A quick look into verification in OS kernels

ZEGAOUI Taquyeddine

June 7, 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 ?

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient
- Each system call performs a single defined action

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient
- Each system call performs a single defined action
- All we need is a way to monitor and filter system calls

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient
- Each system call performs a single defined action
- All we need is a way to monitor and filter system calls

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient
- Each system call performs a single defined action
- All we need is a way to monitor and filter system calls

So what means do we have for system call filtering ?

A solution for in-kernel interpreters

One way of securing an in-kernel interpreter is limiting its powers to stop it from posing a security threat. System call filtering is a great way to do this:

- System calls are THE only way to interact with the system
- They are well known system routines
- Each system call is self sufficient
- Each system call performs a single defined action
- All we need is a way to monitor and filter system calls

So what means do we have for system call filtering ?

What is BPF ?

The BPF language

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

What is 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

What is 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

BPF syntax

```
; 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.

BPF syntax

```
; 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.

Inconvenients of BPF

But BPF is relatively complex to write, and can cause mistakes or errors that can lead to incorrect system call policies. We need an easy way to define BPF system call policies.

Inconvenients of BPF

But BPF is relatively complex to write, and can cause mistakes or errors that can lead to incorrect system call policies. We need an easy way to define BPF system call 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

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.

Use of BPF policies

The previously defined BPF system call policies are then:

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

Use of BPF policies

The previously defined BPF system call policies are then:

- 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

Use of BPF policies

The previously defined BPF system call policies are then:

- 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

Use of BPF policies

The previously defined BPF system call policies are then:

- 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 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

Correctness guarantees in JITK

Using Coq, JITK is able to guarantee correctness regarding:

- Implementation of each of its components
- User-space compilation of SCPL to BPF

Correctness guarantees in JITK

Using Coq, JITK is able to guarantee correctness regarding:

- Implementation of each of its components
- User-space compilation of SCPL to BPF
- Kernel-space compilation of BPF to native code

Correctness guarantees in JITK

Using Coq, JITK is able to guarantee correctness regarding:

- Implementation of each of its components
- User-space compilation of SCPL to BPF
- Kernel-space compilation of BPF to native code
- Application and enforcement of said policies

Correctness guarantees in JITK

Using Coq, JITK is able to guarantee correctness regarding:

- Implementation of each of its components
- User-space compilation of SCPL to BPF
- Kernel-space compilation of BPF to native code
- Application and enforcement of said policies

Introducing Coq

Coq

- A powerful proof assistant, written in Ocaml
- Able to help formally proving software

Introducing Coq

Coq

- A powerful proof assistant, written in Ocaml
- Able to help formally proving software
- Uses a form of mathematical language, called Gallina

Introducing Coq

Coq

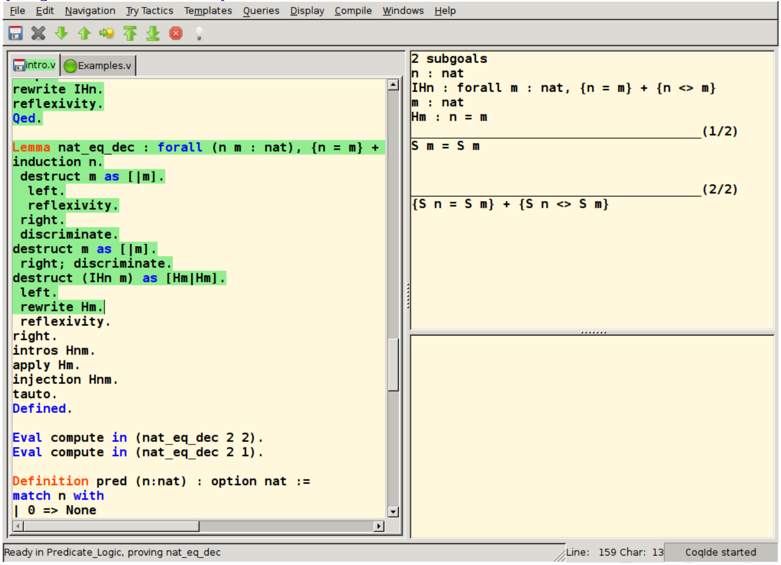
- A powerful proof assistant, written in Ocaml
- Able to help formally proving software
- Uses a form of mathematical language, called Gallina
- Was used to implement CompCert, a certified C2native compiler

Introducing Coq

Coq

- A powerful proof assistant, written in Ocaml
- Able to help formally proving software
- Uses a form of mathematical language, called Gallina
- Was used to implement CompCert, a certified C2native compiler

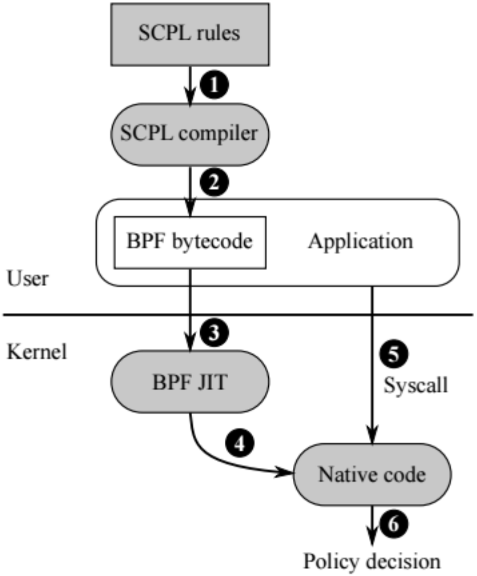
Coq syntax example


File Edit Navigation Try Tactics Templates Queries Display Compile Windows Help
intro.v
Examples.v

```
rewrite IHn.
reflexivity.
Qed.

Lemma nat_eq_dec : forall (n m : nat), {n = m} +
induction n.
 destruct m as [|m].
  left.
  reflexivity.
 right.
 discriminate.
destruct m as [|m].
 right; discriminate.
destruct (IHn m) as [Hm|Hm].
 left.
 rewrite Hm.
 reflexivity.
right.
intros Hnm.
apply Hm.
injection Hnm.
tauto.
Defined.

Eval compute in (nat_eq_dec 2 2).
Eval compute in (nat_eq_dec 2 1).

Definition pred (n:nat) : option nat :=
match n with
| 0 => None
```

```
2 subgoals
n : nat
IHn : forall m : nat, {n = m} + {n <> m}
m : nat
Hm : n = m
______________________________________(1/2)
S m = S m
______________________________________(2/2)
{S n = S m} + {S n <> S m}
```

```
Ready in Predicate_Logic, proving nat_eq_dec    Line:  159 Char:  13    CoqIde started
```


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

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space
- Decoding and checking to get BPF instructions

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space
- Decoding and checking to get BPF instructions
- Translation to Cminor

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space
- Decoding and checking to get BPF instructions
- Translation to Cminor
- Compilation to native assembly using CompCert

JITK operation steps

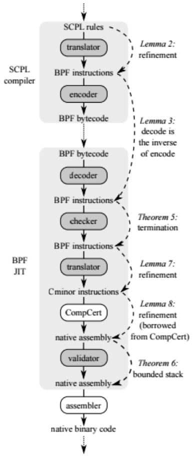
JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space
- Decoding and checking to get BPF instructions
- Translation to Cminor
- Compilation to native assembly using CompCert
- Validation and assembly to get native binary code

JITK operation steps

JITK operation is divided in several steps:

- User definition of SCPL rules
- Compilation of SCPL rules to BPF policies
- Encoding of BPF policies to BPF bytecode
- Transmission of BPF bytecode to kernel space
- Decoding and checking to get BPF instructions
- Translation to Cminor
- Compilation to native assembly using CompCert
- Validation and assembly to get native binary code


SCPL rules
translator
BPF instructions
encoder
BPF bytecode
SCPL compiler
Lemma 2: refinement
Lemma 3: decode is the inverse of encode
BPF bytecode
decoder
BPF instructions
checker
BPF instructions
translator
Cminor instructions
CompCert
native assembly
validator
native assembly
assembler
native binary code
BPF JIT
Theorem 5: termination
Lemma 7: refinement
Lemma 8: refinement (borrowed from CompCert)
Theorem 6: bounded stack

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.