Enhancing fuzzing technique for OKL4 syscalls testing

Amaury Gauthier*, Clément Mazin*, Jean-Louis Lanet* and Julien Iguchi-Cartigny*
*Xlim — Smart Secure Devices
University of Limoges
France
Email: {amaury.gauthier, clement.mazin, jean-louis.lanet, julien.cartigny}@xlim.fr

***Abstract*—Virtual machine monitor is an hot topic in the embedded community. Apart from high end system, current processors for embedded system do not have any instructions helping to virtualize an OS. Based on this fact, most of the current hypervisor for embedded devices use the paravirtualization technique. Especially OKL4 which is a derived L4 micro-kernel and implements among other virtualization of Linux kernel.**

We introduce our ongoing work for testing the security of OKL4. We have chosen to focus on the most low level OKL4 interface usable from an external actor that is the syscall API. Because all operating system components use directly or indirectly these syscalls, a minor flaw at this level can impact in chain the entire system including a virtualized kernel.

After trying random naive fuzzing of OKL4 syscalls which have not been concluding, we have developed a model describing the OKL4 syscalls. This model also describes all constraints applicable to a syscall. Based on these models, we are working on a tool using the constraints to compute a reduced set of syscall input values which are highly likely to generate flaws in OKL4 if they aren't fully checked by the kernel.

***Keywords*-Virtual Machine, Embedded System, Kernel Security, Syscalls**

I. Introduction

Virtualization is an attractive technology which brings a lot of flexibility whatever the application domain. Current server technologies bring a good trade-off between virtualization advantages and security. While the embedded virtual machine monitor technology are very close to the conventional virtual machine monitor, the scale between the server world where virtualization take all of its sense and the embedded world is huge. Indeed, server virtualization are now almost inevitable, but in the embedded world, this technology is only at its beginning.

In this paper we propose a technique to test the security of virtual machine monitor for embedded hardware. Our work aims to provide a reusable method to test these specific software pieces without the complexity and the cost of formal methods while keeping a good test coverage.

We begin this paper with a section to explain main arguments in the favor of embedded virtualization in a mobile phone context. Then, we clarify the limitations of the fuzzing based testing method. In the third section, we expose a new grammar used by our tools. The fourth section is dedicated to the interaction with the fuzzer Peach. The fifth section evaluate our current works. The two last sections expose the related work and our future work.

II. Motivation/Background

Contrary to server context, embedded hypervisor are still in development and we don't know yet all its implication on the current embedded applications. We have chosen to expose some arguments in favor of the mobile phone virtualization because it seems to be currently the most advanced. Indeed, virtual machine monitors for embedded systems promise several enhancements in the mobile phone world.

A. Virtualization in mobile phone

We expose here some of the most obvious use of embedded hypervisor to enhance several factors.

Costs reduction argument comes first. We can use an hypervisor to make currents operating system run seamlessly on new hardware components with only modifications to the hypervisor architecture dependent code. Depending on the hypervisor architecture, we can make the previous argument true when we not change all platform components, but only a device. In this case, we only need to write an hypervisor driver for the new device and all operating system running on top of the hypervisor will be able to use directly the new device functionality. Another argument which fits in the cost reduction category is the ability to run on the same system on chip two operating systems. One real time dedicated to radio communication and another general operating system for user interface (including user experience applications). Nowadays, the majority of mobile phone use two system on chip when the user experience part of the phone is slightly advanced.

The second argument concerns the software lifetime management. The software update management problem appeared with the success of smart phone which run full featured operating system. We face the same problems on these new devices as in "classical" operating system: software vulnerabilities, libraries dependency hell for developers and software updates among other. An hypervisor can reduce the management complexity for some of these tasks by providing operating system snapshots and roll-back features.

The third argument concerns the assets management in an over connected world where business is part of life. We think of employees who use the same mobile phone for unrelated context (e.g. business context and personal context). A company can use the hypervisor capabilities to protect their assets and enforce security policies.

The last argument is related to the three others: we can use hypervisor to built chains of trust at a lower level and increasing at the same time the chain strength.

B. Embedded hypervisor security checking

As we saw in the previous part, the embedded hypervisor is a promising technology and hot topic in the embedded world. Current implementations are still young and as a consequence they haven't been widely deployed and tested. But like all software, hypervisor can suffer from bugs.

Moreover, we think that embedded systems constraints can lead to reduce the number of security checks. This is even truer with micro-kernel target whose are mainly designed for more restrained feature phone.

C. Testing OKL4 micro-kernel

We chose to focus for now on OKL4. OKL4[1] from the OK-Labs company is one of the most complete solution at this time. It provide a micro-kernel derived from the L4 project capable of running guest operating system using the para-virtualization technique.
Our work is based on the open source OKL4 version 3.0 and OKLinux provided by OK-Labs. In the next sections of this paper, we talk only about this version[2].

D. Definition of syscall and why it is important in micro-kernel

Micro-kernel are designed to include only the strict minimum foundation necessary to build an operating system on op of it. They only include these three functionalities: a memory manager, a scheduler and an inter-process communication system. These are ideally, the only parts which run in the privileged processor mode. The other parts of the operating system run on top of the micro-kernel and call its functions when they need a it via a syscall. Each operating system defines a convention that should be used by programs to request an operating system service. Generally, developers don't use this convention directly, but instead use a wrapper library which abstract these calls.

Due to the minimal set of functionalities exposed by a micro-kernel, the number of syscall is also minimal compared to classical kernel. There are only fifteen syscalls in OKL4 compared to the almost 300 in Linux. But higher level API often expose more complex services which are the result of several combined syscalls. As the syscalls implementation are entirely part of trusted computing base, they should be carefully implemented.

In the next sections of this paper, we will take for all our examples the OKL4 IPC syscall. It is used to make two applications communicates. This syscall take four parameters:

- $Target_{IN}$ takes the identity of the receiver application (thread in OKL4 terminology),
- $Source_{IN}$ takes the identity of the current thread (the sender),
- $Acceptor_{IN}$ takes a value of {True, False, NotDefined} and is used to change the behavior of the operation,
- Tag_{IN} takes a value used to change the behavior of the operation.

In addition to these four parameters, the IPC syscall uses a set of another parameters $Data_{iIN}$ which contain the data of the message. The IPC syscall also return four values:

- $ReplyCap_{OUT}$ is a capability which should be used to reply to an incoming message,
- $SenderSpace_{OUT}$ store the address space in which sender application run,
- $ErrorCode_{OUT}$ store the status of the syscall,
- Tag_{OUT} store meta-data of the received message.

In addition to these values, the IPC syscall also store in the set $Data_{iOUT}$ the received message data.

All of these parameters are typed and mapped to registers which are either real (processor register) or virtual. Virtual Register are implemented as local thread variables stored in the user-level thread block.

III. Fuzzing

Fuzzing is a software testing technique based on the analysis of a software comportment when inputs are fed with particular data. It can be invalid or random data for example. There are two approach to fuzzing: mutation and generation.

A. Mutation based fuzzer are inappropriate for syscalls testing

Mutation based fuzzer are the simplest form of fuzzer. They take existing valid input and apply several transformation on them before fed it to the tested software.

This technique isn't appropriate in case of syscall testing. Firstly, it is hard to save syscall parameters and it's execution context. Secondly, the tested space is too vast and the repartition of mutated values are too homogeneous to be interesting when mutation fuzzer are used. Indeed, the probability to make a well formed syscall is very low. This approach is more useful for testing parser system like in network software.

B. Generation based fuzzer need a model which takes into account the data and state of tested program

Generation based fuzzer are smarter than mutation based fuzzer. They provide a set of tools to describe the models of inputs taken by the tested software. They generate new test data from these models.

The principal difficulty with this method is to write a model as accurate as possible. In the case of OKL4, the wide range of syscall parameters make this task particularly hard.

C. Peach

Peach[3] is a framework aimed at easing the construction process of fuzzer. It provides a language to model data and states of a software. It can then generate inputs automatically from these files and transformer script. Its main characteristic is the combination of the two fuzzing method in a single framework.

It works perfectly on a wide set of problems like network protocols testing or parser testing. But its grammar used to describe models is too limited to be useful in our context. Indeed, it is not possible to model low level OKL4 syscalls. Moreover, Peach doesn't allow us to use small model details with ease. But it is these details which reduce considerably the tested value space and thus reduce the testing time.

IV. How to provide a good model to maximize bugs finding : the syscalls model

Based on these fact, we have decided to make a new grammar to model a syscall, and a tool to make computation on them.

A. Syscall formalization for OKL4

In this part, we will explain how syscalls work in the OKL4 micro-kernel. In OKL4, a system call is a function taking one or more arguments and providing one or more results. There are 2 types of inputs and output arguments:

- standard parameters,
- virtual registers.

Standard parameters are like normal arguments of any function in a simple program and virtual registers are objects which are associated to each threads in the system. The user can interact with them with "getter" and "setter" functions. They can be mapped directly on processor registers and must be set before running a syscall.

Before running a syscall, some conditions have to be verified to ensure its success. OKL4 should be in a certain state described in the OKL4 micro-kernel reference manual[4].

After a syscall, some results are returned. There is always a *Result* parameter which indicates if the syscall is successful or not. If the *Result* is set to false, the *ErrorCode* parameter provides some information about the error. The different errors are described[4]. In case of a successful syscall the manual describes in which state OKL4 should be after this call.

B. Grammar

In order to enhance test generation, we have defined a grammar to model OKL4 syscalls. This grammar make possible to model precisely a syscall: its execution context, its parameters and their constraints and the results of the syscall.

We can then use the data stored in these models to make computation. These models have two goals. Firstly, they will be used as input for the space test generation algorithm. Secondly, they will be used to automatically configure the Peach fuzzer.

We have defined this as follow:

```
Syscall ::= 'Name' Name
            'Args' Args
            'InputRegister' InputRegisters
            'Output' Output
            'Success' Success
            'Failure' Failure
Name ::= 'CacheControl'|'Capcontrol'
        |'ExchangeRegisters'
        |'InterruptControl'
        |'Ipc'|'MapControl'
        |'memoryCopy'|'Mutex'
        |'MutexControl'|...
        |'ThreadSwitch'
Args ::= 'Name' ArgName
        'Type' ArgType
        'VirtualRegister' VirtualRegister
        'Constraint' Constraint | Args
ArgName ::= 'TargetIN'|'ControlIN'
           |'ClistIn'|'StackPointerIN'
           |'InstructionPointerIN'
           |'FlagsIN'|'UserHandleIN'
           |'SourceIN'|...
           |'DummyIN'|'PagerIN'
           |'ExceptionHandlerIN'
ArgType ::= 'SpaceID'|'CacheControl'
           |'Word'|'CacheRegionOp'
           |'Flag'|...
           |'ThreadState'|'SpaceControl'
           |'FPage'|'SpaceResources'
VirtualRegister ::= Parameteri|Resulti
                   |MessageDatai|'Acceptor'
Parameteri ::= 'Parameter0'|'Parameter1'
              |'Parameter2'|...
              |'Parameter6'|'Parameter7'
Resulti ::= 'Result0'|'Result1'
           |'Result2'|'Result3'|...
           |'Result6'|'Result7'
Messagedatai = 'MessageData0'|'MessageData1'
              |...
              |'MessageData30'|'MessageData31'
Constraint ::= ArgName.Field Operation Values
Field ::= ...
Operation ::= 'Equal'|'In'|'SubSet'
Values ::= ...
InputRegisters ::= 'Name' InputRegisterName
                   'Type' InputRegType
                   'VirtualRegister' VirtualRegister
                   'Constraint' Constraint
                   |InputRegisters
InputRegisterName ::= 'RegionAddressIN'|'RegionOPIN'
                     |Parameteri|'DataIN'
                     |'ArgumentIN'
InputRegType ::= 'Word'|'CacheRegionOp'
                |'CapParameter'|'IRQParamater'
                |'MapItem'
Output ::= 'Name' OutputName
           'Type' OutputType
           'VirtualRegister' VirtualRegister
OutputName ::= 'ResultOUT'|'ErrorCodeOUT'
              |'ControlOUT'|...
```

```
                |'ThreadHandleOUT'
Success ::= 'ErrorCode' SuccessErrorCode
           |'ResultOUT' SuccessResultOUT
            'ErrorCode' SuccessErrorCode
SuccessErrorCode ::= 'Undefined'
SuccessResultOUT ::= True
Failure ::= 'ErrorCode' FailureErrorCode
           |'ResultOUT' FailureResultOUT
            'ErrorCode' FailureErrorCode
FailureResultOUT ::= 'False'
FailureErrorCode ::= 'InvalidSpace'
                    |'InvalidParameter'|...
                    |'InvalidParam'
```

As this grammar is somewhat combersome, we have removed some part of it to reduce its size.

C. Example

The grammar as is, isn't very speaking. We chose to illustrate it with the model of the *Ipc* syscall:

```
Name Ipc
Args
  Name 'TargetIN'
  Type 'ThreadID'
  VirtualRegister 'Parameter0'
  Constraint 'TargetIN = ilThread'
  Name 'SourceIN'
  Type 'ThreadID'
  VirtualRegister 'Parameter1'
  Constraint
  Name 'AcceptorIN'
  Type 'Acceptor'
  VirtualRegister 'Acceptor'
  Constraint
InputRegisters
  Name'Data0IN'
  Type 'Word'
  VirtualRegister 'MessageData0'
  Constraint
Output
  Name 'ReplyCapOUT'
  Type 'ThreadID'
  VirtualRegister 'Result0'
  Constraint
  Name 'SenderSpaceOUT'
  Type 'SpaceID'
  VirtualRegister 'SenderSpace'
  Constraint
  Name 'ErrorCodeOUT'
  Type 'ErrorCode'
  VirtualRegister 'ErrorCode'
  Constraint
  Name 'TagOUT'
  Type 'MessageTag'
  VirtualRegister 'MessageData0'
  Constraint
  Name 'Data0OUT'
  Type 'Word'
  VirtualRegister 'MessageData0'
  Constraint
Success

Failure
```

V. Constraints and bounds calculation

As this is an on going works, our algorithms are not yet well defined. But we still expose here some thoughts about them.

A. How to use our syscall model to generate useful fuzzer outputs

From the above model, we can compute some properties. The set of properties allow the generation of each type valid syscalls. The set can not be used as fuzzing input directly because it is too large to be tested in a reasonable time.

But, we can use the *Constraint* property of the model in a manner to remove some testing values from this first set and therefore reduce it. Indeed, this property allows the elimination of all values which are not valid. We just need to keep some values which are out of these bounds to effectively test verification operated by the kernel at syscall invocation.

B. How to find interesting state in the tested system

The previous test set is intended to be fed directly to the fuzzer. But we can combine each syscall testing sets between them to make a new testing set containing syscall chain test. That is to say, each generated testing value is itself a sequence of several syscalls.

We think that this set can enhance the method strength and is more likely to stress the kernel and bring it to faulty states. Indeed, this testing set exposes a new fuzzing dimension because it enable the test of the kernel internal states.

VI. Interaction with Peach

When Peach is running, it generates all data needed and send them to OKL4 which interprets data, prepares the syscall and run it. There are three important objects in Peach:

- *DataModel*
- *StateModel*
- *Mutators*

These objects are written in XML file and provide a description of the fuzzed system and how to fuzz it.

A. Data model

DataModel in Peach provides a model of OKL4 syscalls. For each parameter syscall we specify:

- name
- type
- size
- default value

For example, we can take the *L4_CacheControl* syscall. This syscall takes 2 parameters "SpaceSpecifier" and "Control". In Peach, this syscall is represented as follow:

```
<DataModel name="L4_CacheControl">
<String name="L4_CacheControl" value="0" isStatic=
    "true"/>
<String value="#" isStatic="true"/>
<String name="SpaceSpecifier" size="32" value="0">
<Hint name="NumericalString" value="true"/>
</String>
<String value="#" isStatic="true"/>
<String name="control" size="32" value="0">
<Hint name="NumericalString" value="true"/>
</String>
```

```
<String value="##" isStatic="true"/>
</DataModel>
```

We use *String* data with a *NumericalString* hint to generate numbers because this object is more convenient to manipulate for OKL4. The attributes with a "#" value are used as separator for OKL4 and they aren't modify by the fuzzer (with the attribute "isStatic").

B. State model

StateModel describe the execution flow of the fuzzer. We indicate the different syscalls we want to fuzz and their order. We can specify different paths of execution depending on the return of each syscall. Our first approach of fuzzing don't use this *StateModel*. We preferred to fuzz only one syscall at a time but the definition of a good *StateModel* will be in our future works to improve intelligence of the fuzzer.

C. Mutators

Peach provide a lot of mutations for many types of data. For our work, we need only the numbers mutations. We have 3 types of mutations:

- Finite random numbers mutator which generates random numbers using a static seed. We can control the number of generated values,
- Numerical edge case mutator which produces values in a range for all numerical edge cases (signed byte, unsigned byte, signed short, unsigned long),
- and numerical variance mutator which produces all values in a range. this range is controllable.

We use all of these *Mutators* and especially the first one but in our future works, we want to extend Peach with another *Mutators* to decrease the numbers of tested values.

VII. Work evaluation

At this time, the evaluation of our work is limited due to its on going nature.

Our syscall grammar is now mostly in a stable state and it permits to model all of the fifteen OKL4 syscalls. Moreover, we are confident that this grammar will also permits with minimal changes to model syscall of others kernel.

We work currently on the refinement of the algorithms to takes into account the most cases as possible when dealing with the *Constraint* properties of the models.

VIII. Related work

We have found two software which can be used to test kernels and another software designed specifically for virtual machine monitor testing.

A. Sysfuzz

Sysfuzz is the first fuzzer targeting the syscalls. It is a trivial mutator fuzzer which feed with random data the syscall function of Unix like and Windows operating system. It has been designed for classical kernels where we find a single syscall function taking the syscall number as a parameter. Even if this software have found bugs in several kernels, it is not practical to use it in an embedded context.

B. Stress2

Stress2 is tools designed specifically for testing FreeBSD kernel sub-systems as virtual file system or virtual memory. Even if this tools is interesting, it is clearly not usable in the context of micro-kernel.

C. Kemufuzzer

Kemufuzzer is a tool implementing an interesting approach for a completely automated virtual machine monitor. It is based on a mutating fuzzer targeting virtual machine images coupled to an oracle. However, this design cannot be applied to an hypervisor like OKL4 where images are merged with the hypervisor at compile time.

IX. Future work

As we already stated it, this is an on going work. It remains a lot of work to make this method fully functional and automatic.

When this goal will be reached, we will work on the portability of method to make it usable on other micro-kernel and then we will try to apply it to more traditional virtual machine monitor like Xen.

X. Conclusion

We exposed in this paper a proposed enhancement to the fuzzing testing method. Currently, all fuzzing method are fairly dumb. We have suggested several ideas to remedy that. With a tool capable of reading models written with our grammar, it will generate a reduced set of testing sequence likely to generate bug in kernel syscall processing.

We are confident about the potential of this method. We designed it for OKL4 micro-kernel but we also tried to kept it relatively generic to be adaptable to other micro-kernel and hypervisor without much effort.

As stated this is an on going work. We don't have currently many results. However, we are confident about the potential of our work. In fact, it will permit to reduce the construction complexity of testing sets for Peach.

REFERENCES

[1] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence point of microkernels and hypervisors," in *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, Aug 2010, pp. 19–24.

[2] [Online]. Available: http://wiki.ok-labs.com/Microkernel

[3] [Online]. Available: http://peachfuzzer.com

[4] O. K. Labs, "Okl4 microkernel reference manual."