Towards a semantics-driven implementation of JITK, using K-framework

ZEGAOUI Taquyeddine

May 23, 2016

Table of contents

What do Bitcoin transaction scripting, network packet filtering and power management have in common ?

They all make use of in-kernel interpreters !

What do Bitcoin transaction scripting, network packet filtering and power management have in common ?
They all make use of in-kernel interpreters !

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space
- Any error or attack can have tremendous consequences

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space
- Any error or attack can have tremendous consequences

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space
- Any error or attack can have tremendous consequences

What can be done to guarantee safe and correct in-kernel interpreters ?

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space
- Any error or attack can have tremendous consequences

What can be done to guarantee safe and correct in-kernel interpreters ?

JITK: what is it ?

Enter JITK,

- An infrastructure for building in-kernel interpreters
- Guarantees complete functional correctness

JITK: what is it ?

Enter JITK,

- An infrastructure for building in-kernel interpreters
- Guarantees complete functional correctness
- Uses user-defined system call policies to secure execution

JITK: what is it ?

Enter JITK,

- An infrastructure for building in-kernel interpreters
- Guarantees complete functional correctness
- Uses user-defined system call policies to secure execution
- Those policies define what system call the executing code can invoke

JITK: what is it ?

Enter JITK,

- An infrastructure for building in-kernel interpreters
- Guarantees complete functional correctness
- Uses user-defined system call policies to secure execution
- Those policies define what system call the executing code can invoke

JITK: What does it do ?

The system call policies are

- Compiled to native code to be passed to the kernel
- Guaranteed to correctly translate into native code without loss of meaning

JITK: What does it do ?

The system call policies are

- Compiled to native code to be passed to the kernel
- Guaranteed to correctly translate into native code without loss of meaning
- Used to determine behavior regarding every sytem call

JITK: What does it do ?

The system call policies are

- Compiled to native code to be passed to the kernel
- Guaranteed to correctly translate into native code without loss of meaning
- Used to determine behavior regarding every sytem call
- Able to be checked by the interpreter without overhead cost

JITK: What does it do ?

The system call policies are

- Compiled to native code to be passed to the kernel
- Guaranteed to correctly translate into native code without loss of meaning
- Used to determine behavior regarding every sytem call
- Able to be checked by the interpreter without overhead cost

Introducing BPF

The BPF language

- Originally serves for defining packets filters
- Is a low-level language rather close to assembly

Introducing BPF

The BPF language

- Originally serves for defining packets filters
- Is a low-level language rather close to assembly
- Is used here as a system call filter

Introducing BPF

The BPF language

- Originally serves for defining packets filters
- Is a low-level language rather close to assembly
- Is used here as a system call filter

Example of BPF

```
; load syscall number
ld [0]
; deny open() with errno = EACCES
jeq #SYS_open, L1, L2
L1: ret #RET_ERRNO|#EACCES
; allow getpid()
L2: jeq #SYS_getpid, L3, L4
L3: ret #RET_ALLOW
; allow gettimeofday()
L4: jeq #SYS_gettimeofday, L5, L6
L5: ret #RET_ALLOW
L6: ...
; default: kill current process
ret #RET_KILL
```

As seen above, each system call gets an entry in the list of rules, along with the expected behavior regarding this particular sytem call. A default behavior is also defined, should any system call be absent from the previous list.

Example of BPF

```
; load syscall number
ld [0]
; deny open() with errno = EACCES
jeq #SYS_open, L1, L2
L1: ret #RET_ERRNO|#EACCES
; allow getpid()
L2: jeq #SYS_getpid, L3, L4
L3: ret #RET_ALLOW
; allow gettimeofday()
L4: jeq #SYS_gettimeofday, L5, L6
L5: ret #RET_ALLOW
L6: ...
; default: kill current process
ret #RET_KILL
```

As seen above, each system call gets an entry in the list of rules, along with the expected behavior regarding this particular sytem call. A default behavior is also defined, should any system call be absent from the previous list.

Introducing SCPL

We introduce SCPL, a domain specific language for defining sytem call policies

- In a more user-friendly way, being close to natural human language
- In an easier, more intuitive and less prone to errors manner

Introducing SCPL

We introduce SCPL, a domain specific language for defining sytem call policies

- In a more user-friendly way, being close to natural human language
- In an easier, more intuitive and less prone to errors manner
- Which reduces the risk of having incorrect BPF policies

Introducing SCPL

We introduce SCPL, a domain specific language for defining sytem call policies

- In a more user-friendly way, being close to natural human language
- In an easier, more intuitive and less prone to errors manner
- Which reduces the risk of having incorrect BPF policies
- Which will be translated to BPF

Introducing SCPL

We introduce SCPL, a domain specific language for defining sytem call policies

- In a more user-friendly way, being close to natural human language
- In an easier, more intuitive and less prone to errors manner
- Which reduces the risk of having incorrect BPF policies
- Which will be translated to BPF

Example of SCPL

```
{ default_action = Kill;
rules = [
{ action = Errno EACCES; syscall = SYS_open };
{ action = Allow; syscall = SYS_getpid };
{ action = Allow; syscall = SYS_gettimeofday };
...
] }
```

As seen above, SCPL is really close to the natural thought process of defining the rules of sytem call behavior, and this intuitive ease of use guarantees minimal errors within policies definition.

Example of SCPL

```
{ default_action = Kill;
rules = [
{ action = Errno EACCES; syscall = SYS_open };
{ action = Allow; syscall = SYS_getpid };
{ action = Allow; syscall = SYS_gettimeofday };
...
] }
```

As seen above, SCPL is really close to the natural thought process of defining the rules of sytem call behavior, and this intuitive ease of use guarantees minimal errors within policies definition.

Languages implementations

- Too many languages have only vague semantics defined in incomplete manuals
- Several implementations can exist for a single language

Languages implementations

- Too many languages have only vague semantics defined in incomplete manuals
- Several implementations can exist for a single language
- Some or even all implementations often rely on ad-hoc intuition of said language semantics

Languages implementations

- Too many languages have only vague semantics defined in incomplete manuals
- Several implementations can exist for a single language
- Some or even all implementations often rely on ad-hoc intuition of said language semantics
- Results in formally wrong implementations, with possibly terrrible results

Languages implementations

- Too many languages have only vague semantics defined in incomplete manuals
- Several implementations can exist for a single language
- Some or even all implementations often rely on ad-hoc intuition of said language semantics
- Results in formally wrong implementations, with possibly terrrible results

Introducing K-framework

Enter K-framework, a semantic framework

- Allowing implementations to be strict applications of syntax and semantics
- Guaranteeing clean, ambuigity-free implementations conform to specifications

Introducing K-framework

Enter K-framework, a semantic framework

- Allowing implementations to be strict applications of syntax and semantics
- Guaranteeing clean, ambuigity-free implementations conform to specifications
- Eliminating errors stemming from approximative implementations

Introducing K-framework

Enter K-framework, a semantic framework

- Allowing implementations to be strict applications of syntax and semantics
- Guaranteeing clean, ambuigity-free implementations conform to specifications
- Eliminating errors stemming from approximative implementations

Producing formally proven implementations

K-framework is able to generate languages implementations

- Which correctness is guaranteed by the use of Coq and OCaml
- Completely trustworthy and free of errors

Producing formally proven implementations

K-framework is able to generate languages implementations

- Which correctness is guaranteed by the use of Coq and OCaml
- Completely trustworthy and free of errors
- Which can be used to define compilers between defined languages

Producing formally proven implementations

K-framework is able to generate languages implementations

- Which correctness is guaranteed by the use of Coq and OCaml
- Completely trustworthy and free of errors
- Which can be used to define compilers between defined languages

Only by defining syntax and semantics

To generate such language implementations, we only need to define its syntax and semantics, split into three components

- Configurations: descriptions of the program state using nested cells
- Computations: sequences of computational tasks as nested lists

Only by defining syntax and semantics

To generate such language implementations, we only need to define its syntax and semantics, split into three components

- Configurations: descriptions of the program state using nested cells
- Computations: sequences of computational tasks as nested lists
- Rules are a generalisation of conventional rewrite rules

Only by defining syntax and semantics

To generate such language implementations, we only need to define its syntax and semantics, split into three components

- Configurations: descriptions of the program state using nested cells
- Computations: sequences of computational tasks as nested lists
- Rules are a generalisation of conventional rewrite rules

Ongoing progress

Ongoing : definition and implementation of SCPL and BPF using K-framework. Planned: definition and implementation of an SCPL to BPF compiler using K-framework.

Thank you for your attention ! Any questions ?

Further Reading I

- Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, Zachary Tatlock
  Jitk: A Trustworthy In-Kernel Interpreter Infrastructure.
  11th USENIX Symposium on Operating Systems Design and Implementation, 2014.

- Grigore Rosu, Traian Florin Serbanuta
  An Overview of the K Semantic Framework
  J.LAP, 2010.