Enhancing fuzzing technique for OKL4 syscalls testing

Amaury Gauthier*, Clément Mazin*, Julien Iguchi-Cartigny* and Jean-Louis Lanet*

*Xlim — Smart Secure Devices
University of Limoges
France
{amaury.gauthier, clement.mazin, julien.cartigny, jean-louis.lanet}@xlim.fr

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

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: the system call API. Because all operating system components use directly or indirectly these system calls, a minor flaw at this level can impact in chain the entire system including a virtualized kernel. We have developed a model describing the OKL4 system calls. This model also contains all constraints applicable to a system call. Based on these models, we are working on a tool using the constraints to compute a reduced set of system call input values which are highly likely to generate flaws in OKL4 if they are not fully checked by the hypervisor.

Keywords—Virtual Machine, Embedded System, Kernel Security, OKL4, Syscalls

I. INTRODUCTION

Virtualization is an attractive technology which brings a lot of flexibility whatever the application domain is. Current server technologies bring a good trade-off between virtualization advantages and security (sharing a single hardware for multiple operating systems instances while keeping good isolation between them). Whereas the embedded virtual machine monitor technology is 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 monitors 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 the main arguments in favor of embedded virtualization in a mobile phone context. Then, we clarify the limitations of testing methods based on the fuzzing. We continue with the presentation of the grammar used by our tools. The fourth section is dedicated to the interaction with the fuzzer Peach[1] and the two last sections expose the related work and our future work.

II. MOTIVATION

Contrary to server context, embedded hypervisors are still in development and all their involvement on the current embedded applications are not yet known. We have chosen to expose some arguments in favor of 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 phones

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

Cost reduction argument comes first. We can use a hypervisor to make current operating systems 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 do not change all platform components, but only a device. In this case, we only need to write a hypervisor driver for the new device and all operating systems running on top of the hypervisor will be able to use directly the new device functionalities. 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 systems 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 phones which run full featured operating systems. We face the same problems on these new devices as in “classical” OSes: software vulnerabilities, libraries dependency hell for developers and software updates among other. A hypervisor can reduce

This research was funded by the TISPHANIE project.
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 and personal contexts). 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 hypervisors 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 young and as a consequence they have not been widely deployed and tested. But like all softwares, hypervisor may 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 targets whose are mainly designed for more restrained feature phones.

C. Testing the OKL4 micro-kernel

We chose to focus on OKL4[2] because when we began this work, this hypervisor was the only one publicly available. OKL4 from the OK-Labs Company is one of the most complete solution at this time. It provides a micro-kernel derived from the L4 project with the ability of running guest OSes using the paravirtualization technique.

Our work is based on the open source OKL4 version 3.0 and OKLinux provided by OK-Labs[3]. In the next sections of this paper, we will talk only about these versions.

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

Micro-kernels are designed to include only the strict minimum foundation necessary to build an operating system on top 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 it via a system call. Each operating system defines a conventions that should be used by programs to request an operating system service. Generally, developers do not use these conventions 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 system calls is also minimal compared to a conventional kernel. There are only fifteen system calls in OKL4 compared to the almost three hundred in Linux. But higher level APIs often expose more complex services which are the result of several combined system calls. As system calls are entirely part of trusted computing base, they should be carefully implemented.

In the next sections of this paper, we will focus on the OKL4 IPC system call. It is used when two applications communicates together. This system call takes four parameters:

- $\text{Target}_{IN}$ the identity of the receiver application (thread in OKL4 terminology).
- $\text{Source}_{IN}$ the identity of the current thread (the sender).
- $\text{Accept}_{IN}$ a value of $\{\text{True, False, NotDefined}\}$ and is used to change the operation behavior,
- $\text{Tag}_{IN}$ a value used to change the operation behavior.

In addition to these four parameters, the IPC system call uses a set of another parameters $\text{Data}_{IN}$ which contains the message data. The IPC system call also returns four values:

- $\text{ReplyCap}_{OUT}$ is a capability which should be used to reply to an incoming message,
- $\text{SenderSpace}_{OUT}$ stores the address space in which sender application run,
- $\text{ErrorCode}_{OUT}$ stores the status of the system call,
- $\text{Tag}_{OUT}$ stores meta-data of the received message.

In addition to these values, the IPC system call also stores in the set $\text{Data}_{OUT}$ the received message data.

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

As we know how to use OKL4 system calls now, we will see strategies to test them.

III. FUZZING

Fuzzing is a software testing technique based on the analysis of the software behavior when inputs are fed with particular data. For instance, it may be invalid or random data. There are two approaches to fuzzing: mutation and generation.

A. Mutation based fuzzer are inappropriate for system calls testing

Mutation based fuzzer are the simplest form of fuzzer. They take existing valid input and apply several transformations on them before to be send to the tested software.

This technique is not appropriate in case of system calls testing. Firstly, it is hard to save system call parameters and its execution context. Secondly, the tested space is too vast and the distribution of mutated values are too homogeneous to be interesting when mutation fuzzer are used. Indeed, the probability to obtain a well formed system call is very low. This approach is more interesting for testing parser system like in network softwares.
B. Generation based fuzzer need a model which takes into account the data and state of tested program

Generation based fuzzers are smarter than mutation based ones. 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[4].

The main difficulty with this method is to write the most accurate model as possible in order to avoid combinatorial explosion. In the case of OKL4, the wide range of system call parameters make this task particularly hard.

C. Peach

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

It works perfectly on a wide set of problems like network protocols or parser testings. 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 system calls. Moreover, Peach does not allow us to use precise model details with ease. But these details can reduce considerably the tested value space and thus reduce the testing time.

IV. HOW TO PROVIDE A GOOD MODEL TO MAXIMIZE BUGS FINDING : THE SYSTEM CALLS MODEL

Based on these facts, we have decided to develop a new grammar to model system calls, and a tool to reason on them.

A. System calls formalization for OKL4

In OKL4, a system call is a function taking one or more arguments and providing one or more results. There are two types of input and output arguments: standard parameters and 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 system call.

Some conditions must be verified before running a system call to ensure its success. The system should also be in a precise state as described in the OKL4 Micro-kernel Reference Manual[5].

After a system call, some results are returned. There is always a Result parameter which indicates if the system call 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 in [5]. In case of a successful system call, 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 system calls. This grammar allows to model precisely a system call: its execution context, parameters, constraints and the results of the system call.

Then, we can use the data stored in these models to perform 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:

<table>
<thead>
<tr>
<th>Listing 1. The system call grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syscall ::= ‘Name’ Name</td>
</tr>
<tr>
<td>Args Args</td>
</tr>
<tr>
<td>‘InputRegister’ InputRegisters</td>
</tr>
<tr>
<td>‘Output’ Output</td>
</tr>
<tr>
<td>‘Success’ Success</td>
</tr>
<tr>
<td>‘Failure’ Failure</td>
</tr>
<tr>
<td>Name ::= ‘CacheControl’</td>
</tr>
<tr>
<td>‘ExchangeRegisters’</td>
</tr>
<tr>
<td>‘InterruptControl’</td>
</tr>
<tr>
<td>‘Ipc’</td>
</tr>
<tr>
<td>‘MemoryCopy’</td>
</tr>
<tr>
<td>‘MutexControl’</td>
</tr>
<tr>
<td>‘ThreadSwitch’</td>
</tr>
<tr>
<td>ArgName ::= ‘Name’ ArgName</td>
</tr>
<tr>
<td>‘Type’ ArgType</td>
</tr>
<tr>
<td>‘VirtualRegister’ VirtualRegister</td>
</tr>
<tr>
<td>‘Constraint’ Constraint</td>
</tr>
<tr>
<td>ArgType ::= ‘SpaceID’</td>
</tr>
<tr>
<td>‘Word’</td>
</tr>
<tr>
<td>‘Flag’</td>
</tr>
<tr>
<td>‘ThreadState’</td>
</tr>
<tr>
<td>‘FPage’</td>
</tr>
<tr>
<td>VirtualRegister ::= Parameteri</td>
</tr>
<tr>
<td>MessageData</td>
</tr>
<tr>
<td>Parameteri ::= ‘Parameter0’</td>
</tr>
<tr>
<td>‘Parameter2’</td>
</tr>
<tr>
<td>‘Parameter6’</td>
</tr>
<tr>
<td>Resulti ::= ‘Result0’</td>
</tr>
<tr>
<td>‘Result2’</td>
</tr>
<tr>
<td>‘Result6’</td>
</tr>
<tr>
<td>MessageData = ‘MessageData0’</td>
</tr>
<tr>
<td>. . .</td>
</tr>
<tr>
<td>Constraint ::= ArgName. Field Operation Values</td>
</tr>
<tr>
<td>Field ::= . . .</td>
</tr>
<tr>
<td>Operation ::= ‘Equal’</td>
</tr>
<tr>
<td>Values ::= . . .</td>
</tr>
<tr>
<td>InputRegisters ::= ‘Name’ InputRegisterName</td>
</tr>
<tr>
<td>‘Type’ InputRegType</td>
</tr>
<tr>
<td>‘Constraint’ Constraint</td>
</tr>
<tr>
<td>InputRegisterName ::= ‘RegionAddressIN’</td>
</tr>
<tr>
<td>‘Parameteri’</td>
</tr>
<tr>
<td>‘ArgumentIN’</td>
</tr>
<tr>
<td>InputRegType ::= ‘Word’</td>
</tr>
<tr>
<td>‘CapParameter’</td>
</tr>
</tbody>
</table>
### C. Example

The grammar is not very meaningful. We chose to illustrate it with the model of the IPC system call:

#### Listing 2. Model for IPC system call.

```plaintext
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' 'ErrorCode' FailureErrorCode
FailureResultOUT ::= 'False'
FailureErrorCode ::= 'InvalidSpace' 'InvalidParameter' 'InvalidParam'
```

A constraint like TargetIN=NilThread => undefined means that if TargetIN value is NilThread, then the value of ReplyCapOUT will be undefined.

### V. CONSTRAINTS AND BOUNDS CALCULATION

As this is an ongoing works, our algorithms are not yet finished. But we still expose here some thoughts about them.

#### A. How to use our system call model to generate useful fuzzer outputs?

From the above model, we can compute some properties based on constraints. The set of properties allows the generation of each valid system calls. It can not be used as fuzzing input directly because the input domain is too large to be evaluated 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 verifications operated by the kernel at system call invocation.

The main algorithm of our method is defined in the next pseudo-code fragment. It takes as parameter a system call formalized with the grammar and the system state. This last argument enables the computation of constraints which uses the operating system state to restrict the set of values. The function returns a set of values to be tested on the system call in addition to the expected values.

#### Listing 3. Test values generation.

```plaintext
computeValues(Grammar syscall, SystemState state) :
  output =
    foreach syscall.Args, syscall.InputRegisters as arg{
      space = computeConstraint(arg, Constraint, SystemState state)
      foreach space as value{
        output[arg] = value
        output[arg] = lowerBound(value)
        output[arg] = upperBound(value)
      }
    }
  output[Success] =
```
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 system call testing sets between them to make a new testing set containing system call chain test. That is to say, each generated testing value is itself a sequence of several system calls.

This set enhances 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 variations of system calls and sends them to our OKL4 cell which interprets data, prepares the system call and run it. To describe our grammar in Peach, we use an XML file. This file includes three important objects: the DataModel, the StateModel and the Mutators. These objects provide a description of the fuzzed system and how to fuzz it.

A. Data model

DataModel in Peach provides a model of OKL4 system calls. For each system call parameter we specify: the name, the type, the size and default values. Each parameter is separated by a "neutral" element whose value is "#" or "##" to indicate the end of the DataModel. These elements must not be mutated by Peach so they have an attribute "isStatic" set to "true".

For example, we can take the L4_Ipc system call. It takes four parameters and one input register. In Peach, we represented it as follow:

Listing 4. Peach DataModel for IPC system call

```xml
<DataModel name="L4_Ipc">
  <String name="L4_Ipc" value="4" isStatic="true"/>
  <String value="#" isStatic="true"/>
  <String name="tag" size="32" value="0"/>
  <Hint name="NumericalString" value="true"/>
</String>
</DataModel>
```

We use String data with a NumericalString hint to generate numbers because this object is more convenient to manipulate for OKL4.

B. State model

StateModel describes the execution flow of the fuzzer. We indicate the different system calls we want to fuzz and their order. We can specify different execution path depending on the return of each system call. In our first approach of fuzzing, we used a simple StateModel: we fuzzed one system call at a time. We currently work on completing the definition of the StateModel (chaining and mixing several system calls) to improve intelligence of the fuzzer.

C. Mutators

Peach provides a lot of mutations for many types of data. For our work, we only use number mutators. We have three 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 short),
- 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 will extend Peach with another Mutators to decrease the numbers of tested values.

VII. RELATED WORK

We have found three softwares which can be used to test kernels. The last one is designed specifically for virtual machine monitor testing.

Sysfuzz is the first fuzzer targeting system calls. It is a trivial mutator fuzzer which feeds with random data the system call function of Unix like and Windows operating systems. It has been designed for classical kernels where we find a single system call function taking the system call number as a parameter. Even if this software have found bugs in several kernels, it is not practical to use it on a micro-kernel as the system call interface of these kernels differ greatly from a monolithic kernel.

Stress2 is a tool specifically designed 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 micro-kernel context because it was not designed to test this type of kernel.

*Kemufuzzer*[6] 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.

In addition to these tools, we have found these publications which inspired us[7], [8]. [7] is about testing the processor and system emulators Qemu[9], Valgrind[10], Pin[11] and BOCHS[12]. [8] describe how bugs have been found in the main hypervisors for desktop computers and servers: Qemu[9], BOCHS[12], VMware, Xen[13].

**VIII. FUTURE WORKS**

Currently, we have the OKL4 system calls formalized in models and a tool which is able to read them and generates values needed by Peach. The next step is the integration of the generator tool with the fuzzer to make the method automatic.

When this goal will be done, we will focus on enhancing the inputs generation process to test a sequence of system calls instead of testing them one by one.

We will also work on the portability of this method to make it usable with other micro-kernels and try to apply it to more traditional virtual machine monitors like Xen[13].

**IX. CONCLUSION**

We exposed in this paper an enhancement to the fuzzing testing method. Currently, all fuzzing methods are very basic. We have suggested several ideas to correct that. With a tool capable of reading models written with our grammar, it will generate a reduced set of testing sequences likely to generate bug in kernel system call processing.

This method was designed for OKL4 micro-kernel but we also tried to keep it relatively generic to be adaptable to other micro-kernel and hypervisor without much effort. It may also be useful to test more conventional monolithic kernels.

**REFERENCES**


