# Internship Report

## A semantic-driven approach to system call filtering

ZEGAOUI Taquyeddine

FiT building, Tsinghua University, Beijing, China

September 1, 2016




# Table of contents

# Section 1

# Context of project

beginframe

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 ?

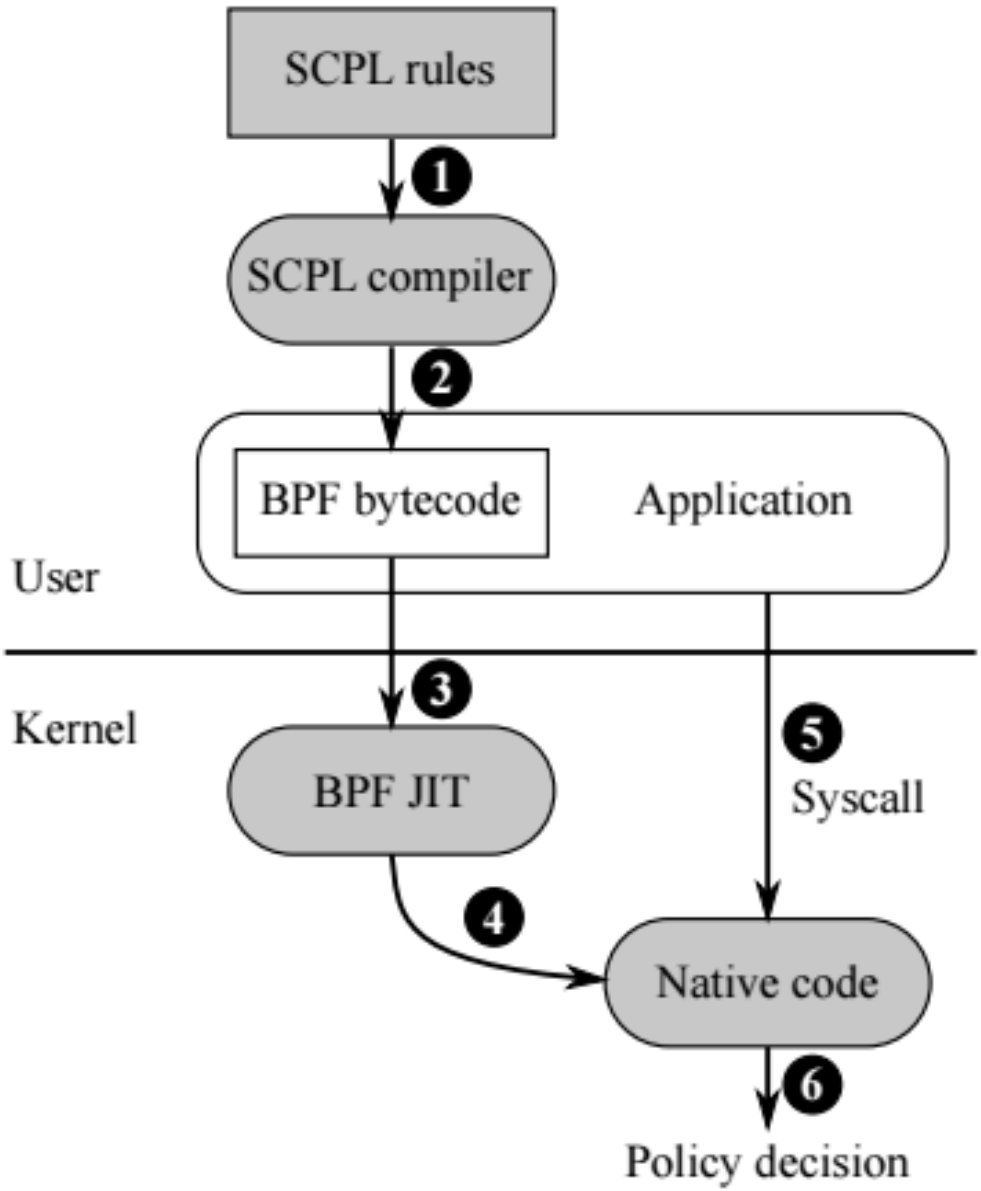
The system call policies are

- 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


SCPL rules
1
SCPL compiler
2
BPF bytecode
Application
User
Kernel
3
BPF JIT
5
Syscall
4
Native code
6
Policy decision

# 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

# Section 2

# Design Choices

# SCPL syntactic approach

Regarding the SCPL syntax

- DSL made for JITK
- Simple and close to thought process

# SCPL syntactic approach

Regarding the SCPL syntax

- DSL made for JITK
- Simple and close to thought process
- No formal language definition

# SCPL syntactic approach

Regarding the SCPL syntax

- DSL made for JITK
- Simple and close to thought process
- No formal language definition
- Ad-hoc definition used in JITK

# SCPL syntactic approach

Regarding the SCPL syntax

- DSL made for JITK
- Simple and close to thought process
- No formal language definition
- Ad-hoc definition used in JITK
- Clean redefinition of the syntax

# SCPL syntactic approach

Regarding the SCPL syntax

- DSL made for JITK
- Simple and close to thought process
- No formal language definition
- Ad-hoc definition used in JITK
- Clean redefinition of the syntax

# BPF syntactic approach

Regarding the BPF syntax

- Widely used general purpose language
- Specific use as call filter

# BPF syntactic approach

Regarding the BPF syntax

- Widely used general purpose language
- Specific use as call filter
- Numerous features left unused

# BPF syntactic approach

Regarding the BPF syntax

- Widely used general purpose language
- Specific use as call filter
- Numerous features left unused
- Redefinition of relevant used syntax

# BPF syntactic approach

Regarding the BPF syntax

- Widely used general purpose language
- Specific use as call filter
- Numerous features left unused
- Redefinition of relevant used syntax

# Comparative waterfall approach

Rewriting-based semantic definition

- User input is successively compared to each rule
- A rule with a different system call gets deleted

# Comparative waterfall approach

Rewriting-based semantic definition

- User input is successively compared to each rule
- A rule with a different system call gets deleted
- A matching rule deletes all other rules but itself

# Comparative waterfall approach

Rewriting-based semantic definition

- User input is successively compared to each rule
- A rule with a different system call gets deleted
- A matching rule deletes all other rules but itself
- Result is extracted from the last standing rule

# Comparative waterfall approach

Rewriting-based semantic definition

- User input is successively compared to each rule
- A rule with a different system call gets deleted
- A matching rule deletes all other rules but itself
- Result is extracted from the last standing rule

# Variable map approach

Environment-based semantic definition

- Each rule is first translated into a map variable
- Variable couple: SystemCall bound to ActionCode

# Variable map approach

Environment-based semantic definition

- Each rule is first translated into a map variable
- Variable couple: SystemCall bound to ActionCode
- Comparison is treated as a variable lookup

# Variable map approach

Environment-based semantic definition

- Each rule is first translated into a map variable
- Variable couple: SystemCall bound to ActionCode
- Comparison is treated as a variable lookup
- Result is the value of the matching name variable

# Variable map approach

Environment-based semantic definition

- Each rule is first translated into a map variable
- Variable couple: SystemCall bound to ActionCode
- Comparison is treated as a variable lookup
- Result is the value of the matching name variable

# Abstract assembly approach

Use of an abstract machine for BPF semantic

- Being an assembly language, BPF relies on machine structures
- Abstraction of those structures through the use of K own structures

# Abstract assembly approach

Use of an abstract machine for BPF semantic

- Being an assembly language, BPF relies on machine structures
- Abstraction of those structures through the use of K own structures
- Abstract configuration cells tasked with machine simulation

# Abstract assembly approach

Use of an abstract machine for BPF semantic

- Being an assembly language, BPF relies on machine structures
- Abstraction of those structures through the use of K own structures
- Abstract configuration cells tasked with machine simulation

# Section 3

# Results and evolutions

# Final features

Final features

- Full SCPL and BPF syntaxes, true to the new definitions
- Parsing of user-submitted SCPL or BPF scripts

# Final features

Final features

- Full SCPL and BPF syntaxes, true to the new definitions
- Parsing of user-submitted SCPL or BPF scripts
- Evaluation and splitting of scripts in computation cells

# Final features

Final features

- Full SCPL and BPF syntaxes, true to the new definitions
- Parsing of user-submitted SCPL or BPF scripts
- Evaluation and splitting of scripts in computation cells
- Waterfall and environment based comparisons in BPF

# Final features

Final features

- Full SCPL and BPF syntaxes, true to the new definitions
- Parsing of user-submitted SCPL or BPF scripts
- Evaluation and splitting of scripts in computation cells
- Waterfall and environment based comparisons in BPF

# Delayed features

Delayed features

- Full machine abstraction via configuration cells
- Proper syntactic type casting

# Delayed features

Delayed features

- Full machine abstraction via configuration cells
- Proper syntactic type casting
- Proper input and output behavior

# Delayed features

Delayed features

- Full machine abstraction via configuration cells
- Proper syntactic type casting
- Proper input and output behavior

# Possible evolutions

Possible evolutions

- Formal SCPL and BPF syntax and semantic definitions
- K-framework guide to make up for rare and rough online documentation

# Possible evolutions

Possible evolutions

- Formal SCPL and BPF syntax and semantic definitions
- K-framework guide to make up for rare and rough online documentation
- Proper logical proof research

# Possible evolutions

Possible evolutions

- Formal SCPL and BPF syntax and semantic definitions
- K-framework guide to make up for rare and rough online documentation
- Proper logical proof research
- SCPL to BPF compiler

# Possible evolutions

Possible evolutions

- Formal SCPL and BPF syntax and semantic definitions
- K-framework guide to make up for rare and rough online documentation
- Proper logical proof research
- SCPL to BPF compiler

# Section 4

# Discussion

# Advantages of K

- Lightweight, portable implementations
- Relative safety and correctness provided

# Advantages of K

- Lightweight, portable implementations
- Relative safety and correctness provided
- Deterministic and explicit behavior

# Advantages of K

- Lightweight, portable implementations
- Relative safety and correctness provided
- Deterministic and explicit behavior
- Relative ease and efficiency of implementation once used

# Advantages of K

- Lightweight, portable implementations
- Relative safety and correctness provided
- Deterministic and explicit behavior
- Relative ease and efficiency of implementation once used

# Drawbacks of K

Coq

- Rapidly changing project, esp parsers
- Learning curve for implementing languages

# Drawbacks of K

Coq

- Rapidly changing project, esp parsers
- Learning curve for implementing languages
- Lack of definite and up-to-date documentation and ressources

# Drawbacks of K

Coq

- Rapidly changing project, esp parsers
- Learning curve for implementing languages
- Lack of definite and up-to-date documentation and ressources
- Somewhat lacking safety and correctness compared to Coq

# Drawbacks of K

Coq

- Rapidly changing project, esp parsers
- Learning curve for implementing languages
- Lack of definite and up-to-date documentation and ressources
- Somewhat lacking safety and correctness compared to Coq

# Section 5

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.

Steven McCanne and Van Jacobson Lawrence Berkeley Laboratory
The BSD Packet Filter: A New Architecture for User-level Packet Capture
December 19, 1992

# Further Reading II

Mihail Asvoae, Computer Science Department, University "Al I Cuza" Iasi, Romania

K Semantics for Assembly Languages: A Case Study

Electronic Notes in Theoretical Computer Science 304 (2014) 111125