-
>propositions[policyNo][subIdx][freeVarCal(freeVarNo, i,
dependencyTable[recordIdx].varIdx[1])];
        if (next->propositions[policyNo][currIdx][firstCalIdx])
            return 0;
    }
    return 0;
}
```

Figure 12: Example of operation function

In the operation function, instead of using multiple nested loops, I used one loop with the same number of iteration. The *mainLoopsNo* is the number of iterations that need to calculate the operation properly. This is the power of the free variables number with the base *app_num* (See the *Figure 10*). The Variable Indices are computed by the function *freeVarCal()* in each iteration. To be able to calculate the Variable Index, the value of free variables need to be computed based on the number of free variables and the current iteration number. After all the indices are determined, the function does the normal logic calculation.

Each operation function has 3 parameters: The *next* and *prev* are *History* pointers and one integer *recordIdx*. *recordIdx* is the execution order of operation functions. The ID variable of the Dependency Table Record will decide which operation function is used. By using the function pointers, the source code are shorten and more efficient in comparison with using one switch or many if/else statements.

The following figures show how the function pointers are used in the Monitor:

```
// Function pointer (used in History_Process)
int (*functionPointer[11])(History * next, History * prev, int recordIdx);
...
// Function pointer using in History_Process
functionPointer[0]  = exist_function;
functionPointer[1]  = and_function;
functionPointer[2]  = or_function;
functionPointer[3]  = not_function;
functionPointer[4]  = forall_function;
functionPointer[5]  = diamond_dot_metric_function;
functionPointer[6]  = diamond_dot_function;
functionPointer[7]  = diamond_metric_function;
functionPointer[8]  = diamond_function;
functionPointer[9]  = since_metric_function;
functionPointer[10] = since_function;
```

Figure 13: Function Pointer declaration

History_Process function is the core function in LogicDroid_checkEvent that is used to execute the policy.

```
// This function is used to execute the policy
char History_Process(History *next, History *prev) {
    int recordIdx;
    for (recordIdx = 0; recordIdx < dependencyTableRecordNo; recordIdx++)
        (*functionPointer[dependencyTable[recordIdx].id])(next, prev,
recordIdx);
    return next->propositions[0][0][0];
}
```

Figure 14: History Process function

After finishing the implementation of operation functions, I want to test the Monitor with the Dependency Table as the input. To be able to do so, I modified the module to be runnable in Linux. I removed the module declaration functions and replaced the functions which only run in Linux kernel (kmalloc, kfree) by the common library functions (malloc, free).

In the first plan, I intended to put the string input in a file inside the kernel, so that when I change the policy, the policy data will be written in that file. After the LogicDroid is rebooted, the new policy will automatically be applied. Therefore, I created a file to store the Dependency Table. Because at this time, the Resources Allocation Table had not finished yet, I modified the source code from Policy Monitoring that allows the module can run on Linux for the testing purpose. The module was generated normally by Policy Monitoring with hardcoded Resources Allocation Table but the Dependency Table is read from file. After testing the correctness in the logic of the operation functions, the remaining part of string input is implemented.

3.2.3 Resource Allocation Table Structure

The Resource Allocation Table shows how the resources can be allocated in the Monitor. Because each policy requires different amount of memory, the Resource Allocation Table is created to allocate exactly what the policy needs and specify the link between some components of Monitor. This is an important step before executing the security policy.

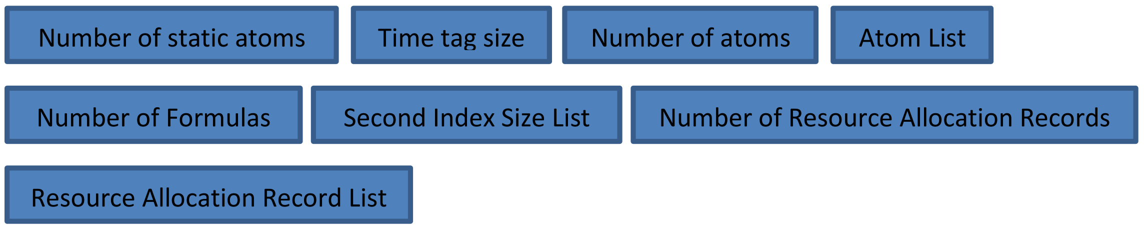


Figure 15: The Resources Allocation Table Structure

Notes:

- The time tag size is the number of operation which requires the metric
- Number of atoms is used to read the Atom List
- Number of Formulas is the size of first index of propositions array in History. This number is also used to read the Second Index Size List.

- Second Index Size List is the list of second index's size for propositions array in History
- The number of Resource Allocation Records is used to read the Resource Allocation Record List

As mentioned in the previous *History* Structure section, the *History_Constructor* function allocates the memory to *History* based on the policy. The following figure shows how the *History_Constructor* works with the policy:

$$\exists x. \left((trans(x, INTERNET) \wedge \neg system(x) \wedge \neg trusted(x) \right)$$

```
History* History_Constructor(long timestamp) {

    History *retVal = (History*) kmalloc(sizeof(History), GFP_KERNEL);
    retVal->atoms = (char**) kmalloc(sizeof(char*) * 3, GFP_KERNEL);

    retVal->atoms[0] = (char*) kmalloc(sizeof(char) * app_num * app_num, GFP_KERNEL); //
call
    retVal->atoms[1] = trusted; // trusted
    retVal->atoms[2] = system; // system

    retVal->propositions = (char***) kmalloc(sizeof(char**) * 1, GFP_KERNEL);
    retVal->propositions[0] = (char**) kmalloc(sizeof(char*) * 7, GFP_KERNEL);
    retVal->propositions[0][0] = (char*) kmalloc(sizeof(char), GFP_KERNEL);
    retVal->propositions[0][1] = (char*) kmalloc(sizeof(char) * app_num, GFP_KERNEL);
    retVal->propositions[0][2] = retVal->atoms[0];
    retVal->propositions[0][3] = (char*) kmalloc(sizeof(char) * app_num, GFP_KERNEL);
    retVal->propositions[0][4] = retVal->atoms[1];
    retVal->propositions[0][5] = (char*) kmalloc(sizeof(char) * app_num, GFP_KERNEL);
    retVal->propositions[0][6] = retVal->atoms[2];

    retVal->timestamp = timestamp;
    return retVal;
}
```

Figure 16: An example of *History_Constructor()*

According to the figure above, indices for the propositions and the memory allocation size are the constants. However, some of the propositions elements are pointing to atoms elements and some atoms elements are pointing to the static relation array elements.

Resource Allocation for Atoms

In the generated version of Monitor, the relation arrays are created with fix name such as *call*, *trusted*, *system*, *trans*. The *trusted* and *system* relations are static relations, their values will remain unchanged after calculating the policy. The other relations *call* and *trans* are dynamic relations. The value of dynamic relation is changed whenever the policy is computed. I created the arrays called *staticAtoms*, this is the array of static relations. Both the static atoms from *next* and *prev* History are set to point to this *staticAtoms* array. This means the values *next* and *prev* static atoms will be the same, they will both be changed at the same time when *staticAtoms* changes. The dynamic atoms such as *call*, *trans* are different in *next* and *prev* History. Therefore I need to allocate the memory for these kinds of atoms. I used the following structure to present the atom type:

```
typedef struct tAtom{
    //For Atom : type 0 -> static | 1 -> dynamic
    int type;
    int value;
} Atom;
```

Figure 17: Atom structure

If the *type* variable is 0, the atom is static and *value* variable is the index in *staticAtoms* array, this static atom will point to *staticAtoms* element with that index. If the *type* variable is 1, the atom is dynamic and the *value* variable is the atom's dimension.

For example, the atoms declaration in Resource Allocation Table in *Figure 14* is presented in the String format as shown below:

3| 1|2 | 0|0 |0|1

3 is the Number of atoms, 1|2 | 0|0 | 0|1 is the Atom List (see the *Figure 14*)

1|2 is the *call* dynamic relation that has the 2 dimensions

0|0 and 0|1 are the static relations: *system* and *trusted*

Resource Allocation Record structure

The Resource Allocation Record is used to allocate the resources to the propositions array in History. According to the resource allocation to propositions array as shown in *Figure 04*, the first index and second index of propositions array are constants. The propositions element can be mapped to the atoms element or allocated memory. The structure of the Resources Allocation Record should be able to differentiate between these two types of allocations. The structure of Resource Allocation Record:

```
typedef struct tResourceAllocationRecord{
    //type 0 -> allocate | 1 -> mapping
    int type;
    int firstIndex;
    int secondIndex;
    int value;
} ResourceAllocationRecord;
```

The *type* variable indicates the kind of the resource allocation. If *type* = 0, the propositions element will be allocated memory which has the size is power of *value* variable with the base *app_num*. If *type* = 1, the propositions element will be mapped to the atoms which has the index is the *value* variable.

For example, the resource allocation of propositions in *Figure 16* is presented in the String format as shown below:

7 | 0|0|0|0 | 0|0|1|1 | 1|0|2|0 | 0|0|3|1 | 1|0|4|1 | 0|0|5|1 | 1|0|6|2

7 is the Number of Resource Allocation Records
0|0|0|0 | 0|0|1|1 | 1|0|2|0 | 0|0|3|1 | 1|0|4|1 | 0|0|5|1 | 1|0|6|2 is the Resource Allocation Record List (see the *Figure 15*)

After finishing both the Dependency Table and Resources Allocation Table processing functions, I want to change the monitor module back to a loadable module in Linux kernel to test with the Android Emulator. Some of the libraries are not supported inside the Linux kernel, therefore I need to reprogram all the required functions.

Because of the insecure in reading and writing files inside the kernel, I decided to pass the argument to the module when it is installed. To allow arguments to be passed to the module, I declared the variables that will take the values of the command line arguments as global and then used the *module_param()* macro, (defined *inlinux/moduleparam.h*) to set the mechanism up. At runtime, *insmod* filled the variables with any command line arguments that were given. After installing the module, the input data was read and used to initialize the variables. This method has the advantage of being able to done in a running instance of Android, the application only need to use the *insmod* command with the calculated arguments. However to be able to install the module this way, the application has to have the super user permission

The Monitor is now successfully run inside the kernel and be able to pass all the test cases which were used with the original version. In the next step, I tried to remove the module and merge the Monitor and the module source code. To be able to do this, I need to recompile the entire kernel every time I modify the Monitor source code.

3.3 Framework and library layer modification

The next implementation of the Monitor is to get the data directly from the upper level. Because the original version of LogicDroid is able to send the event and data from upper level of Android OS to the Linux kernel, I follow the exactly same way to implement the data path. This requires adding more system calls in the Linux kernel. The system call is how a program requests a service from an operating system's kernel. I decided to add two more system calls: *sys_LogicDroid_registerMonitor* and *sys_LogicDroid_unregisterMonitor*. This implementation requires the modification and recompilation of the entire Linux kernel and Android library layer.

The Java Native Interface is used in Android library layer in order to enable the communication between Java files in Operating System and C files in Linux kernel. The two JNI methods are implemented to connect to the new system calls. This allows the API in the framework layer be able to use these functions to register and unregister the Monitor.

In order to be able to send data from Application layer, the data have to go through Framework layer. In this layer, I created two APIs: *registerMonitor()* and *unregisterMonitor()*. This two APIs can only be called if the user has the following permission.

```
<uses-permission android:name=”android.permission.CHANGE_POLICY_MONITOR”/>
```

At the runtime, when an application wants to call these APIs, the system will check for the permission before executing the request. If the application does not have the *CHANGE_POLICY_MONITOR* permission, a Security Exception will be thrown.

These two APIs are implemented in Monitor.java. This class contains the important methods: *initializeMonitor()*, *checkEvent()* and *renewMonitorVariable()* that are used to manipulate the Monitor inside the kernel.

3.4 Security Manager App

Because Policy Monitoring is written in Java, it is able to integrate the Policy Monitoring to an Android application. The Security Manager App is implemented using the source code from Policy Monitoring. The new feature has been added so that the Security Manager app can get the policy xml files from SD card and from the Internet. The permission of Monitor API (as shown in the previous section) was included in application Manifest in order to be able to register and unregister policy in Monitor.

IV. Testing and Optimization

4.1 Testing

The testing was done continuously during three important phases of implementation.

First phase of project: The testing was performed to verify correctness of logic operation functions. In this phase, the module was modified to be able to run as a normal C program. The events were inputted manually with the pre-calculated testing scenarios. These testing scenarios are the chain of call events simulation. The values of static relations were easily to change at the runtime and the values of each dynamic relation were checked carefully after each event for both the *next* and *prev* History. At the end of this phase, all the operation functions passed the test and the module gave the desired values for the policy calculation.

Second phase of the project: In this phase, I need to test the correctness of reading input data and some implemented functions which do not have the supported library in Linux kernel. All of these functions were verified by adding to separate C programs. I also need to test the use of memory functions (kmalloc, kfree). To be able to do this, the module was modified to be able to get the arguments in the install time. At the end of this phase, the functions worked properly and the module did not show any kernel corrupted error when allocating and freeing memory for all the testing scenarios.

Third phase of the project: The testing was performed to verify the behavior of the system after finishing implement the Security Manager App. The policy now can be changed from Application level. The testing is aim to test the whether the system is properly functioning.

For the second phase and the third phase, the following applications are used to test the system:

- **Trust Manager App**: Change the value of static relations
- **Socket Manager and Fun App**: Demonstrate the privilege escalation attack on accessing Internet and sending SMS
- **LocalServer.java**: The data sink that receives data sent by Socket Manager
- **CRT Kolme**: Demonstrate the confused deputy attack on making hidden phone call

4.2 Benchmark and Performance

The security policy now can be changed in the application level when the LogicDroid is running and the monitor works properly with different policies. I want to do the benchmark to compare the performance of the current design with the generated module mechanism. The versions of LogicDroid which were used to benchmark are the original version with generated modules and the latest version with on-the-fly update policy.

To do the benchmark, I constructed a chain of ten apps, making successive calls between them, and measured the time needed for one end to reach the other. I measured two different average timings in milliseconds (ms) for different scenarios, based on whether the apps are in the background cache (i.e., suspended) or not.

The policies were used in this benchmark are:

- `Policy 1: exist_x((call(x, internet) and !(trusted(x)) and !(system(x))))`
- `Policy 2: exist_x((trans(x, internet) and !(system(x)) and !(hasInternetPermission(x))))`
- `Policy 3: exist_x((trans(x, call_privileged) and !(system(x)) and !(trusted(x))))`
- `Policy 4: exist_x((trans(x, internet) and !(system(x)) and !(trusted(x)) and DiamondDot(call(x, contact))))`

The following figures show the results of the benchmark:

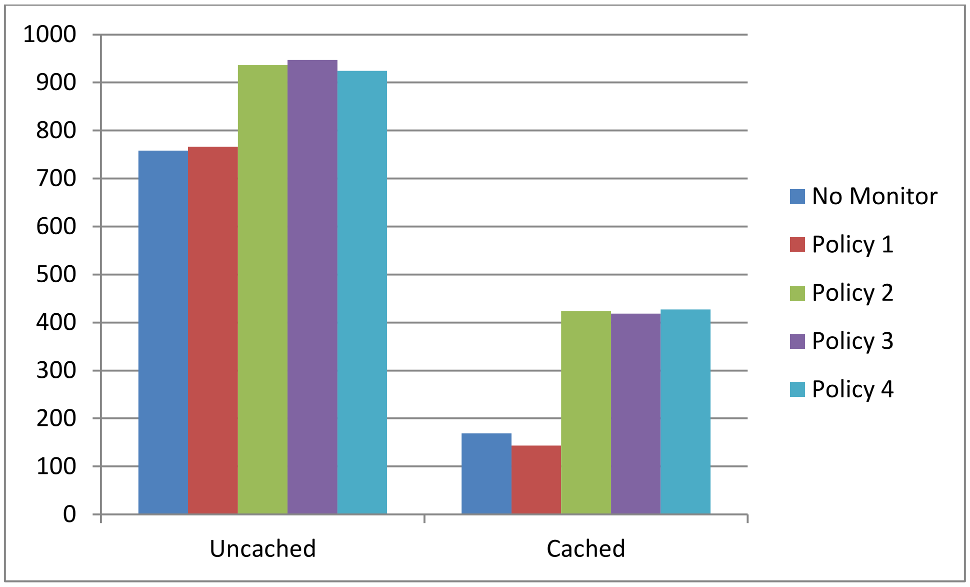


Figure 18: The performance of LogicDroid with loadable kernel module

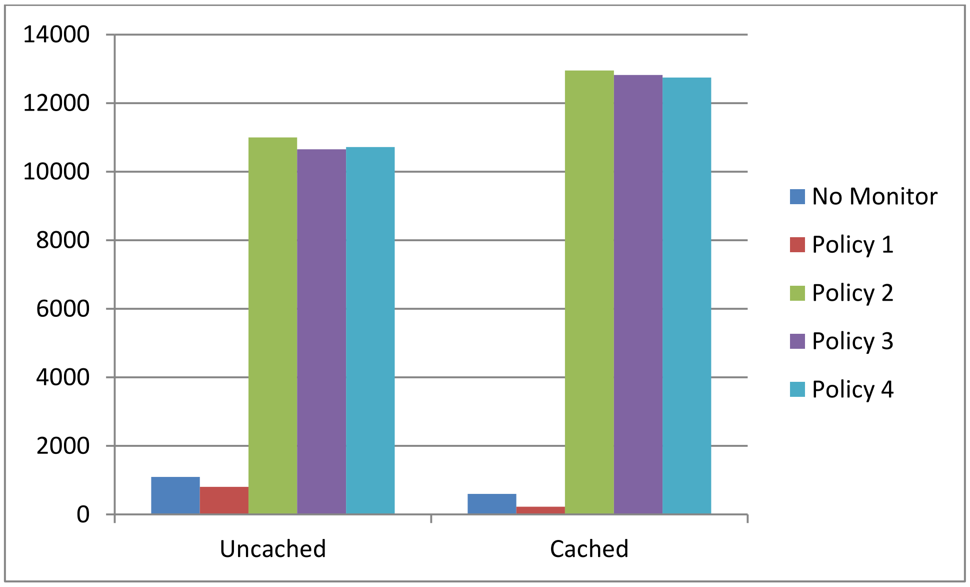


Figure 19: The performance of LogicDroid with on-the-fly policy update

Based on the results of the benchmark as shown in the *Figure 16* and *Figure 17*, the performance of the new version of LogicDroid is much slower (roughly more than 10 times) in comparison with the old version. This might due to the inefficient algorithms I used when implementing the Monitor. In the next step, I will try to optimize the Monitor implementation.

4.3 Optimization

The graph in *Figure 17* shows the performance of the current version of LogicDroid with different policies. The significant difference in the execution time between the Policy 1 and the other policy can be easily recognized. The inefficient part might be in the function which is needed to calculate the policy 2, 3 and 4.

The difference between the policy 1 and other policy is that, the other policy has 3 free variables while the policy 1 only has 1. The slowdown factor might come from the calculations in each iteration in operation function. I did several experiments and find out the computation the index function was the main reason that slows down the performance.

```
// Calculate the value of firstIndex, secondIndex
int freeVarCal(int freeVarNo, int loopNo, VariableIndex varIdx){
    int i;
    int value = 0;
    int kfreeVarCurrValue[freeVarNo];

    for (i = freeVarNo - 1; i >= 0; i--) {
        kfreeVarCurrValue[i] = loopNo % app_num;
        loopNo = loopNo/app_num;
    }

    for (i = 0; i < varIdx.varLength; i++) {
        if (varIdx.var[i].type)
            value = value * app_num + kfreeVarCurrValue[varIdx.var[i].value];
        else
            value = value * app_num + varIdx.var[i].value;
    }
    return value;
}
```

Inside the *freeVarCal* function, I used the expensive operator division and modulo to calculate the free variables and the free variables are need to recalculate in every iteration. To reduce the calculation time, I need to reduce the number of loops, multiplications and divisions. I came up with the new idea to calculate the Variable Index.

```
void calculateCountArray(){
    int i;
    countArray[0]++;

    for (i = 0; i < MAX_COUNT && countArray[i] == app_num; i++) {
        countArray[i] = 0;
        countArray[i+1] ++;
    }
}
```

Figure 20: New algorithm to calculate the free variables

I used an array called *countArray* to store the value of *freeVariable*, for each loop in operation function; the value of *countArray* is updated. Doing this way, I do not need to recalculate all the free variables but still has enough number of iterations for each operation function. The loop in each operation function is implemented as shown in the following figure:

```
while(countArray[freeVarNo] == 0) {
    calculateCountArray();
    ...
}
```

Figure 21: The loop implementation in each operation function

The following graph shows the execution time after optimization:

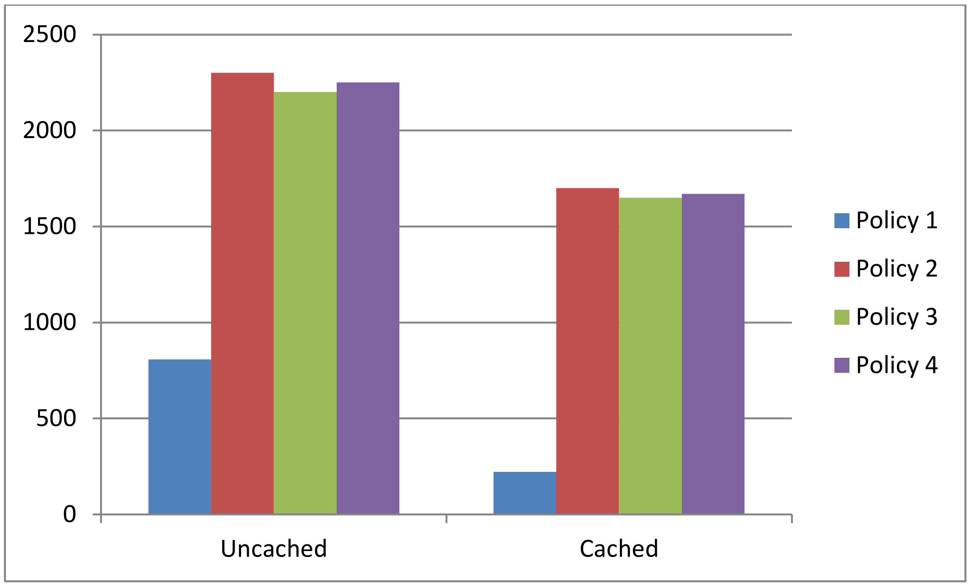


Figure 22: The performance of LogicDroid for the new free variable calculation

According to the result, the performance is 4 times better. However, the execution time is still very high in comparison with the original version of LogicDroid. I tried several methods to improve the LogicDroid performance but they are only optimized for the policy which has less than 4 free variables.

1. I create the mathematical function which is used to calculate the Variable Index, this function's formula is changed depend on the Variable Index structure. The Variable Index structure only needs to be read one to construct this calculation function. When the monitor wants to compute the value of Variable Index, it does not need to re-read the Variable Index structure and the free variable is passed directly to the calculation function, therefore it helps to improve the performance.

The following figure shows the running time of the LogicDroid with this method:

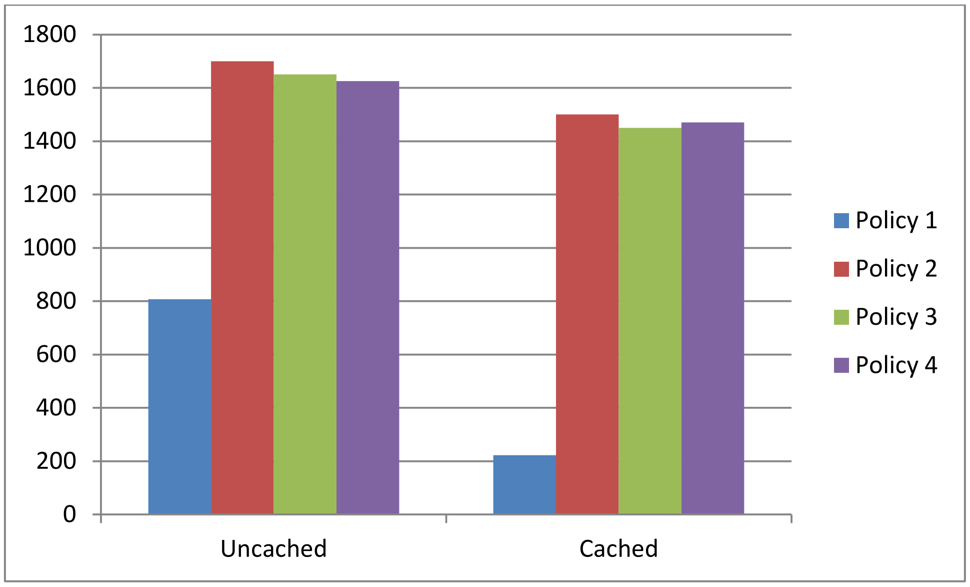


Figure 23: The performance of LogicDroid with method 1

2. The next value of Variable Index is decided by the previous value

For each operation function, I read the structure of the Variable Index once, and establish the formula for the Variable Index that the next value of Variable Index can be obtained by adding the current value with constant. This method does not require the calculation of free variables and does not need to read the structure of Variable Indices in the loop. All the multiplications are changed to additions.

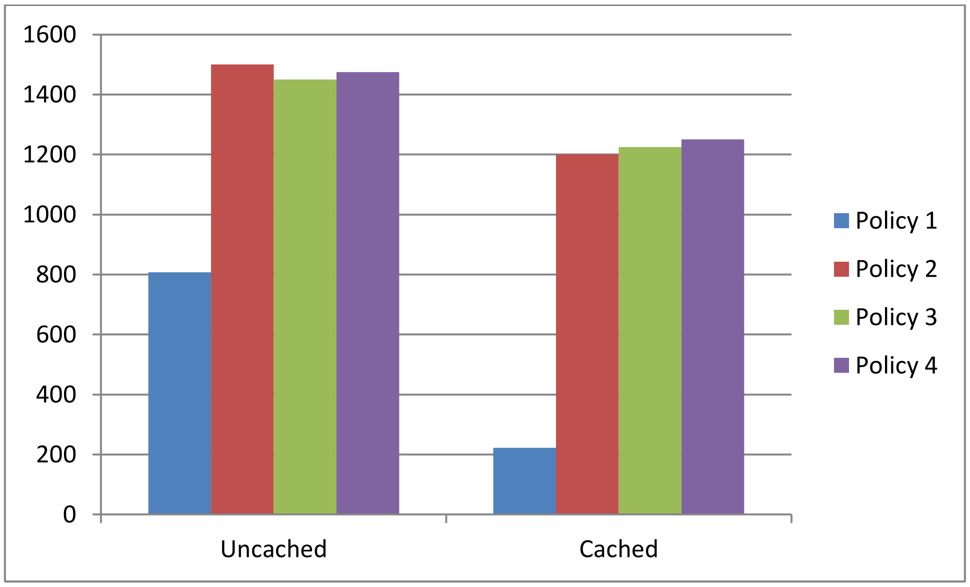


Figure 24: The performance of LogicDroid with method 2

These above methods help to improve the performance of LogicDroid. However they are only optimized for the policies which have less than 4 free variables.

V. Application of LogicDroid to real world problems

This section will discuss how security mechanism of LogicDroid is able to apply to real world problem.

In July 2014, a security vulnerability was found by the website CureSec.com. This vulnerability involves the exploitation of the bug in *com.android.phone* that could be used for privilege escalation attack. An application with no specific permission can be able to make a hidden phone call without notifying the user. This is a typical confused deputy attack that was mentioned earlier. Because LogicDroid was implemented before the discovery of this vulnerability and there is a hook in Android OS which can detect the event when an process try to access the resources to make a phone call, LogicDroid should be able to detect this privilege escalation attack. The following policy is used to attempt to stop the exploit:

$$\exists x.\,(\,(trans(x, CALL\ PRIVILEGED) \wedge \neg system(x) \wedge \neg trusted(x)\,)$$

where the *trans* predicate is as defined earlier in Section 2.3.1. In this policy, the CALL PRVILEGED is the virtual UID for the resources that can be used to make the phone call. If exist a call event from a process x to the CALL PRIVILEGED resources directly or indirectly where x is not a system app and not a trusted app, the call event will be considered to violate the policy and will be blocked.

I have tested the runtime monitor against this particular exploit. CureSec.com provides a prove-of-concept app (called "Kolme") that demonstrates this exploit. The test is to run this app inside LogicDroid emulator, which is still at version 4.1.1, and so still contains the vulnerable codes. The monitor detected the call from Kolme to *com.android.phone* followed by the call from *com.android.phone* to CALL PRIVILEGED within a certain reasonably short time. This means Kolme tried to call CALL PRIVILEGED indirectly. Therefore if user trusts Kolme app and sets the value of *trusted(Kolme)* to true, the action of call event from Kolme to CALL PRIVILEGED will be legal and does not violate the policy. If Kolme is not a trusted app, the call event will violate the policy and will be blocked by LogicDroid.

VI. Conclusion

In this report, I have shown the general idea of LogicDroid to detect the privilege escalation attack. By placing various hooks in Android OS, LogicDroid is able to monitor the call chains based on RTML algorithms. The offline policy updating mechanism with generated loadable kernel module is replaced by the on-the-fly updating mechanism. The new method has the advantage of allowing the policy to be changed from the application level in a running instance of LogicDroid. The Security Manager Application is developed to be the intermediate interpreter to interpret and pass the policy data in the input string format to the Monitor inside the kernel. The new policy updating mechanism is tested with different scenarios to assure that the LogicDroid works properly and be able to capture the privilege escalation attacks. However, the performance of the new implementation is still low in comparison with the previous version.

Future work: Implement new logic operators to be able to capture more form of attacks and continuously secure, optimize the system.

References

[1] H. Gunadi and A. Tiu, “Efficient runtime monitoring with metric temporal logic: A case study in the android operating system,” in FM 2014, ser. Lecture Notes in Computer Science, vol. 8442, 2014, pp. 296–311.

[2] H. Gunadi and A. Tiu, “Android Security: A Case for Runtime Verification”

[3] Abhishek Duneyi and Anmol Misra, “Android Security – Attack and Defenses”

[4] Android Open Source Project, 2015. *Security*
Available at: https://source.android.com/devices/tech/security/index.html
[Accessed: 2015-02-28]

[5] Android Developers, 2015, *Android, the world’s most popular mobile platform*
Available at: http://developer.android.com/about/index.html
[Accessed: 2015-03-21]

Appendix A – Policy XML file sample

The XML file for the policy:

$$\exists x.\,(\,(trans(x, INTERNET) \wedge \neg system(x) \wedge \neg trusted(x)\,)$$

```xml
<?xml version="1.0" encoding="UTF-8"?>
<policy>
   <policyid>4</policyid>
   <formulas>
      <formula>
         <type>main</type>
         <exist>
            <var>x</var>
            <and>
               <atom>
                  <atom_type>dynamic</atom_type>
                  <rel>trans</rel>
                  <var>x</var>
                  <var>object:internet</var>
               </atom>
               <not>
                  <atom>
                     <atom_type>static</atom_type>
                     <rel>system</rel>
                     <var>x</var>
                  </atom>
               </not>
               <not>
                  <atom>
                     <atom_type>static</atom_type>
                     <rel>trusted</rel>
                     <var>x</var>
                  </atom>
               </not>
            </and>
         </exist>
      </formula>
      <formula>
         <type>rec:trans</type>
         <or>
            <atom>
               <atom_type>dynamic</atom_type>
               <rel> call </rel>
               <var> x </var>
               <var> y </var>
            </atom>
            <exist>
               <var> z </var>
               <and>
                  <diamonddot>
                     <metric> 10 </metric>
                     <atom>
                        <atom_type>dynamic</atom_type>
                        <rel> trans </rel>
                        <var> x </var>
                        <var> z </var>
                     </atom>
                  </diamonddot>
                  <atom>
                     <atom_type>dynamic</atom_type>
                     <rel> call </rel>
                     <var> z </var>
                     <var> y </var>
                  </atom>
               </and>
            </exist>
         </or>
      </formula>
   </formulas>
</policy>
```