Extensible Virtual Call Integrity

Yuancheng Jiang, Gregory J. Duck, Roland H.C. Yap,
Zhenkai Liang, Pinghai Yuan

ESORICS 2022

NUS
National University
of Singapore

C++ Dynamic Dispatch

The process of selecting which implementation of a polymorphic operation (method or function) to call at run time

C++ Dynamic Dispatch

The process of selecting which implementation of a polymorphic operation (method or function) to call at run time

C++ Dynamic Dispatch

The process of selecting which implementation of a polymorphic operation (method or function) to call at run time

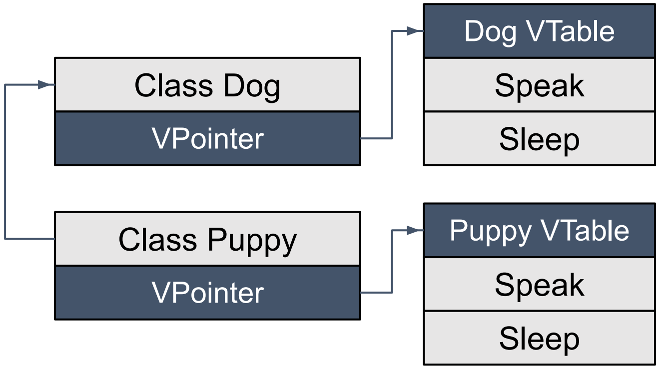
Supported Data Structure: Virtual Table/ Virtual Pointer

C++ Dynamic Dispatch

The process of selecting which implementation of a polymorphic operation (method or function) to call at run time

Supported Data Structure: Virtual Table/ Virtual Pointer

- VPointer: the pointer points to VTable

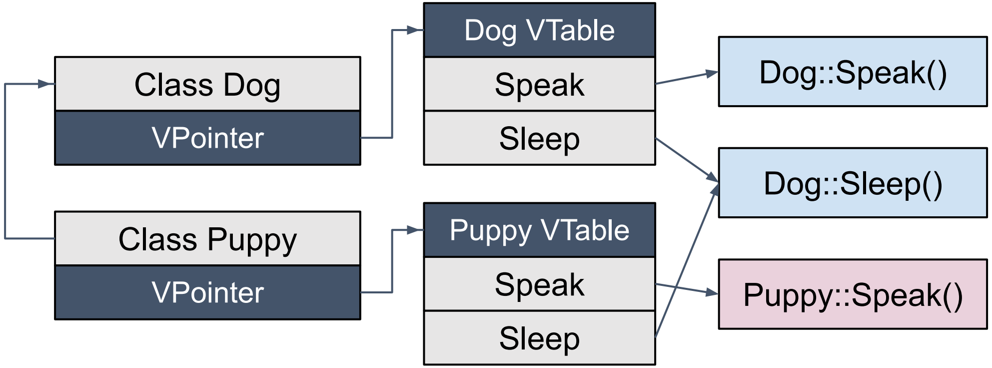


C++ Dynamic Dispatch

The process of selecting which implementation of a polymorphic operation (method or function) to call at run time

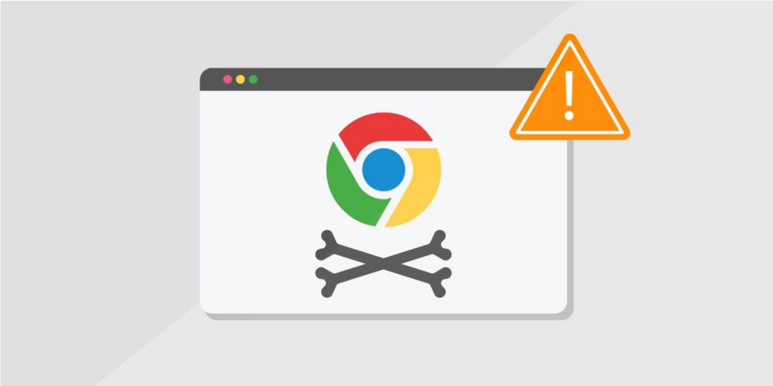
Supported Data Structure: Virtual Table/ Virtual Pointer

- VPointer: the pointer points to VTable
- VTable: the mapping from name to method



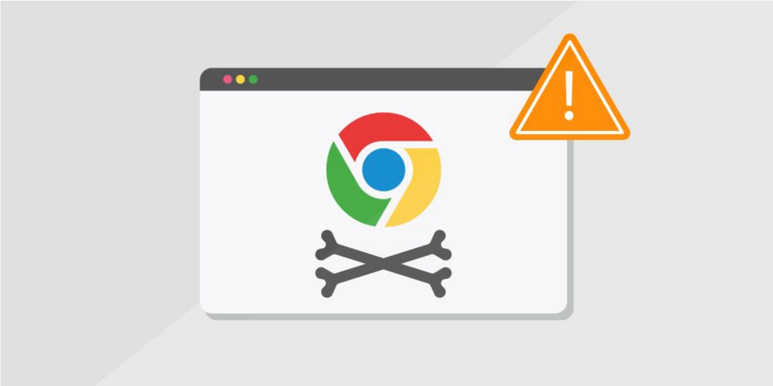
Is Virtual Call Secure?

Memory errors widely exist in all kinds of applications



Is Virtual Call Secure?

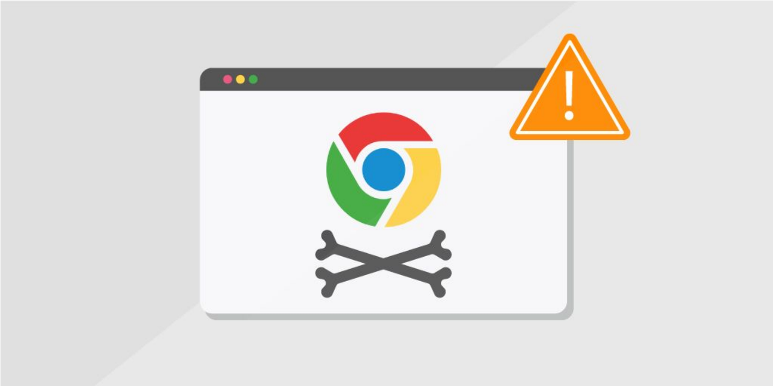
Memory errors widely exist in all kinds of applications



Are virtual tables/pointers safe against memory errors?

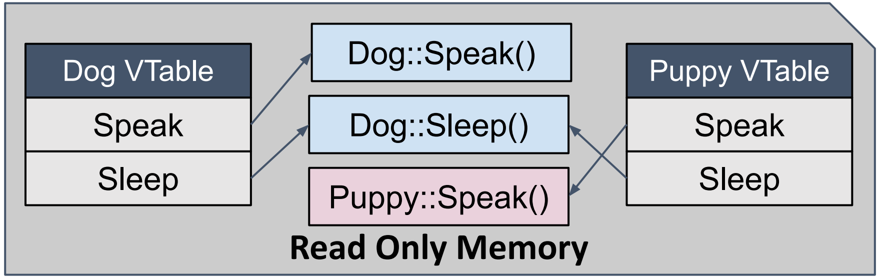
Is Virtual Call Secure?

Memory errors widely exist in all kinds of applications



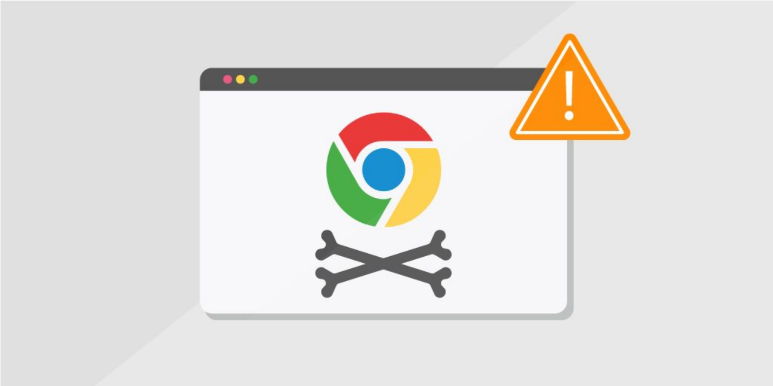
Are virtual tables/pointers safe against memory errors?

☺ Virtual Table: Read Only



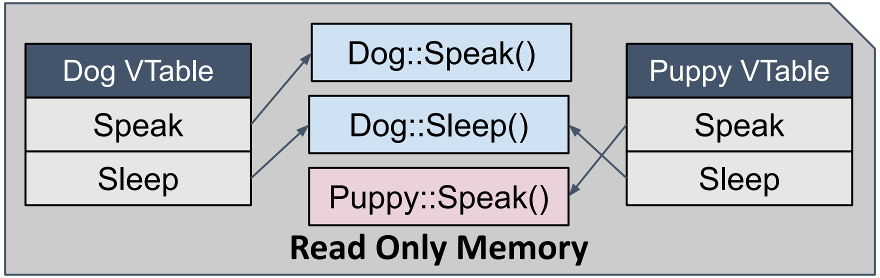
Is Virtual Call Secure?

Memory errors widely exist in all kinds of applications

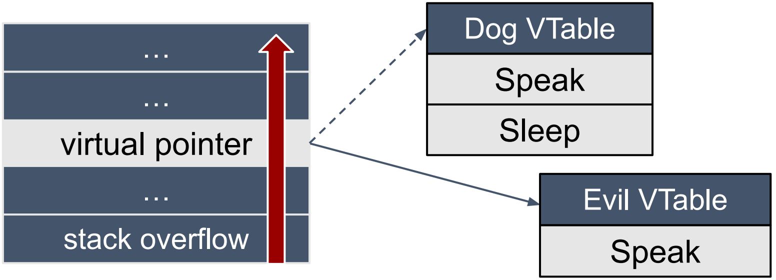


Are virtual tables/pointers safe against memory errors?

☺ Virtual Table: Read Only

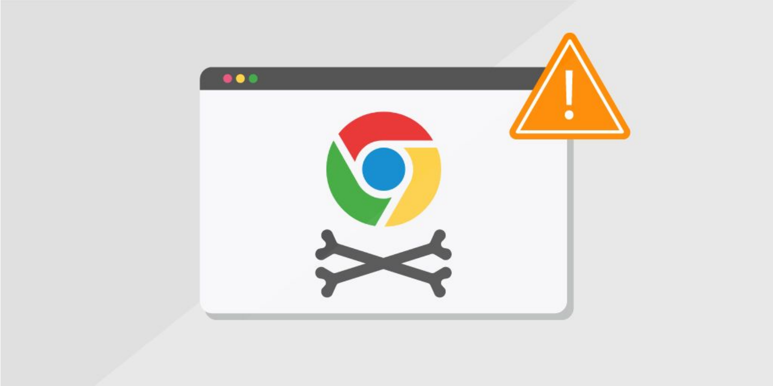


☹ Virtual Pointer: Stack/Heap



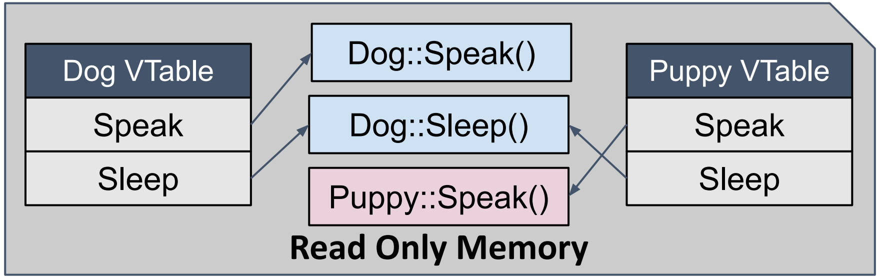
Is Virtual Call Secure?

Memory errors widely exist in all kinds of applications

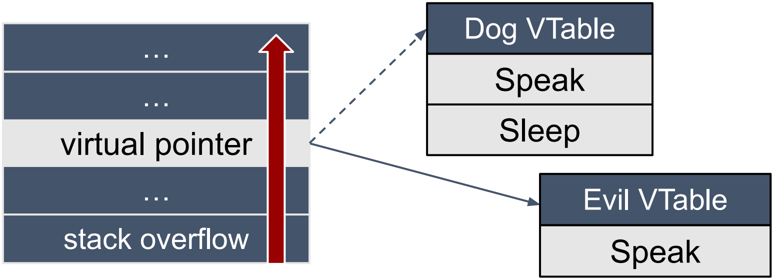


Are virtual tables/pointers safe against memory errors?

☺ Virtual Table: Read Only



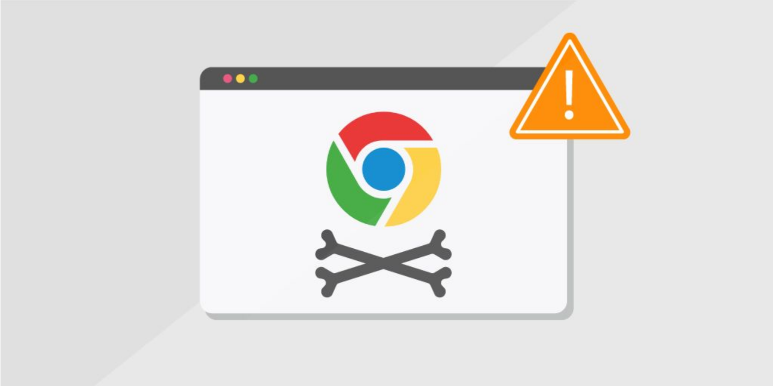
☹ Virtual Pointer: Stack/Heap



Memory Error => VCall Hijiack ✓

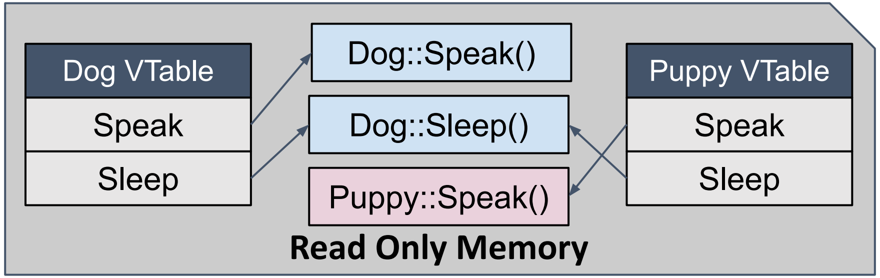
Is Virtual Call Secure?

Memory errors widely exist in all kinds of applications

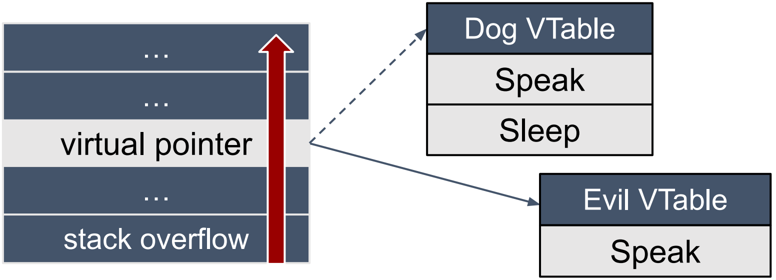


Are virtual tables/pointers safe against memory errors?

☺ Virtual Table: Read Only



☹ Virtual Pointer: Stack/Heap



Memory Error => VCall Hijiack ✓

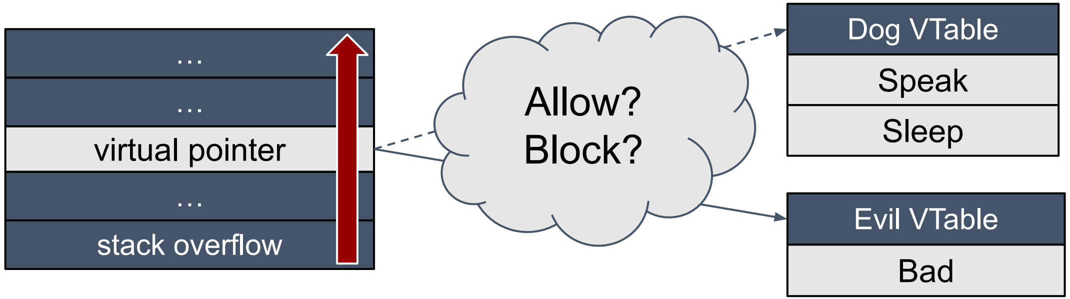
Research Solution: Virtual Control Flow Integrity(VCFI)

Virtual Control Flow Integrity (VCFI)

The goal of VCFI: check the validity of each virtual call's target

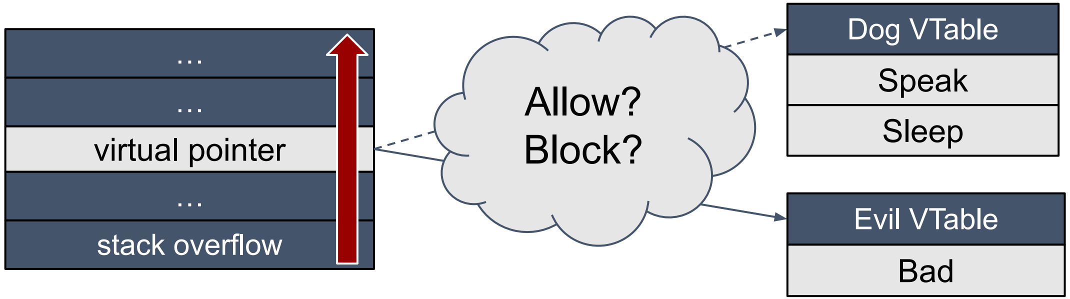
Virtual Control Flow Integrity (VCFI)

The goal of VCFI: check the validity of each virtual call's target



Virtual Control Flow Integrity (VCFI)

The goal of VCFI: check the validity of each virtual call’s target

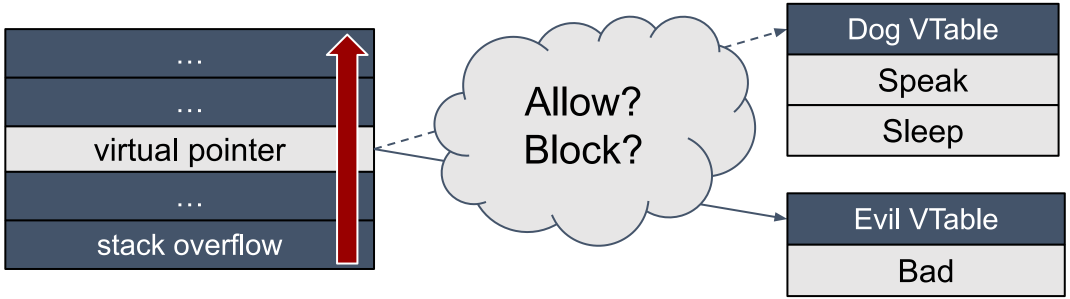


The Virtual Control Flow Integrity can be defined as:

$$c.\texttt{vcall}(\ldots); \qquad c.\texttt{vptr} \in Allow_C?$$

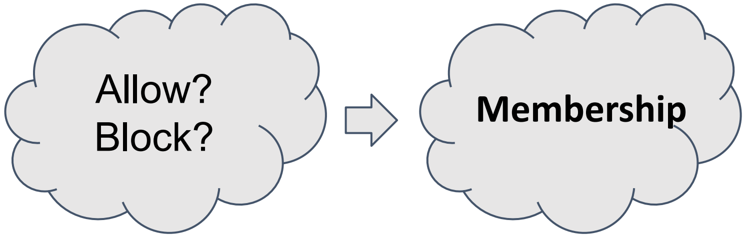
Virtual Control Flow Integrity (VCFI)

The goal of VCFI: check the validity of each virtual call's target



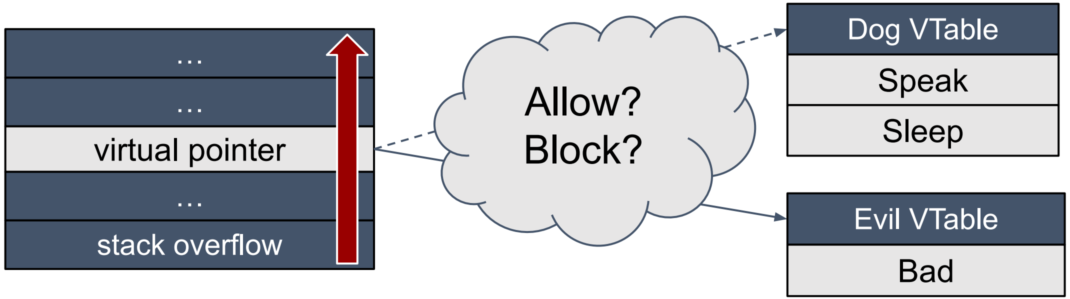
The Virtual Control Flow Integrity can be defined as:

$$c.\texttt{vcall}(\ldots); \qquad c.\texttt{vptr} \in Allow_C?$$



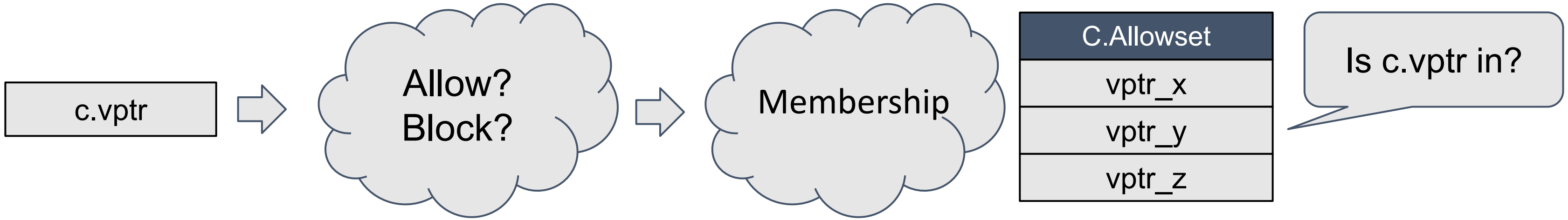
Virtual Control Flow Integrity (VCFI)

The goal of VCFI: check the validity of each virtual call's target



The Virtual Control Flow Integrity can be defined as:

$$c.\texttt{vcall}(\ldots); \qquad c.\texttt{vptr} \in Allow_C?$$



Practical VCFI

Practical VCFI

VCFI is a multi-trading defense:

Accurate, efficient, extensible, secure, etc.

Practical VCFI

VCFI is a multi-trading defense:

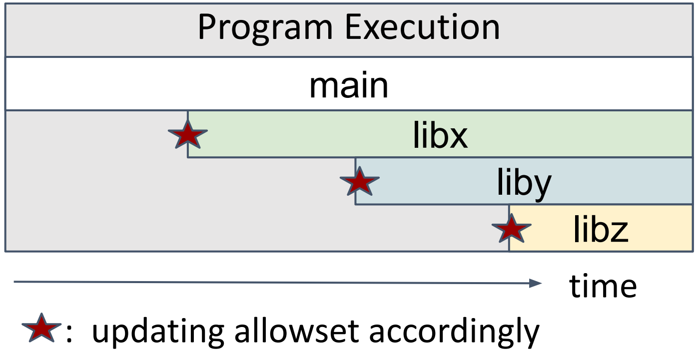
Accurate, efficient, **extensible**, secure, etc.

Extensibility becomes the main challenge for practical VCFI:

Practical VCFI

VCFI is a multi-trading defense:
Accurate, efficient, **extensible**, secure, etc.

Extensibility becomes the main challenge for practical VCFI:



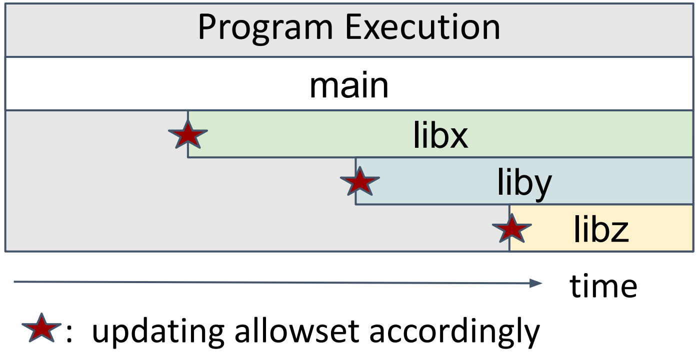
Dynamic Linking/Loading

Practical VCFI

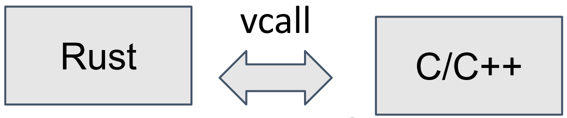
VCFI is a multi-trading defense:

Accurate, efficient, **extensible**, secure, etc.

Extensibility becomes the main challenge for practical VCFI:



Dynamic Linking/Loading

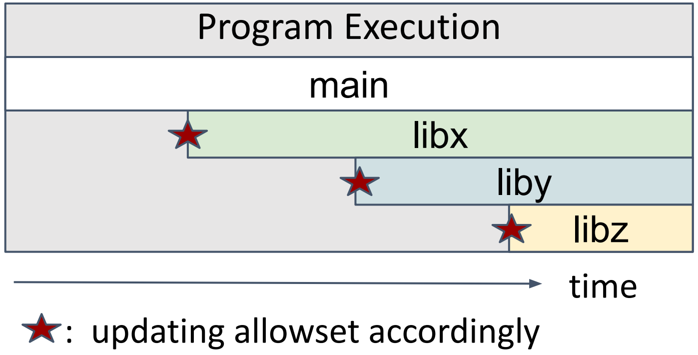


Inter-language Compatibility

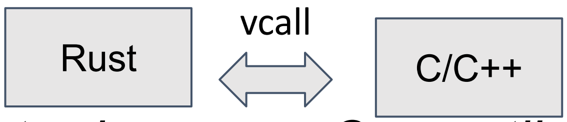
Practical VCFI

VCFI is a multi-trading defense:
Accurate, efficient, **extensible**, secure, etc.

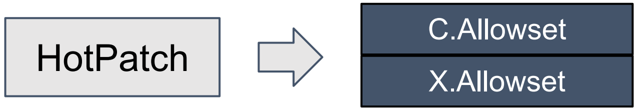
Extensibility becomes the main challenge for practical VCFI:



Dynamic Linking/Loading



Inter-language Compatibility

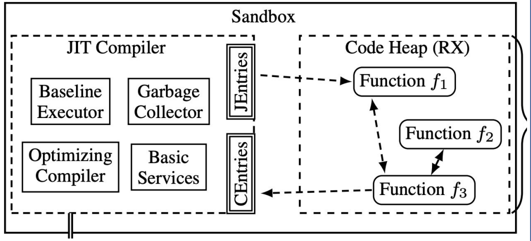


Ad-hoc Extension

Existing Works

Existing Works

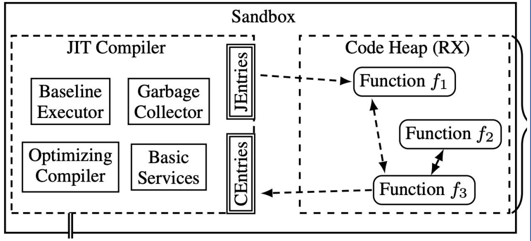
MCFI&RockJIT



* Sandbox
* Type-based policy
* High overhead

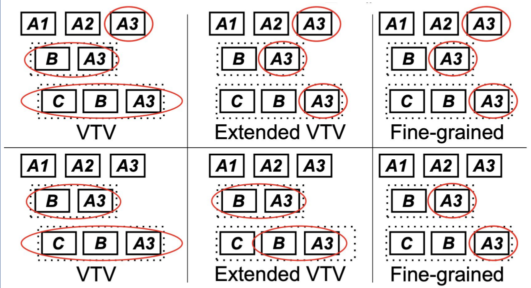
Existing Works

MCFI&RockJIT



* Sandbox
* Type-based policy
* High overhead

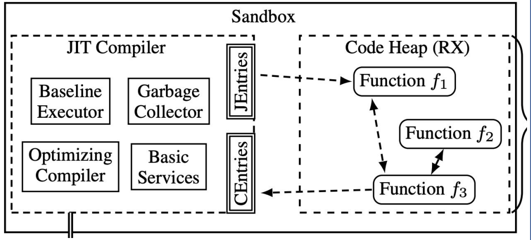
VTV&Shrinkwrap



* Unsatisfying policy
* High overhead

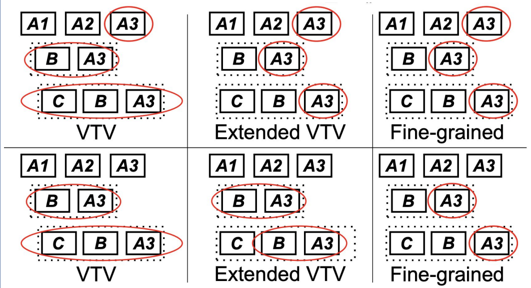
Existing Works

MCFI&RockJIT



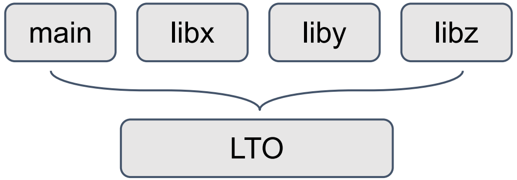
* Sandbox
* Type-based policy
* High overhead

VTV&Shrinkwrap



* Unsatisfying policy
* High overhead

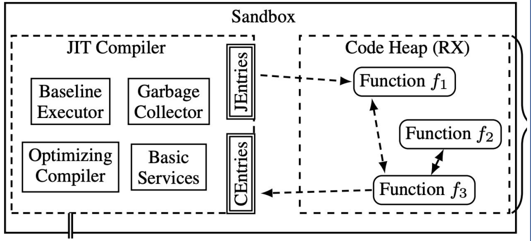
LLVM-VCFI



* LTO required
* Extensible when cross-dso but slow

Existing Works

MCFI&RockJIT

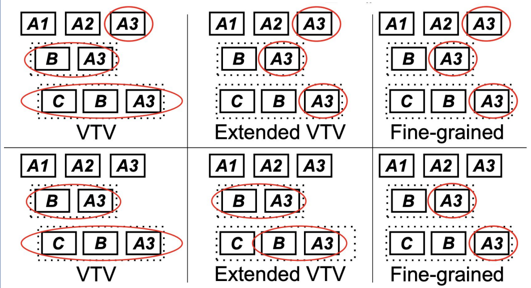


* Sandbox

* Type-based policy

* High overhead

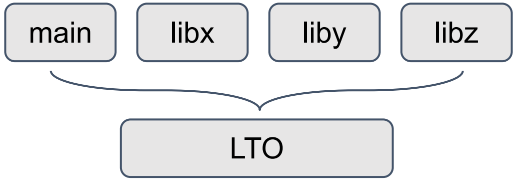
VTV&Shrinkwrap



* Unsatisfying policy

* High overhead

LLVM-VCFI



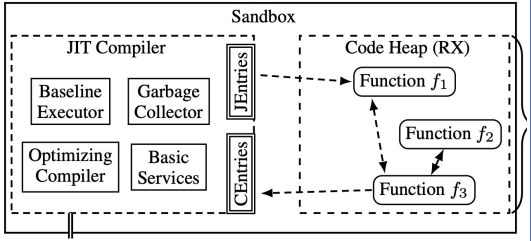
* LTO required

* Extensible when cross-dso but slow

☹ Existing works: either inefficient or limited in extensibility

Existing Works

MCFI&RockJIT

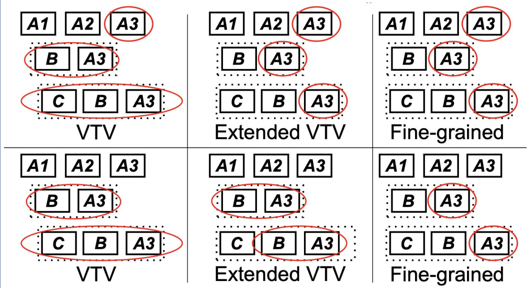


* Sandbox

* Type-based policy

* High overhead

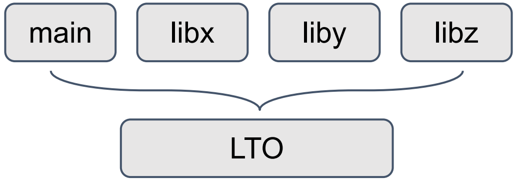
VTV&Shrinkwrap



* Unsatisfying policy

* High overhead

LLVM-VCFI



* LTO required

* Extensible when cross-dso but slow

☹ Existing works: either inefficient or limited in extensibility

☹ None of them are widely deployed in real applications

Intuition

- VCFI inherently is the membership problem
  - checking whether c.vptr is in c.allowset

Intuition

- VCFI inherently is the membership problem
  - checking whether c.vptr is in c.allowset
- Lookup table is naive and effective, but
  - now known as limited in extensibility

Intuition

- VCFI inherently is the membership problem
  - checking whether c.vptr is in c.allowset
- Lookup table is naive and effective, but
  - now known as limited in extensibility
- Any good data structure suits this task?

Intuition

- VCFI inherently is the membership problem
  - checking whether c.vptr is in c.allowset
- Lookup table is naive and effective, but
  - now known as limited in extensibility
- Any good data structure suits this task?

Insight: Approximate Membership Query Filter

Approximate Membership Query Filter (AMQ-Filter) is the space-efficient probabilistic data structure that supports approximate membership queries.

Intuition

- VCFI inherently is the membership problem
  - checking whether c.vptr is in c.allowset
- Lookup table is naive and effective, but
  - now known as limited in extensibility
- Any good data structure suits this task?

⭐ **Insight: Approximate Membership Query Filter**

Approximate Membership Query Filter (AMQ-Filter) is the space-efficient probabilistic data structure that supports approximate membership queries.

Then, Why not use AMQ-Filter to implement VCFI?

Bloom Filter

- Bloom Filter is the most well-known AMQ Filter

Bloom Filter

- Bloom Filter is the most well-known AMQ Filter
  - *O(1)* time membership checking

Bloom Filter

- Bloom Filter is the most well-known AMQ Filter
  - *O(1)* time membership checking
  - low storage requirement

Bloom Filter

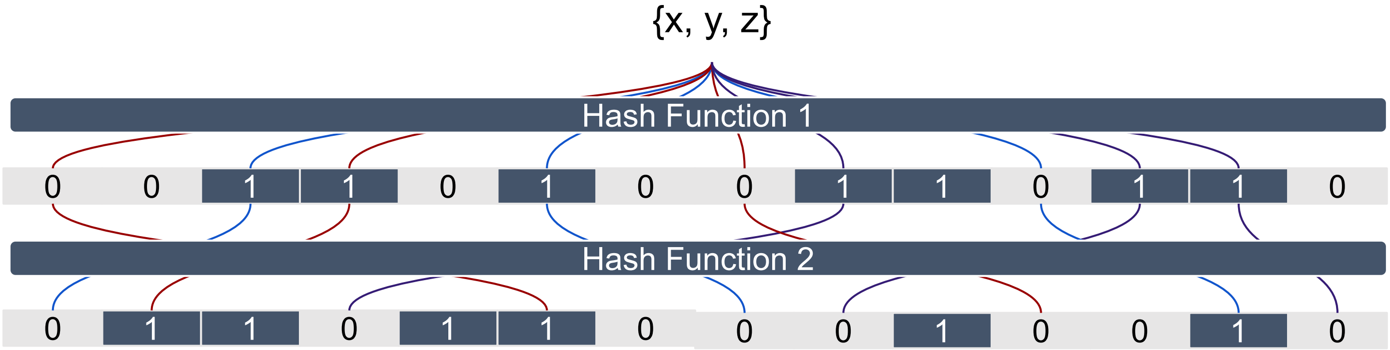
- Bloom Filter is the most well-known AMQ Filter
  - *O(1)* time membership checking
  - low storage requirement
  - no false negatives

Bloom Filter

- Bloom Filter is the most well-known AMQ Filter
  - *O(1)* time membership checking
  - low storage requirement
  - no false negatives
  - controlled and low false positive probability

Bloom Filter

- Bloom Filter is the most well-known AMQ Filter
  - *O(1)* time membership checking
  - low storage requirement
  - no false negatives
  - controlled and low false positive probability

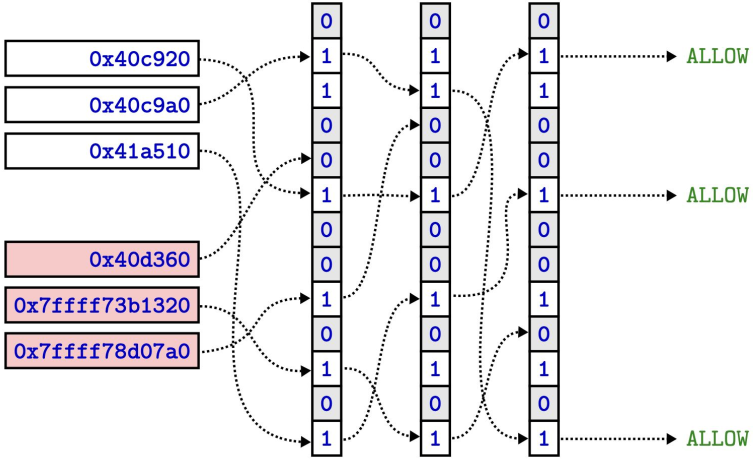


Extensible VCFI Enforcement

- VCFI Bloom Filter

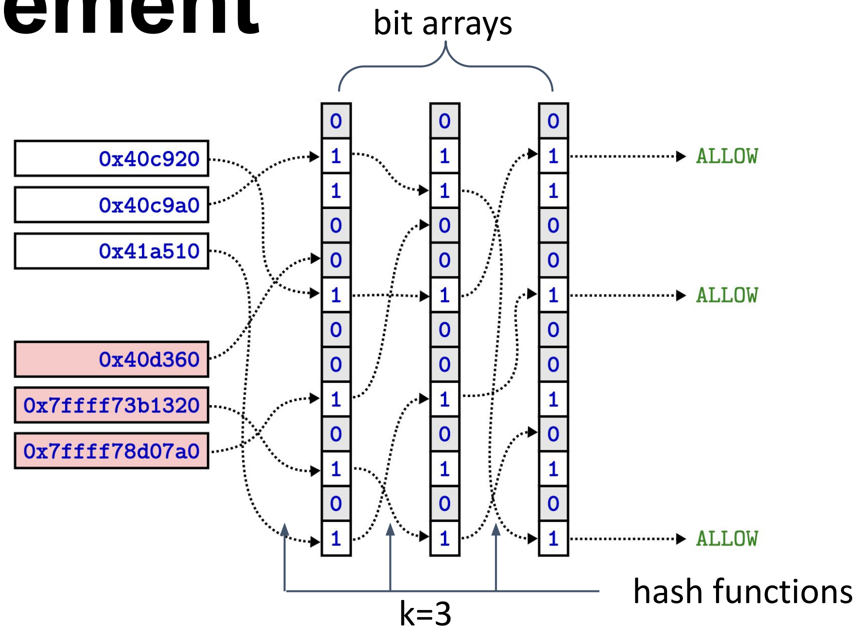
Extensible VCFI Enforcement

- VCFI Bloom Filter:
  - input: virtual pointer
  - output: allow or block



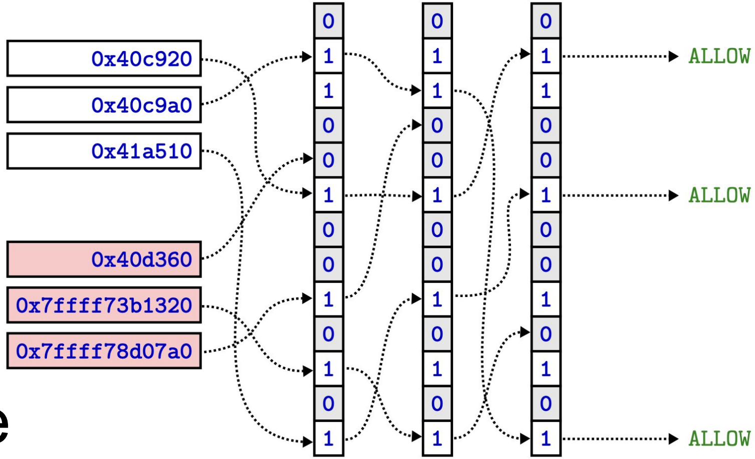
Extensible VCFI Enforcement

- VCFI Bloom Filter:
  - input: virtual pointer
  - output: allow or block
  - k: the number of hashes
  - bit array: result of hashes



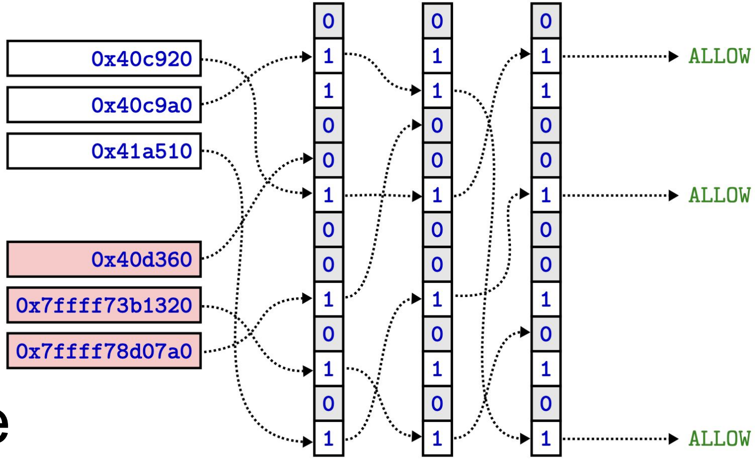
Extensible VCFI Enforcement

- VCFI Bloom Filter:
  - input: virtual pointer
  - output: allow or block
  - k: the number of hashes
  - bit array: result of hashes
  - feature: efficient, extensible



Extensible VCFI Enforcement

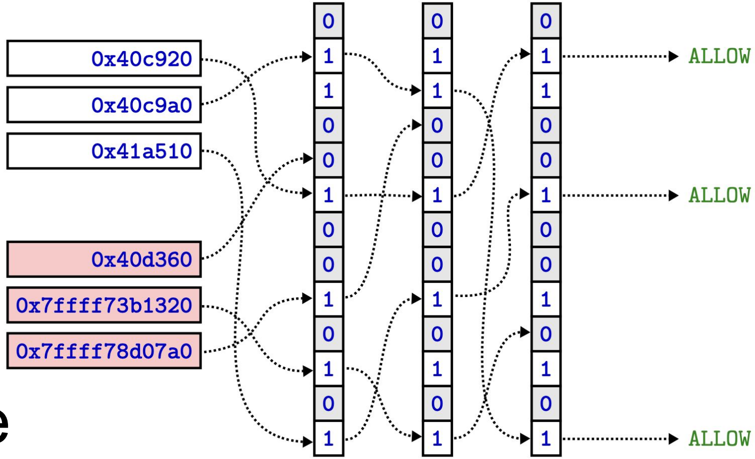
- VCFI Bloom Filter:
  - input: virtual pointer
  - output: allow or block
  - k: the number of hashes
  - bit array: result of hashes
  - feature: efficient, extensible



Membership Checking Policies: $B[hash_1(x)] \neq 0 \wedge \cdots \wedge B[hash_k(x)] \neq 0$

Extensible VCFI Enforcement

- VCFI Bloom Filter:
  - input: virtual pointer
  - output: allow or block
  - k: the number of hashes
  - bit array: result of hashes
  - feature: efficient, extensible

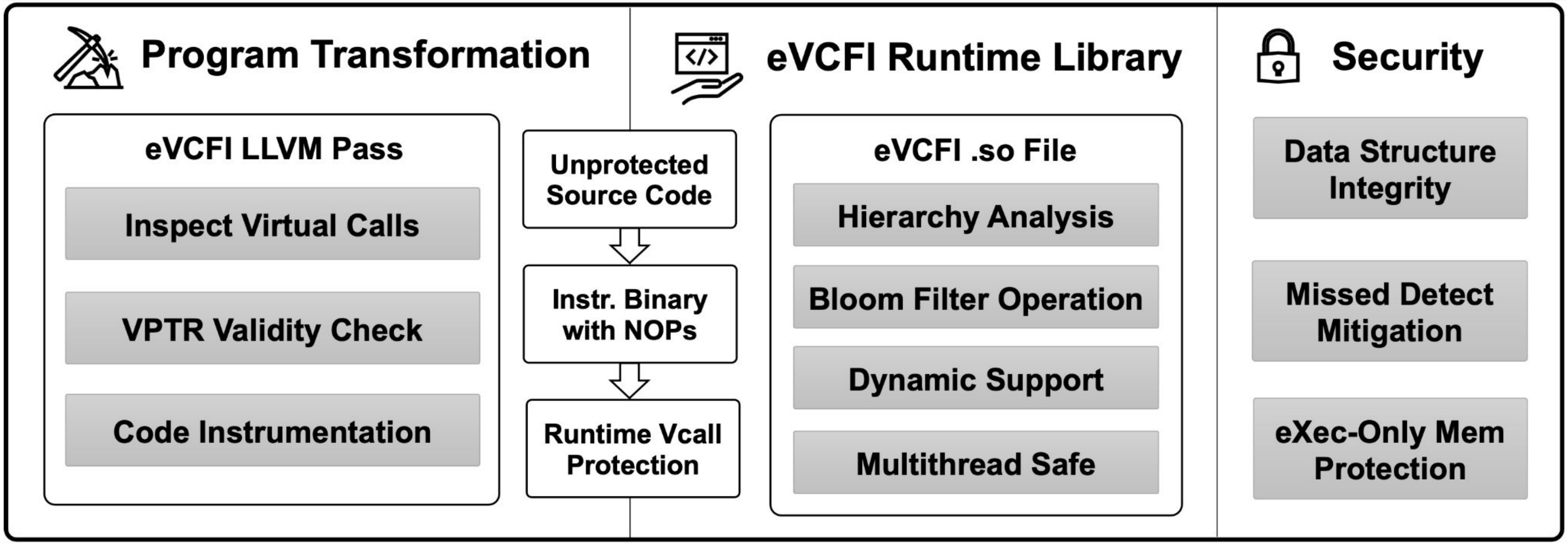


Membership Checking Policies: $B[hash_1(x)] \neq 0 \land \cdots \land B[hash_k(x)] \neq 0$

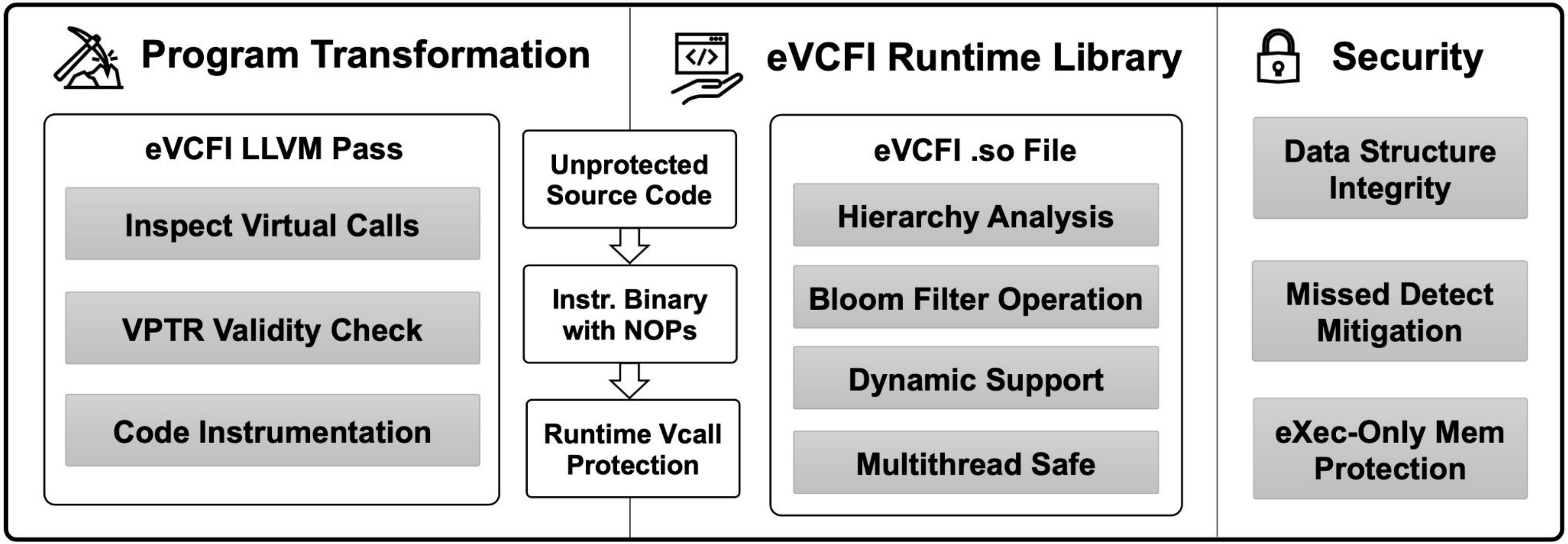
Only { all `1` hits in 3 tests } indicates the Valid VCall.

{ any `0` miss in 3 tests } indicates the Invalid VCall.

EVCFI Overview

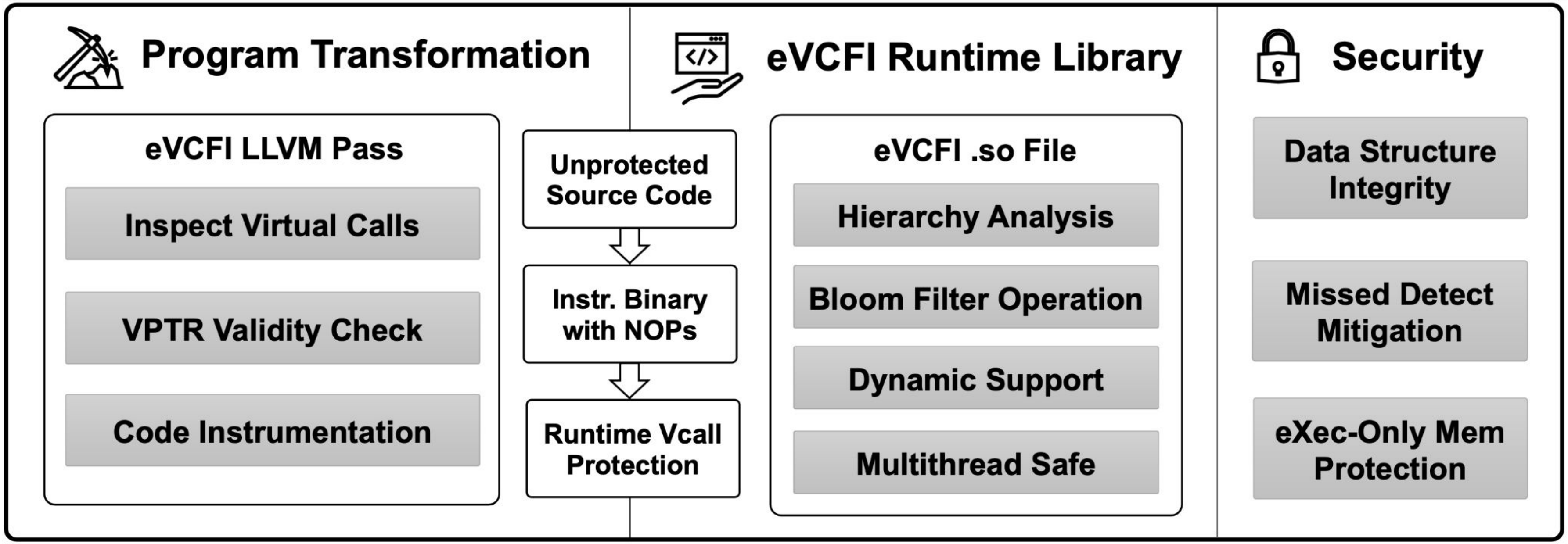


EVCFI Overview



Static – LLVM Pass: static allowset analysis

EVCFI Overview



Static – LLVM Pass: static allowset analysis

Dynamic – Runtime: updating allowset dynamically

EVCFI Design

```
movabs $SALT,%rdi    # Load 64-bit SALT
imul %rax,%rdi       # Multiply
xor %esi,%esi        # Zero accumulator
crc32q %rdi,%rsi     # CRC32
```

Bloom Filter Lookup

EVCFI Design

```
movabs $SALT,%rdi    # Load 64-bit SALT
imul %rax,%rdi       # Multiply
xor %esi,%esi        # Zero accumulator
crc32q %rdi,%rsi     # CRC32
```

Bloom Filter Lookup

Salted Hash Function

$$hash(salt, vptr) = crc32(salt \times vptr)$$

EVCFI Design

```
movabs $SALT,%rdi     # Load 64-bit SALT
imul %rax,%rdi        # Multiply
xor %esi,%esi         # Zero accumulator
crc32q %rdi,%rsi      # CRC32
```

Bloom Filter Lookup

```
mov (%rdi),%rax       # Load vptr
...                   # Hash into %rsi
movabs $BLOOM,%rdx    # Load Bloom base
testb $0,(%rdx,%rsi)
jnz .LOK              # Entry non-zero?
ud2                   # Invalid vptr
.LOK:
...                   # Repeat for k > 1
...                   # Setup parameters
callq *INDEX(%rax)    # Call virtualFn()
```

VCALL Hardening

Salted Hash Function

$$hash(salt, vptr) = crc32(salt \times vptr)$$

EVCFI Design

```
movabs $SALT,%rdi     # Load 64-bit SALT
imul %rax,%rdi        # Multiply
xor %esi,%esi         # Zero accumulator
crc32q %rdi,%rsi      # CRC32
```

Bloom Filter Lookup

Salted Hash Function

$$hash(salt, vptr) = crc32(salt \times vptr)$$

```
mov (%rdi),%rax          # Load vptr
...                      # Hash into %rsi
movabs $BLOOM,%rdx       # Load Bloom base
testb $0,(%rdx,%rsi)
jnz .LOK                 # Entry non-zero?
ud2                      # Invalid vptr
.LOK:
...                      # Repeat for k > 1
...                      # Setup parameters
callq *INDEX(%rax)       # Call virtualFn()
```

VCALL Hardening

Efficiency: only 8*k instructions to complete vcall hardening

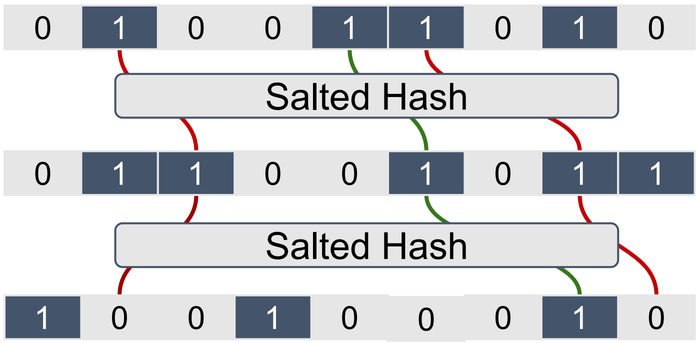
EVCFI Design – Security

Several Security Concerns:

EVCFI Design – Security

Several Security Concerns:

- Hash Collision (Miss Detection)

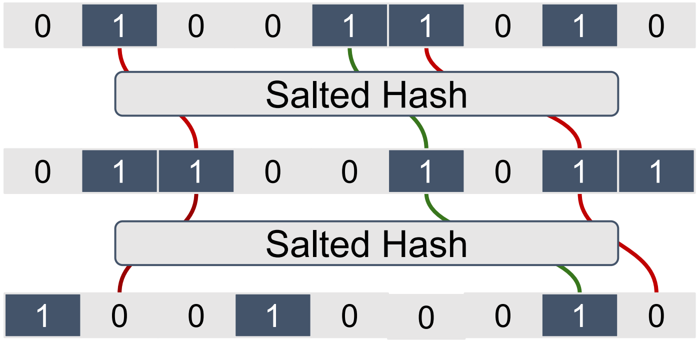


BF is inherently probabilistic

EVCFI Design – Security

Several Security Concerns:

- Hash Collision (Miss Detection)
    - Adding salt value

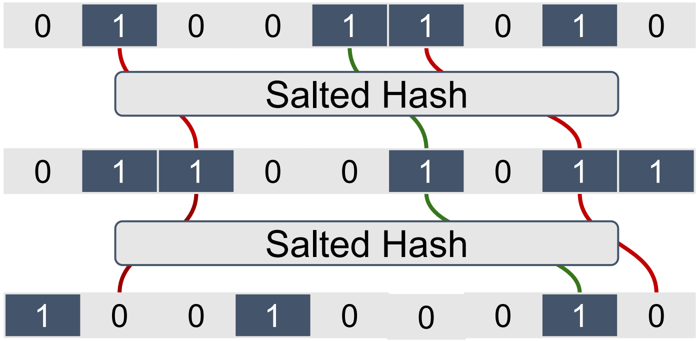


BF is inherently probabilistic

EVCFI Design – Security

Several Security Concerns:

- Hash Collision (Miss Detection)
    - Adding salt value
    - Increasing k value

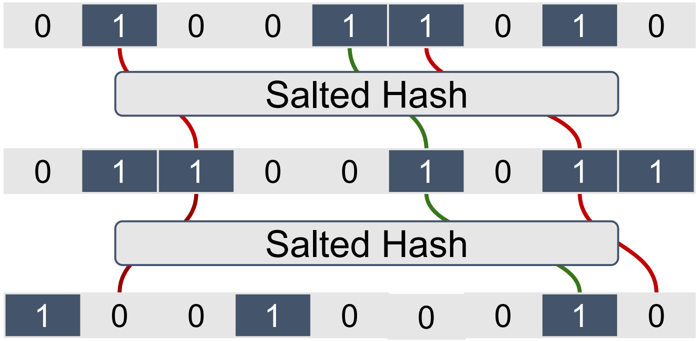


BF is inherently probabilistic

EVCFI Design – Security

Several Security Concerns:

- Hash Collision (Miss Detection)
    - Adding salt value
    - Increasing k value
    - Using more secure hash functions

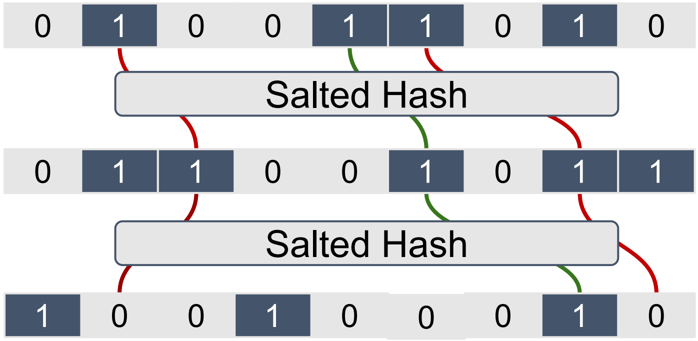


BF is inherently probabilistic

EVCFI Design – Security

Several Security Concerns:

- Hash Collision (Miss Detection)
  - Adding salt value
  - Increasing k value
  - Using more secure hash functions

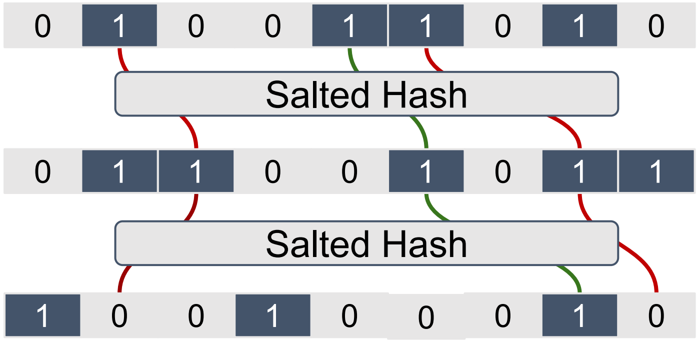


BF is inherently probabilistic

- Bloom Filter Corruption

EVCFI Design – Security

Several Security Concerns:



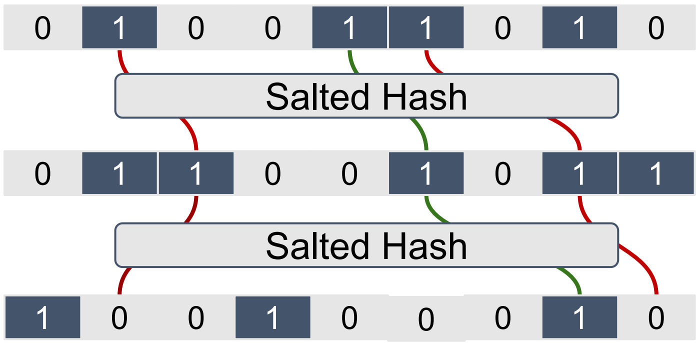
BF is inherently probabilistic

- Hash Collision (Miss Detection)
  - Adding salt value
  - Increasing k value
  - Using more secure hash functions

- Bloom Filter Corruption
  - Randomized bloom base

EVCFI Design – Security

Several Security Concerns:

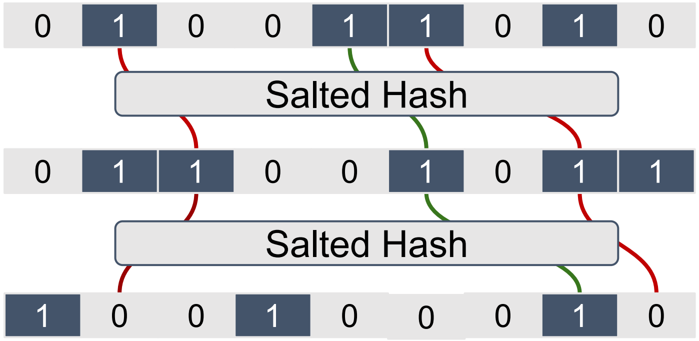


BF is inherently probabilistic

- Hash Collision (Miss Detection)
  - Adding salt value
  - Increasing k value
  - Using more secure hash functions
- Bloom Filter Corruption
  - Randomized bloom base
- Leak salt from instructions

EVCFI Design – Security

Several Security Concerns:



BF is inherently probabilistic

- Hash Collision (Miss Detection)
  - Adding salt value
  - Increasing k value
  - Using more secure hash functions

- Bloom Filter Corruption
  - Randomized bloom base

- Leak salt from instructions
  - eXecution Only Memory

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable							
VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Policies
  - MCFI: weakest type-based CFI-policy

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Policies
  - MCFI: weakest type-based CFI-policy
  - VTV: it does not detect the derived class attacks

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable							
VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Policies
  - MCFI: weakest type-based CFI-policy
  - VTV: it does not detect the derived class attacks
  - LLVM: it requires global class hierarchy statically

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable							
VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Features
  - Link Time Optimization(LTO): statical class hierarchy

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable							
VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Features
  - Link Time Optimization(LTO): statical class hierarchy
  - non-LTO: VCFI defense is applicable without LTO

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Features
  - Link Time Optimization(LTO): statical class hierarchy
  - non-LTO: VCFI defense is applicable without LTO
  - Ad Hoc: e.g., supporting foreign language interfaces

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Static Overhead
  - LLVM does not support dynamic cases

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable							
VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Static Overhead
  - LLVM does not support dynamic cases
  - MCFI - astar: high overhead even in low vcall program

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- VCFI Static Overhead
    - LLVM does not support dynamic cases
    - MCFI - astar: high overhead even in low vcall program
    - VTV/ShrinkWrap - xalanc: high overhead

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- Limitations of existing approaches:
  - Limited VCFI policy: MCFI, VTV, LLVM
  - LTO requisite: LLVM, LLVM-xDSO
  - High overhead: MCFI, VTV, Shrinkwrap

Evaluation – Overall Comparison

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

- Limitations of existing approaches:
  - Limited VCFI policy: MCFI, VTV, LLVM
  - LTO requisite: LLVM, LLVM-xDSO
  - High overhead: MCFI, VTV, Shrinkwrap

eVCFI is the first VCFI that achieves efficiency and extensibility

Evaluation – Firefox Support

- FireFox – highly modular application

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables
  - 185k+ virtual call-sites

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables
  - 185k+ virtual call-sites
  - foreign language interfaces, e.g., Rust and C++

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables
  - 185k+ virtual call-sites
  - foreign language interfaces, e.g., Rust and C++

No VCFI Support

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables
  - 185k+ virtual call-sites
  - foreign language interfaces, e.g., Rust and C++

- With high extensibility, eVCFI defense can support Firefox

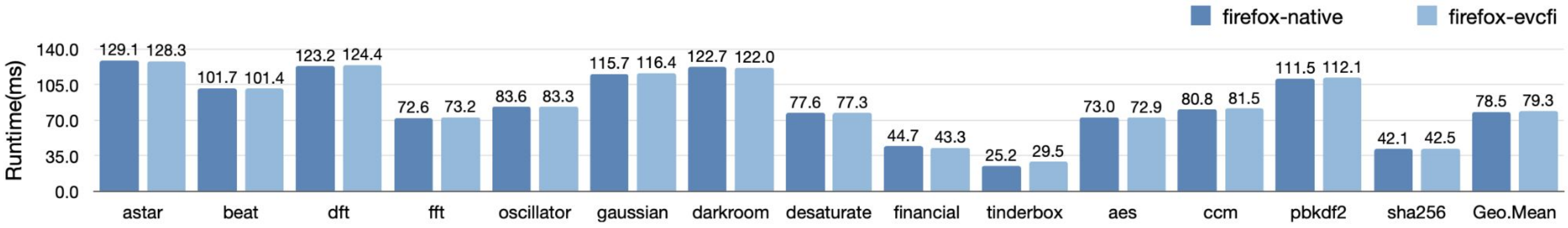
eVCFI Support

Evaluation – Firefox Support

- FireFox – highly modular application
  - 5000+ virtual tables
  - 185k+ virtual call-sites
  - foreign language interfaces, e.g., Rust and C++

eVCFI Support

- With high extensibility, eVCFI defense can support Firefox



- With high efficiency, eVCFI only incurs 1.01% slowdown

Summary

- We propose a novel Extensible-VCFI (EVCFI)
  - The first AMQ-based VCFI
- EVCFI is efficient, extensible, and secure
- EVCFI can support the Firefox, one real-world challenging application for extensibility, which has not been supported by existing VCFIs

Thanks