Kernel CARAT --- “KARAT”

Souradip Ghosh, Drew Kersner, Gaurav Chaudhary
Brian Suchy, Peter Dinda

Outline

- CARAT --- Overview
- Kernel CARAT --- “KARAT”
  - CARAT in Nautilus
  - Compiler Mechanics
  - Runtime Mechanics
- Next Steps

Outline

- **CARAT --- Overview**
- Kernel CARAT --- “KARAT”
  - CARAT in Nautilus
  - Compiler Mechanics
  - Runtime Mechanics
- Next Steps

CARAT --- Overview

Replaces the **paging model** of memory management with a **software-only** abstraction

Comprised of two overarching pieces:

- Runtime --- To dynamically build a view of a program's memory
- Compiler Transforms ---
  - To inject code that calls runtime methods for building a memory map
  - To provide protection for accessing and/or referencing memory

Outline

- CARAT --- Overview
- **Kernel CARAT --- “KARAT”**
  - CARAT in Nautilus
  - Compiler Mechanics
  - Runtime Mechanics
- Next Steps

Kernel CARAT --- “KARAT”

- The current implementation of CARAT is at **user level**
- To truly test the concept, we need to try it at **kernel level**
- This is a **non-trivial** task:
    - Need to account for complex paging systems --- CARAT works at the allocation granularity of pages
    - Applying whole-kernel compiler transformations is tricky
    - Multiple processors, context switching, etc. will cause headaches
    - Need to confirm that KARAT will not affect the memory allocation system

Outline

- CARAT --- Overview
- **Kernel CARAT --- “KARAT”**
  - **CARAT in Nautilus**
  - Compiler Mechanics
  - Runtime Mechanics
- Example
- Next Steps

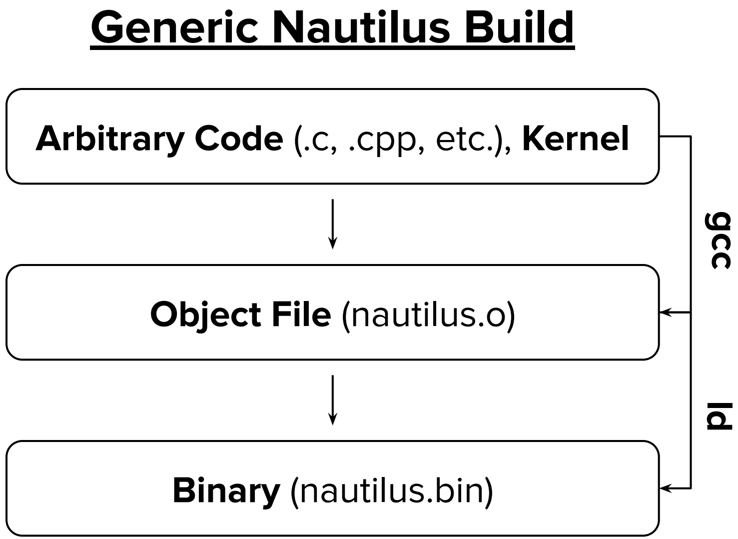
KARAT: CARAT in Nautilus

- To run CARAT in Nautilus, we need to accomplish the following:
  - **Port** the CARAT **runtime** into Nautilus
  - **Port** and **run** the CARAT **compiler transforms** on all Nautilus code, injecting calls to new runtime and custom instructions into the kernel
- Nautilus has **advantages**:
  - Uses the **simplest paging** implementation
  - Nautilus **can compile** with **several compiler toolchains**, including ones with custom transformations

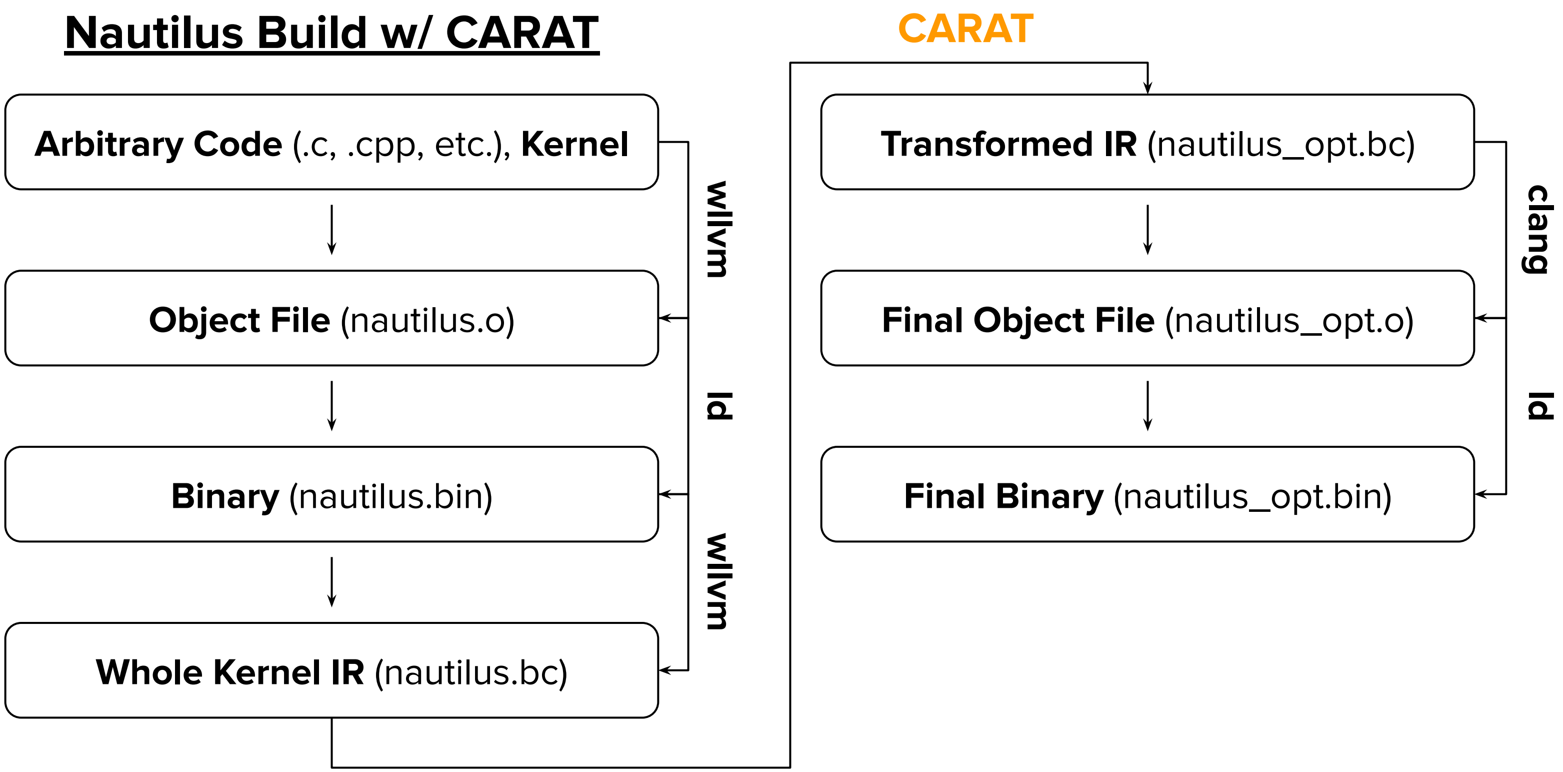
Outline

- CARAT --- Overview
- **Kernel CARAT --- “KARAT”**
    - CARAT in Nautilus
    - **Compiler Mechanics**
    - Runtime Mechanics
- Next Steps

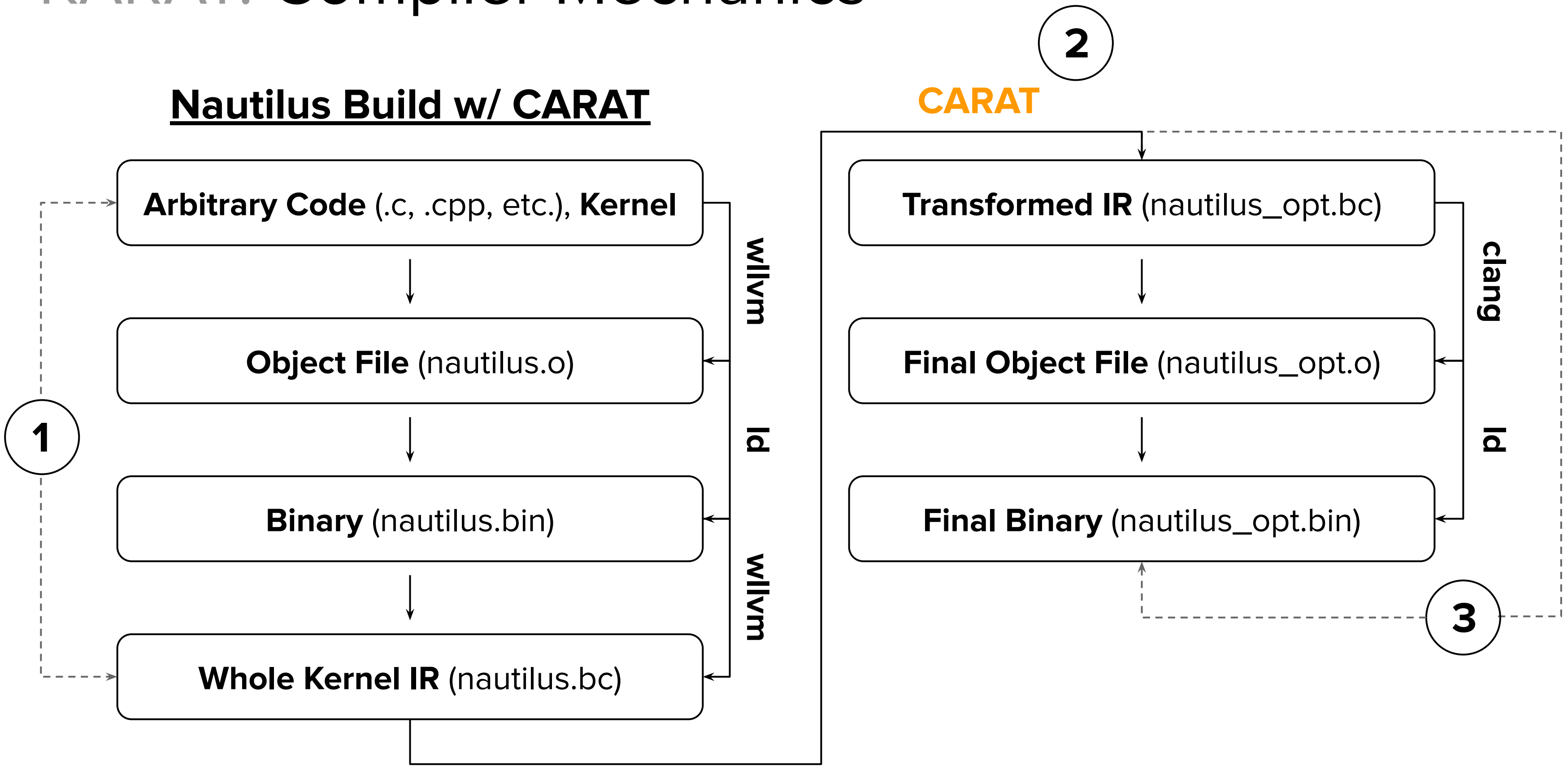
KARAT: Compiler Mechanics



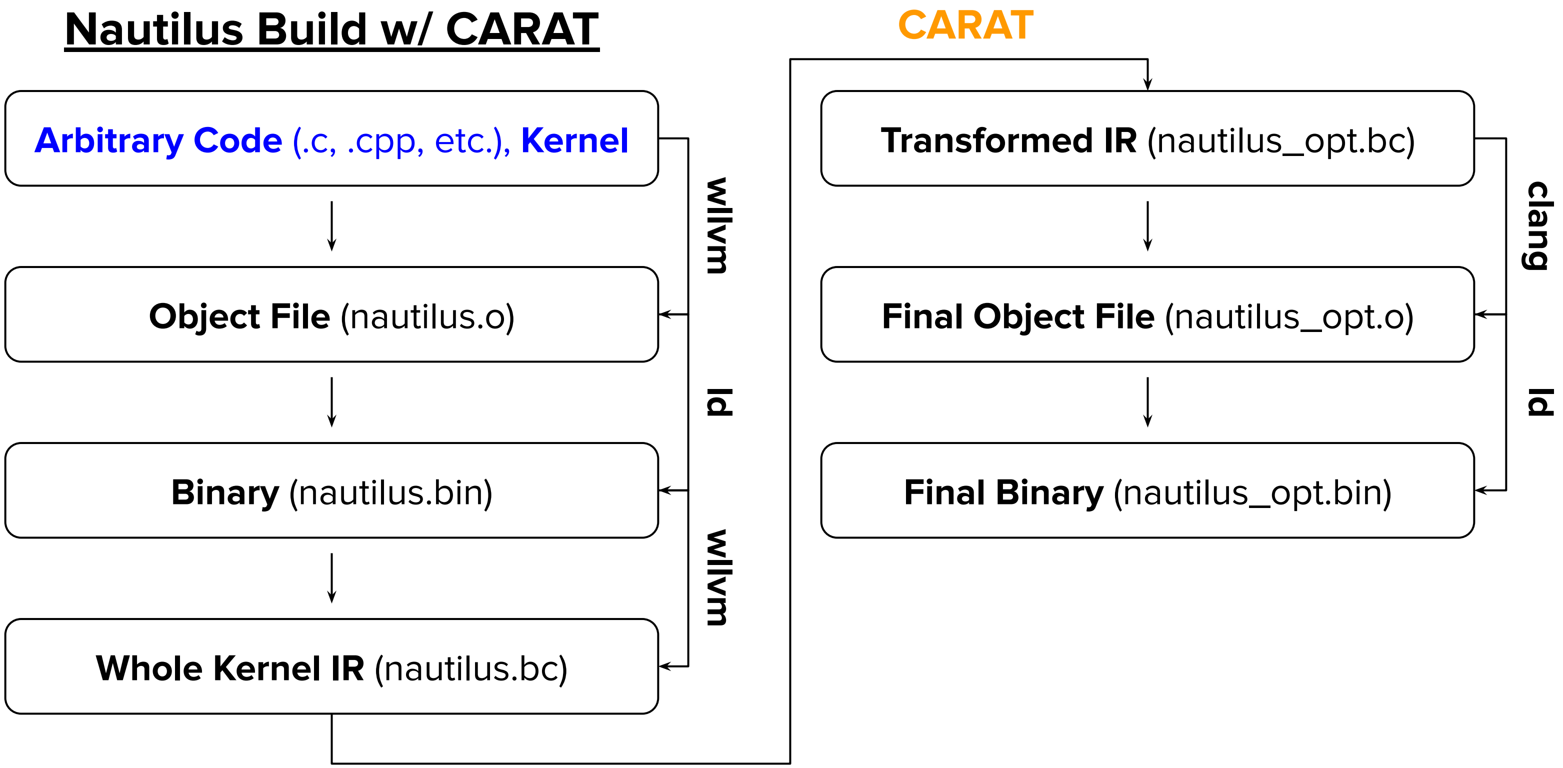
KARAT: Compiler Mechanics



KARAT: Compiler Mechanics



KARAT: Compiler Mechanics



Arbitrary Code (.c, .cpp, etc.), **Kernel**

↓

. . .

CARAT Runtime

`nk_set`

`nk_map`

`nk_slist`

Arbitrary Code (.c, .cpp, etc.), **Kernel**

...

CARAT Runtime

`nk_set`

`nk_map`

`nk_slist`

- We're porting a simplified version of the CARAT runtime
- But ... porting from C++ to C is **complicated**

CARAT Runtime

`nk_set`

`nk_map`

`nk_slist`

- Nautilus does **not** include the C++ STL or many data structures
- We have to **build** the ones we need in **C**

CARAT Runtime

`nk_set`

`nk_map`

`nk_slist`

- We need sorted **maps** and sorted **sets**

CARAT Runtime

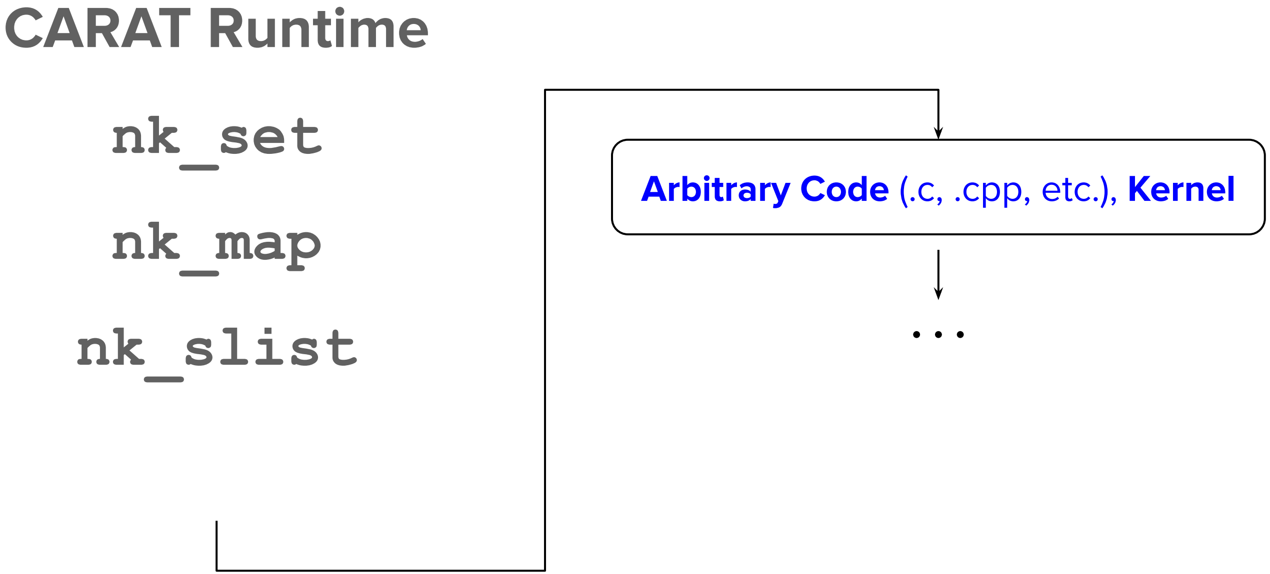
`nk_set`

`nk_map`

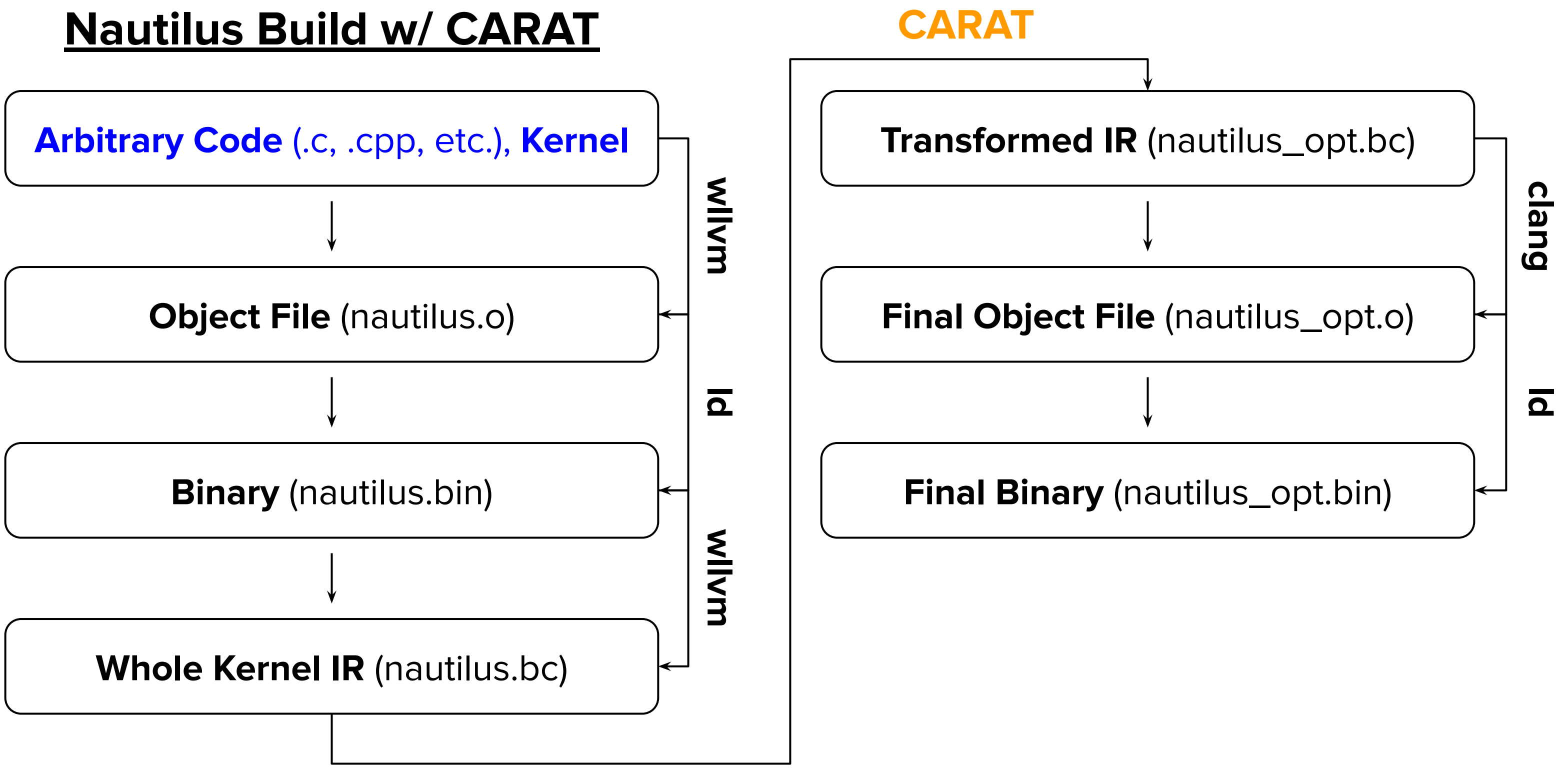
`nk_slist`

- We need sorted **maps** and sorted **sets**

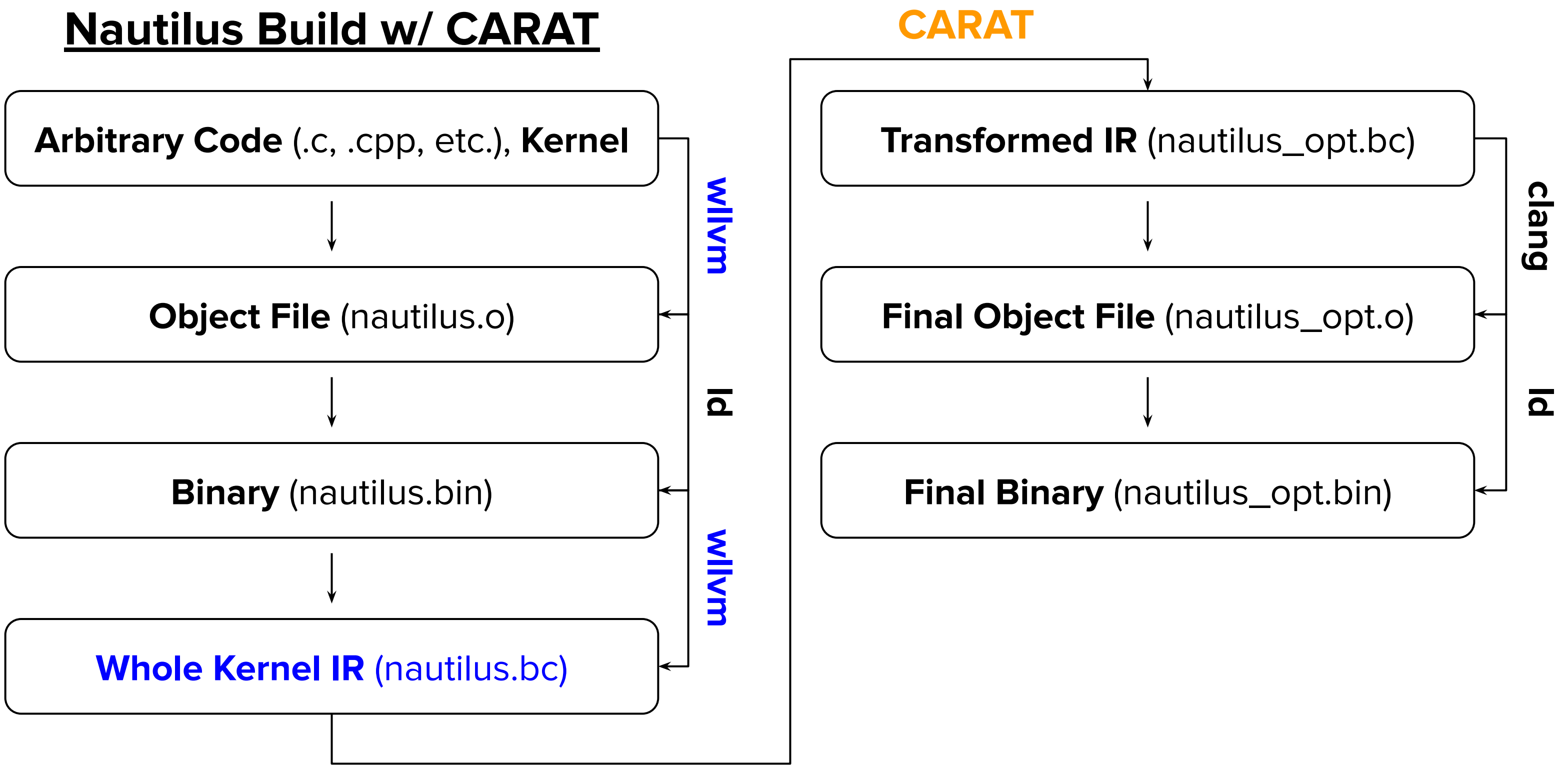
- We're using **skiplists** as the underlying data structure to create these abstractions

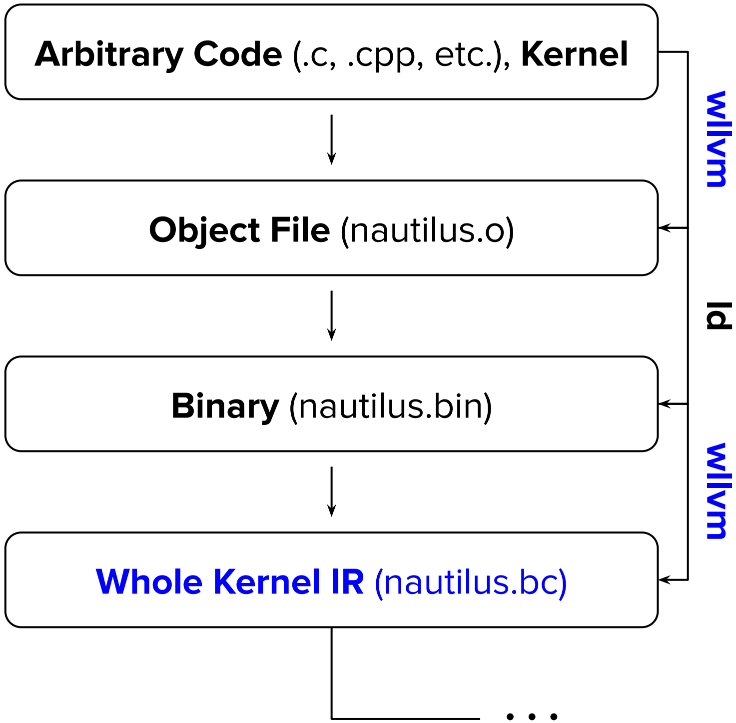

CARAT Runtime
nk_set
nk_map
nk_slist
Arbitrary Code (.c, .cpp, etc.), Kernel
...

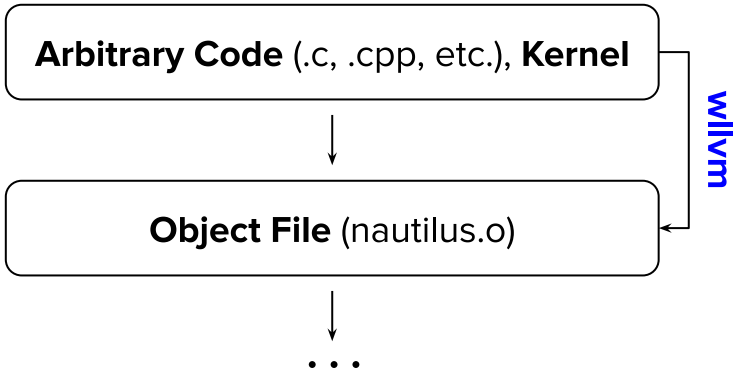
KARAT: Compiler Mechanics



KARAT: Compiler Mechanics



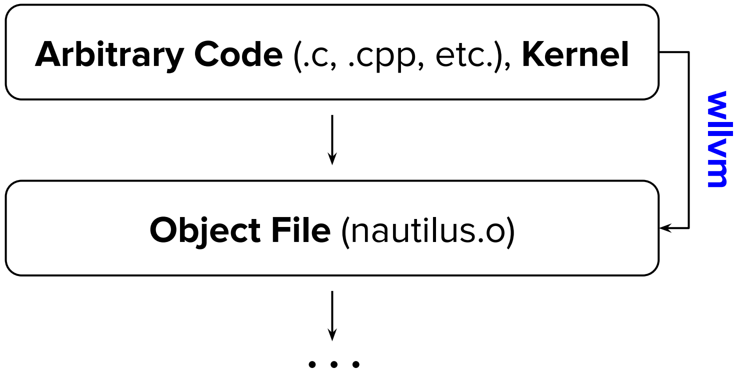

Arbitrary Code (.c, .cpp, etc.), **Kernel**
Object File (nautilus.o)
Binary (nautilus.bin)
Whole Kernel IR (nautilus.bc)
wllvm
ld
wllvm
. . .


Arbitrary Code (.c, .cpp, etc.), **Kernel**
wllvm
Object File (nautilus.o)
. . .

LLVM

clang

wllvm



LLVM

- LLVM is the **compiler infrastructure** we're using to build and transform Nautilus
- We're targeting the middle-end (**IR**)

LLVM

clang

wllvm

- Clang is the **front-end** that LLVM uses to compile C and C++ sources --- we usually refer to the whole compiler as “clang”

LLVM

clang

wllvm

- WLLVM, or **whole program LLVM**, is a wrapper built on top of clang that separately produces **LLVM IR** for any compiled source

LLVM

clang

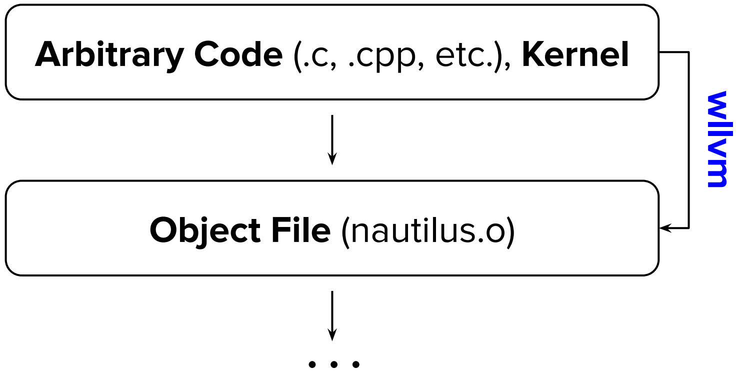
wllvm

- WLLVM uses a specialized **linker** and **IR generator** that builds and stores the IR in a specific section of the object/binary

LLVM

clang

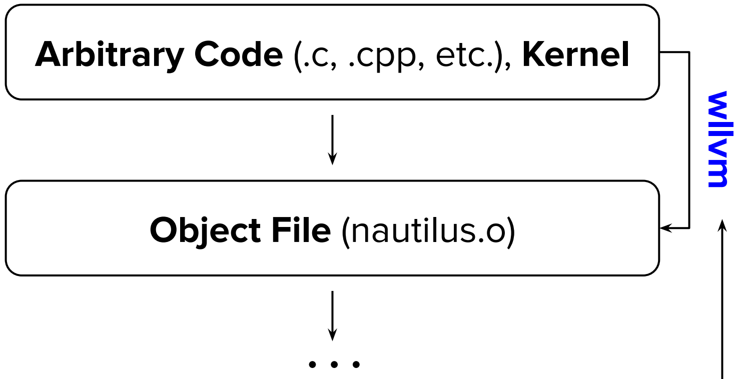
wllvm



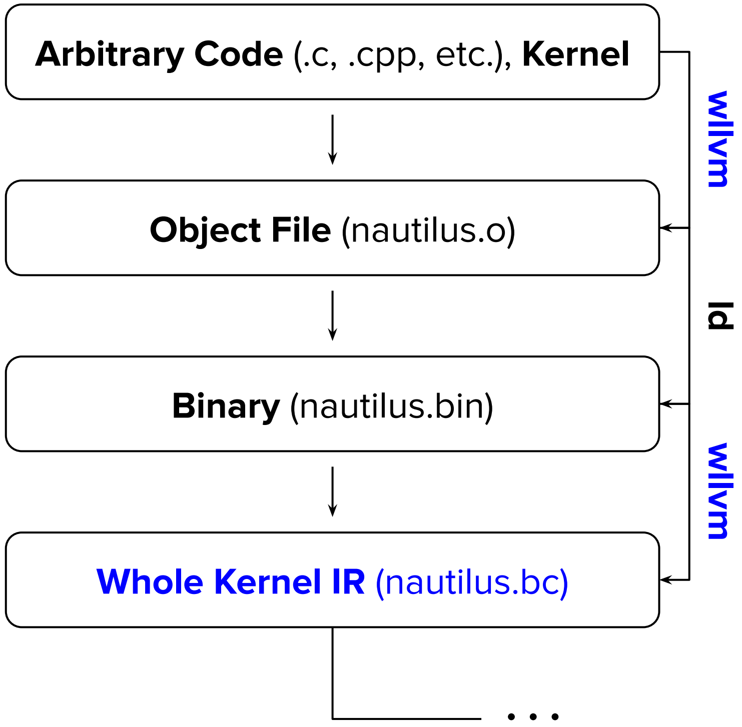
LLVM

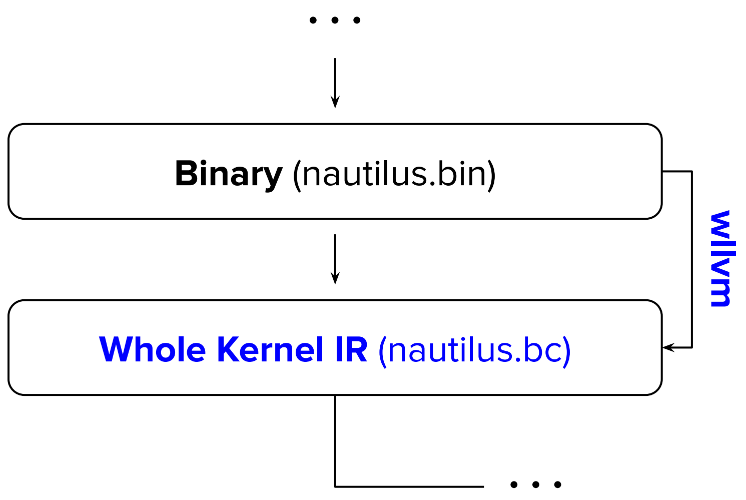
clang

wllvm



All three pieces are invoked in this step!

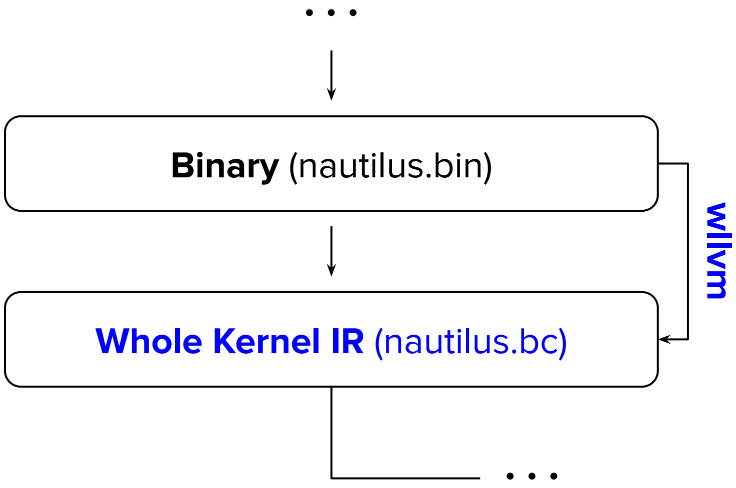

Arbitrary Code (.c, .cpp, etc.), Kernel
Object File (nautilus.o)
Binary (nautilus.bin)
Whole Kernel IR (nautilus.bc)
wllvm
ld
wllvm
...



. . .
Binary (nautilus.bin)
wllvm
Whole Kernel IR (nautilus.bc)
. . .

wllvm

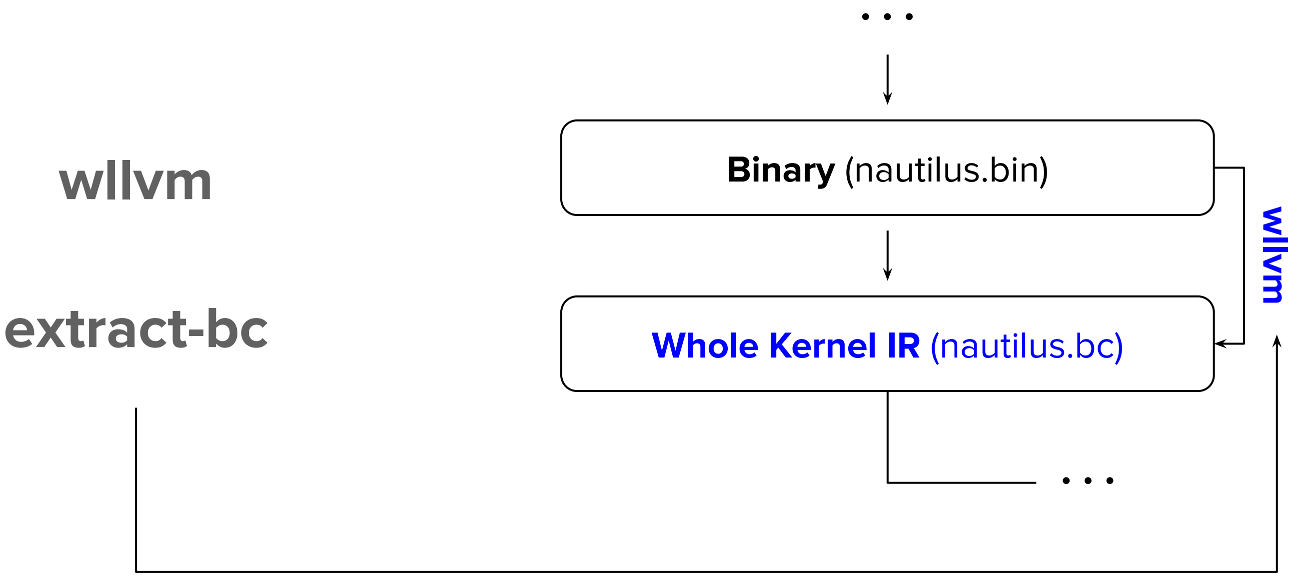
extract-bc



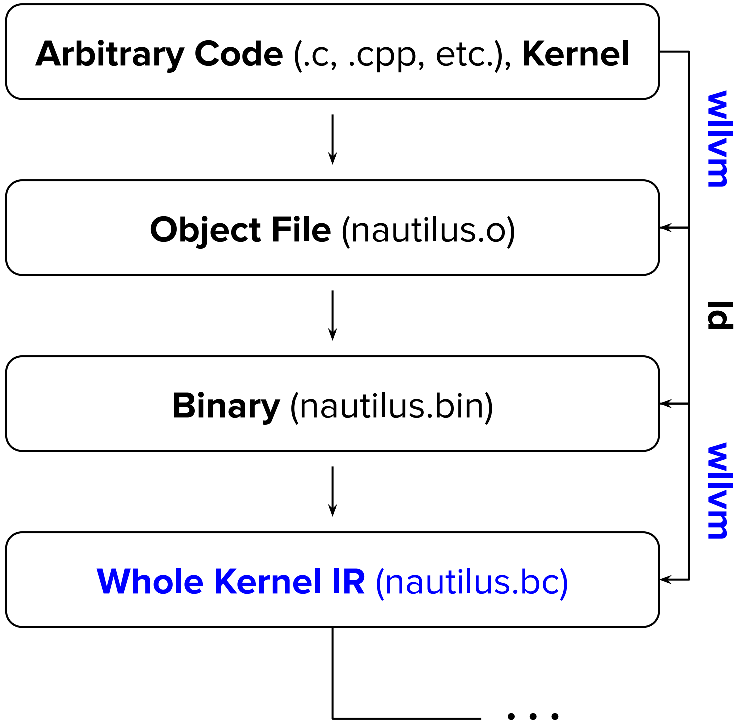
wllvm

extract-bc

- extract-bc is the feature of WLLVM that allows a user to extract **all LLVM IR** from a object/binary

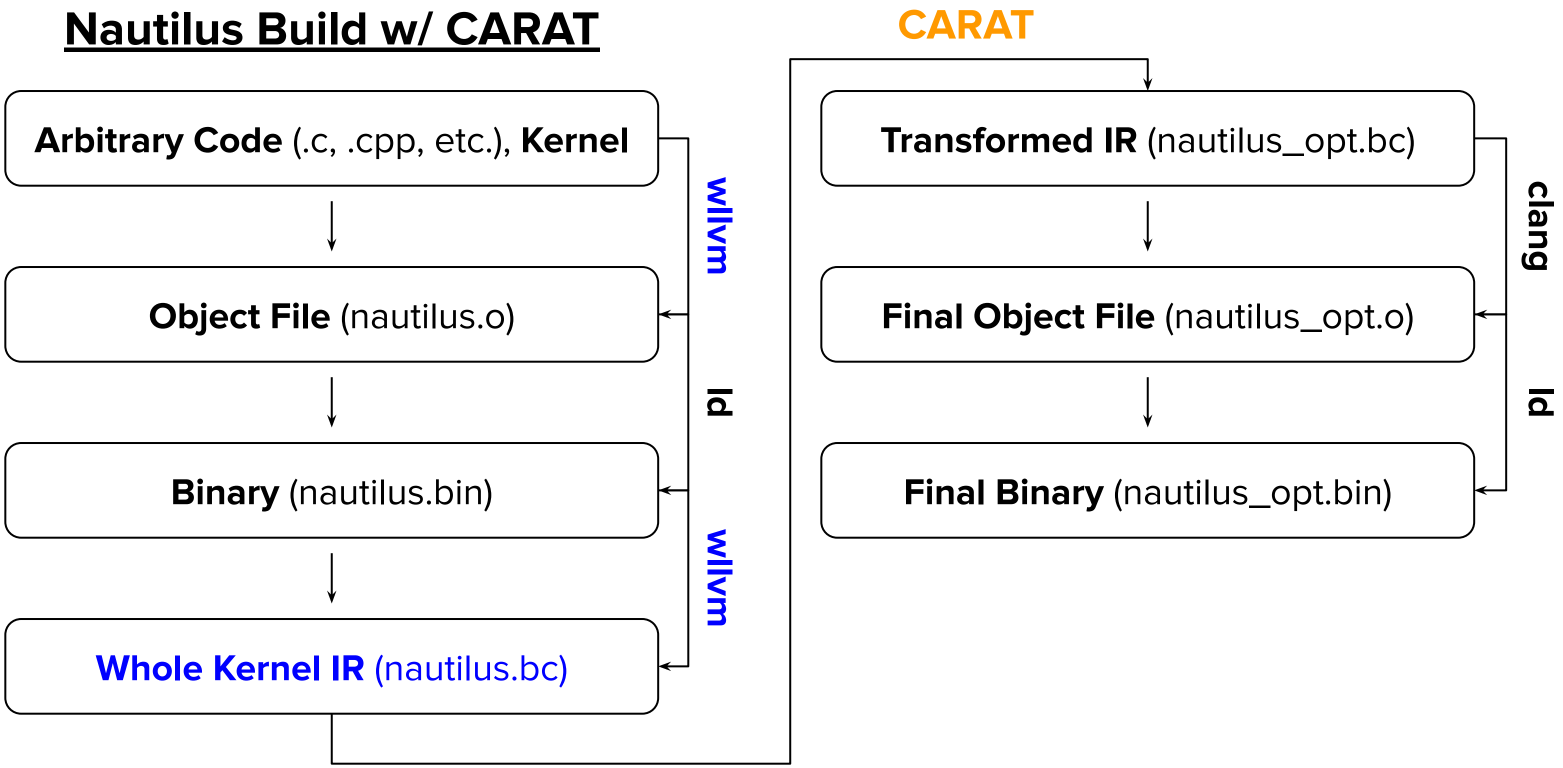


extract-bc gets the **whole kernel IR** from nautilus.bin

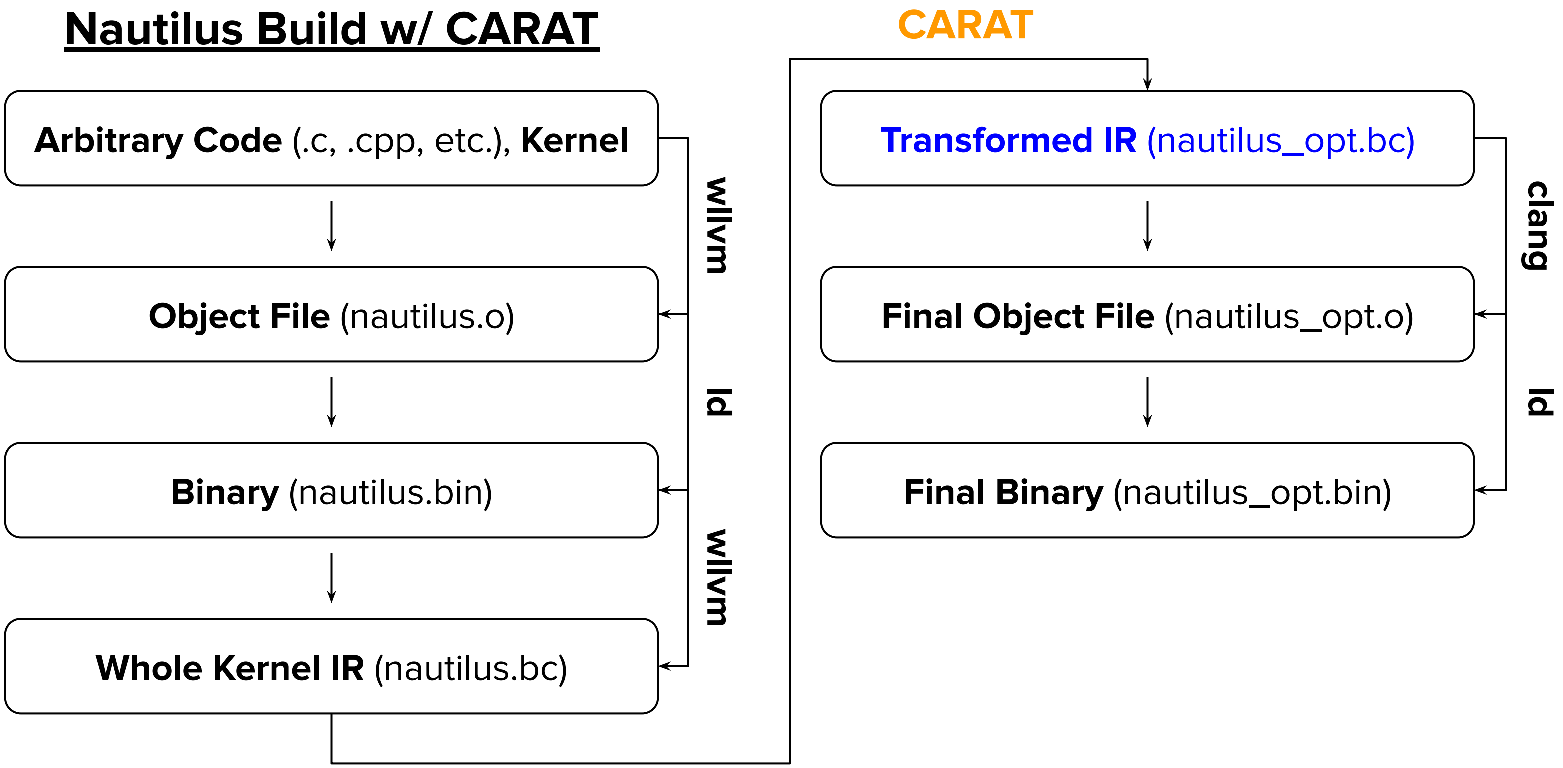


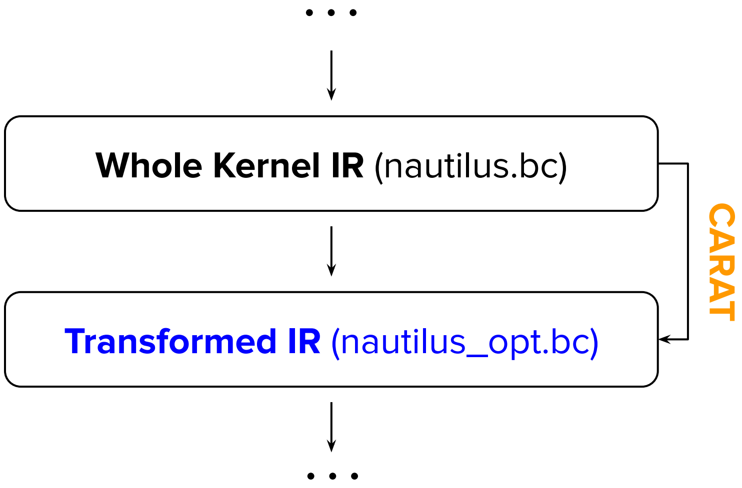
Arbitrary Code (.c, .cpp, etc.), **Kernel**
Object File (nautilus.o)
Binary (nautilus.bin)
Whole Kernel IR (nautilus.bc)
wllvm
ld
wllvm
. . .

KARAT: Compiler Mechanics



KARAT: Compiler Mechanics

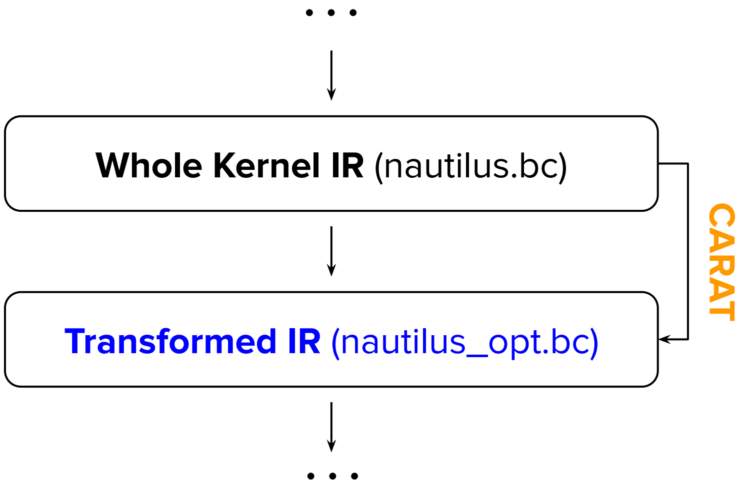



...
Whole Kernel IR (nautilus.bc)
Transformed IR (nautilus_opt.bc)
CARAT
...

Allocation Tracking

Escapes Tracking

Protections



Allocation Tracking

Escapes Tracking

Protections

- Allocation tracking is a feature in CARAT that tracks **heap memory** while the kernel is running

Allocation Tracking

Escapes Tracking

Protections

- In Nautilus, this means all calls to **`kmem_malloc`**, **`kmem_realloc`**, and **`free`** are tracked

Allocation Tracking

Escapes Tracking

Protections

- All **globals** are also tracked, since global variables are generally heap-allocated

Allocation Tracking

Escapes Tracking

Protections

- But what does **tracking** mean? All calls are succeeded with a call to the runtime that records the **pointer** and **size of the allocation** to a table

Allocation Tracking

Escapes Tracking

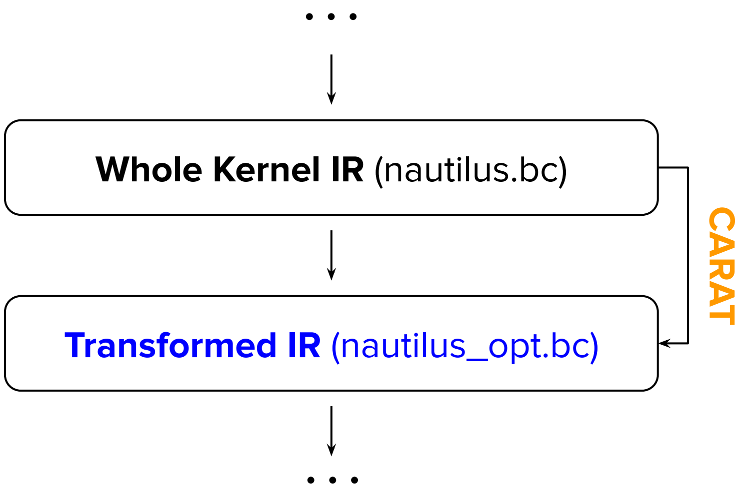
Protections

- Allocation tracking is implemented as a middle-end LLVM transform --- **injecting calls** to the runtime

Allocation Tracking

Escapes Tracking

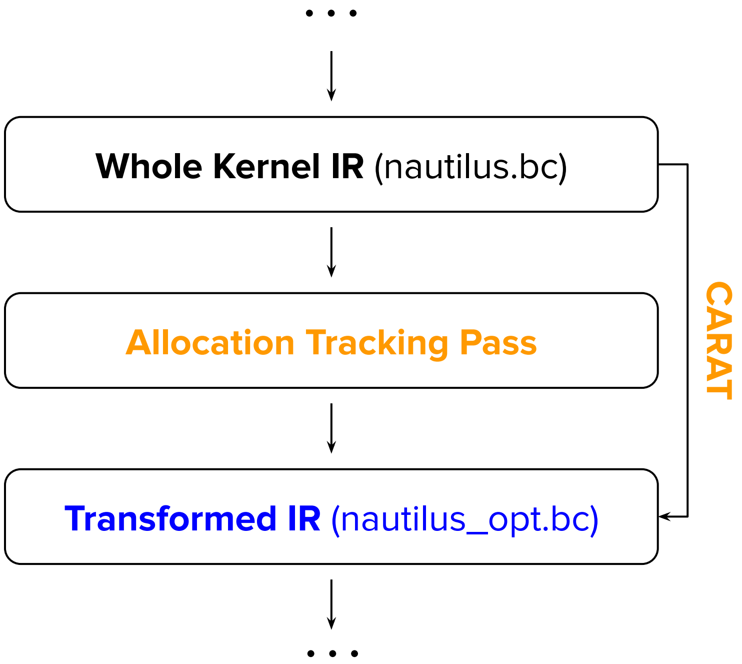
Protections



Allocation Tracking

Escapes Tracking

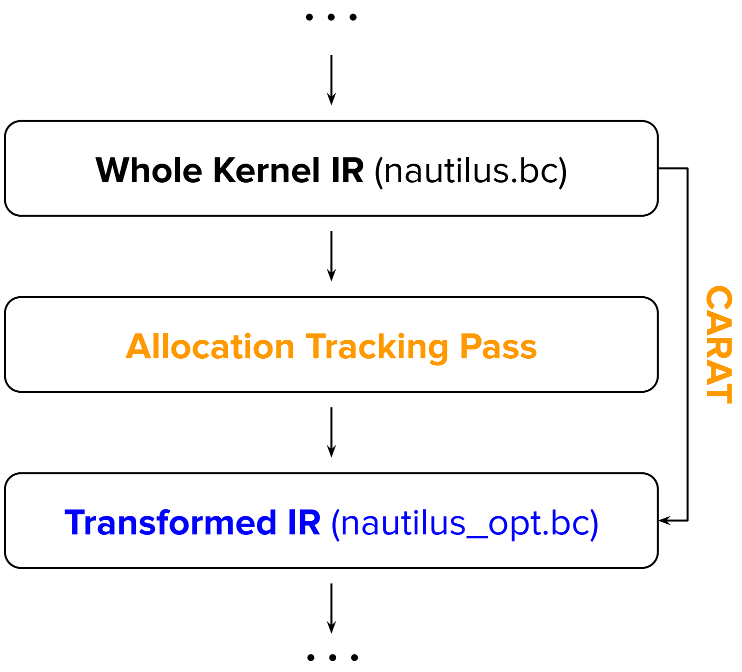
Protections



Allocation Tracking

Escapes Tracking

Protections



Allocation Tracking

Escapes Tracking

Protections

- Escapes tracking is a feature in CARAT that tracks all “escapes” or **references to allocated memory**

Allocation Tracking

Escapes Tracking

Protections

- Here, **tracking** applies to all **stores** of pointers
- All stores are succeeded with a call to the runtime to account for the **escaped pointer**

Allocation Tracking

Escapes Tracking

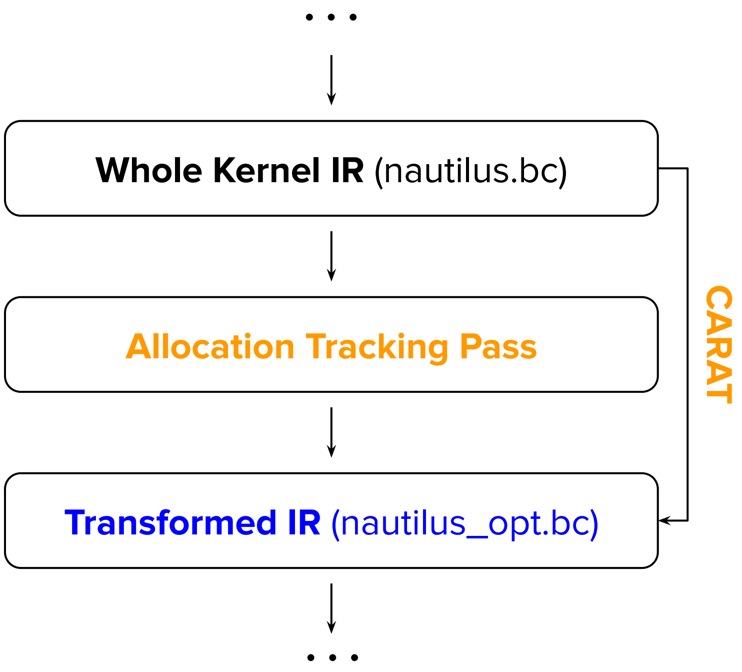
Protections

- Escapes tracking is also implemented as a middle-end LLVM transform --- **injecting calls** to the runtime

Allocation Tracking

Escapes Tracking

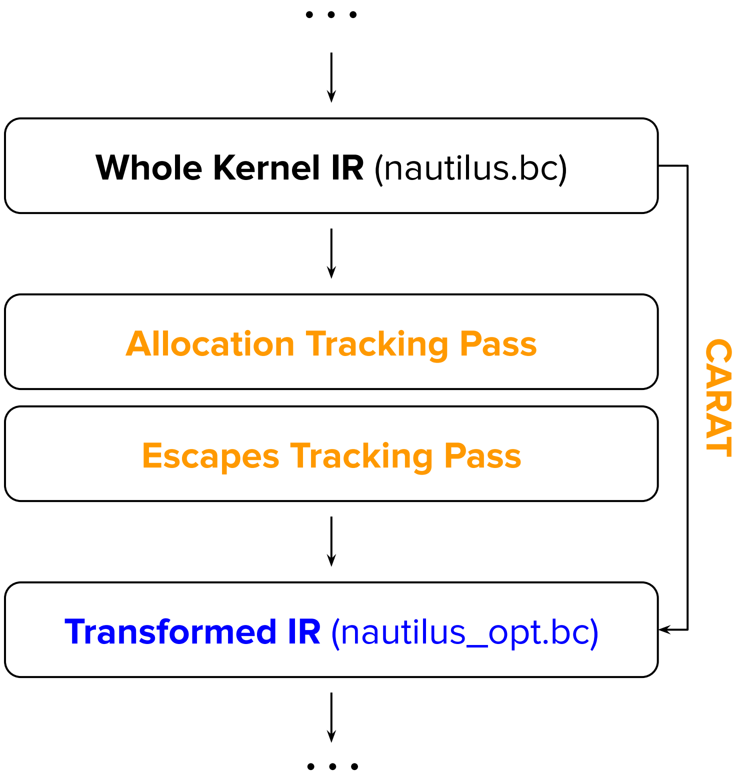
Protections



Allocation Tracking

Escapes Tracking

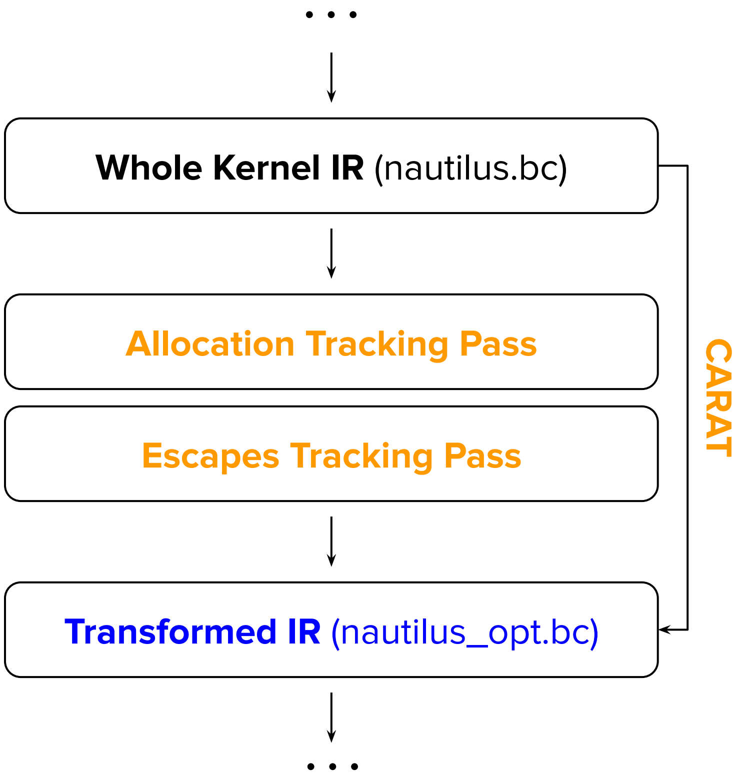
Protections



Allocation Tracking

Escapes Tracking

Protections



Allocation Tracking

Escapes Tracking

Protections

- CARAT provides protections to reads and writes --- making sure all references are **valid**

Allocation Tracking

Escapes Tracking

Protections

- **Valid** addresses in Nautilus correspond to **canonical** addresses, and depend on the address space that Nautilus is using at any given time

Allocation Tracking

Escapes Tracking

Protections

- Protections require checking **all references**
- The injected code will cause a **panic** if a reference is not valid

Allocation Tracking

Escapes Tracking

Protections

- Checking every reference is **expensive** --- CARAT employs a custom **data-flow analysis** and **loop invariant analysis** to reduce overhead

Allocation Tracking

Escapes Tracking

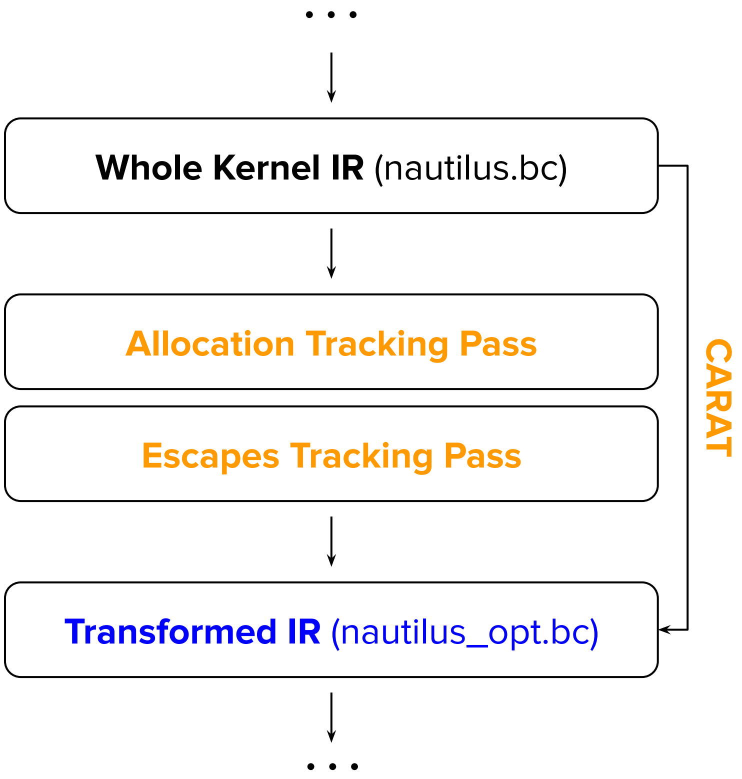
Protections

- Protections are also implemented as a middle-end LLVM transform --- **injecting IR** directly into *nautilus.bc*

Allocation Tracking

Escapes Tracking

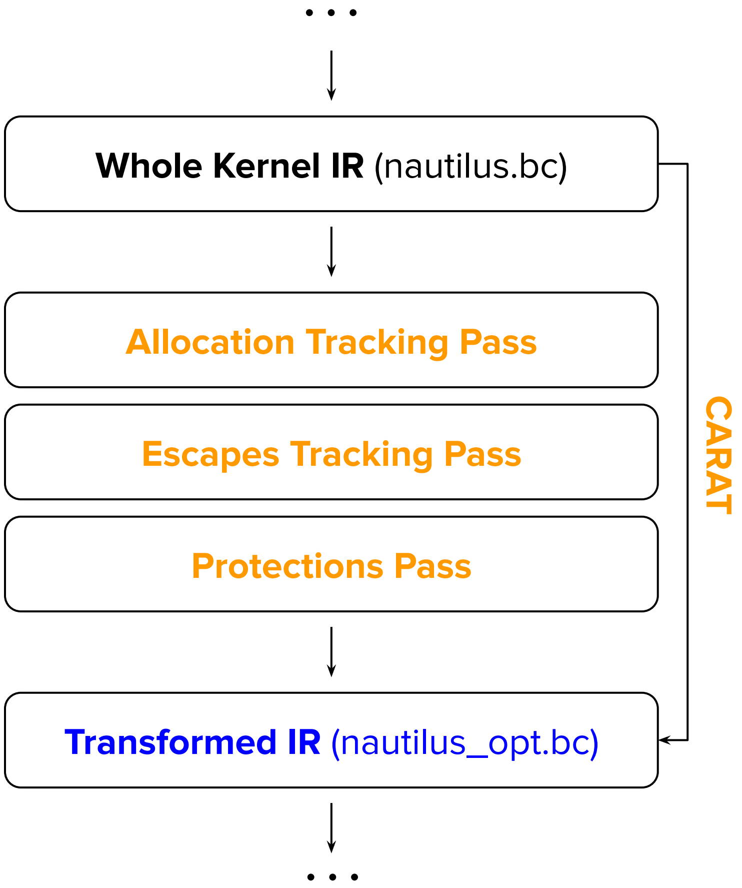
Protections



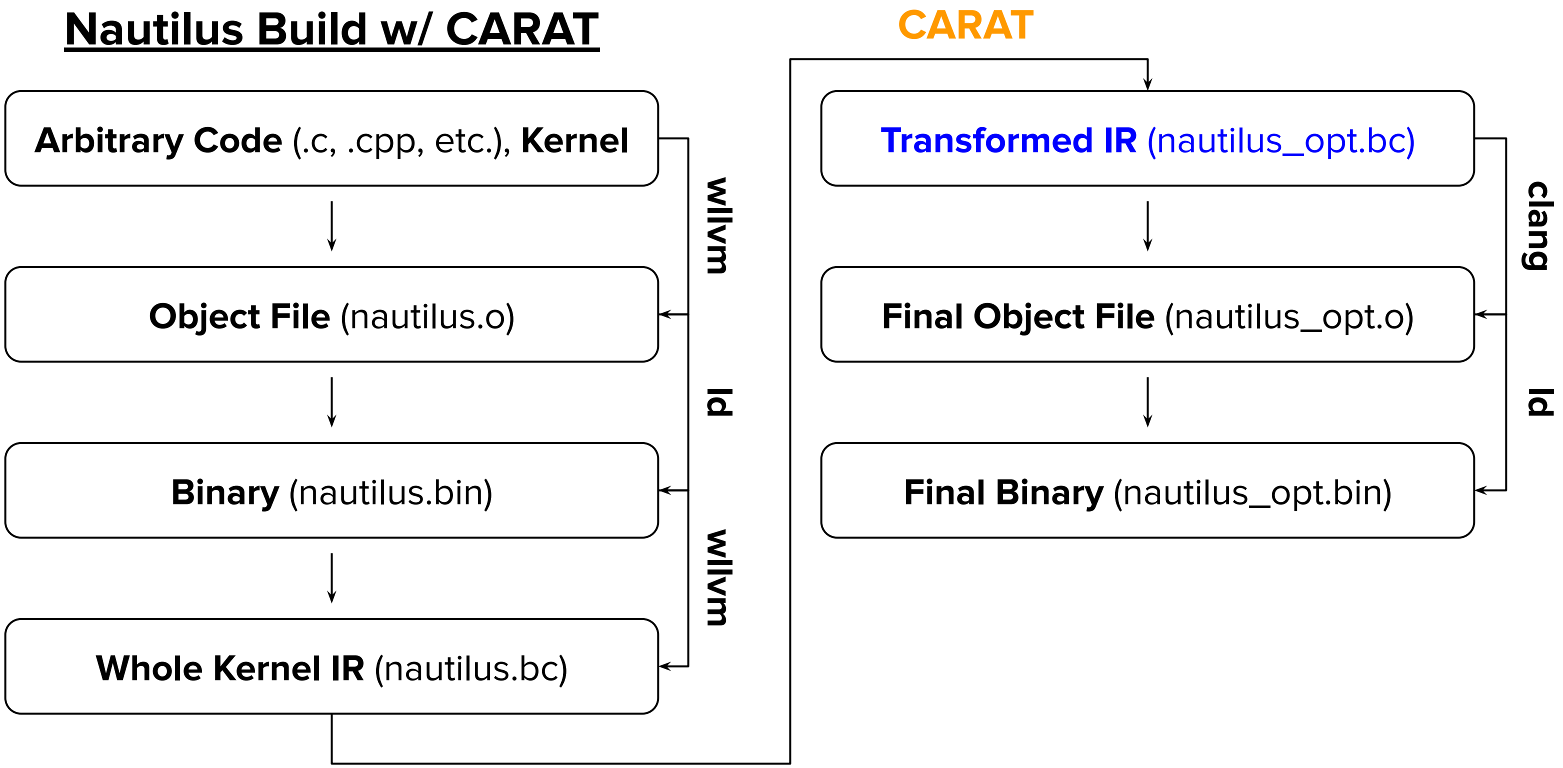
Allocation Tracking

Escapes Tracking

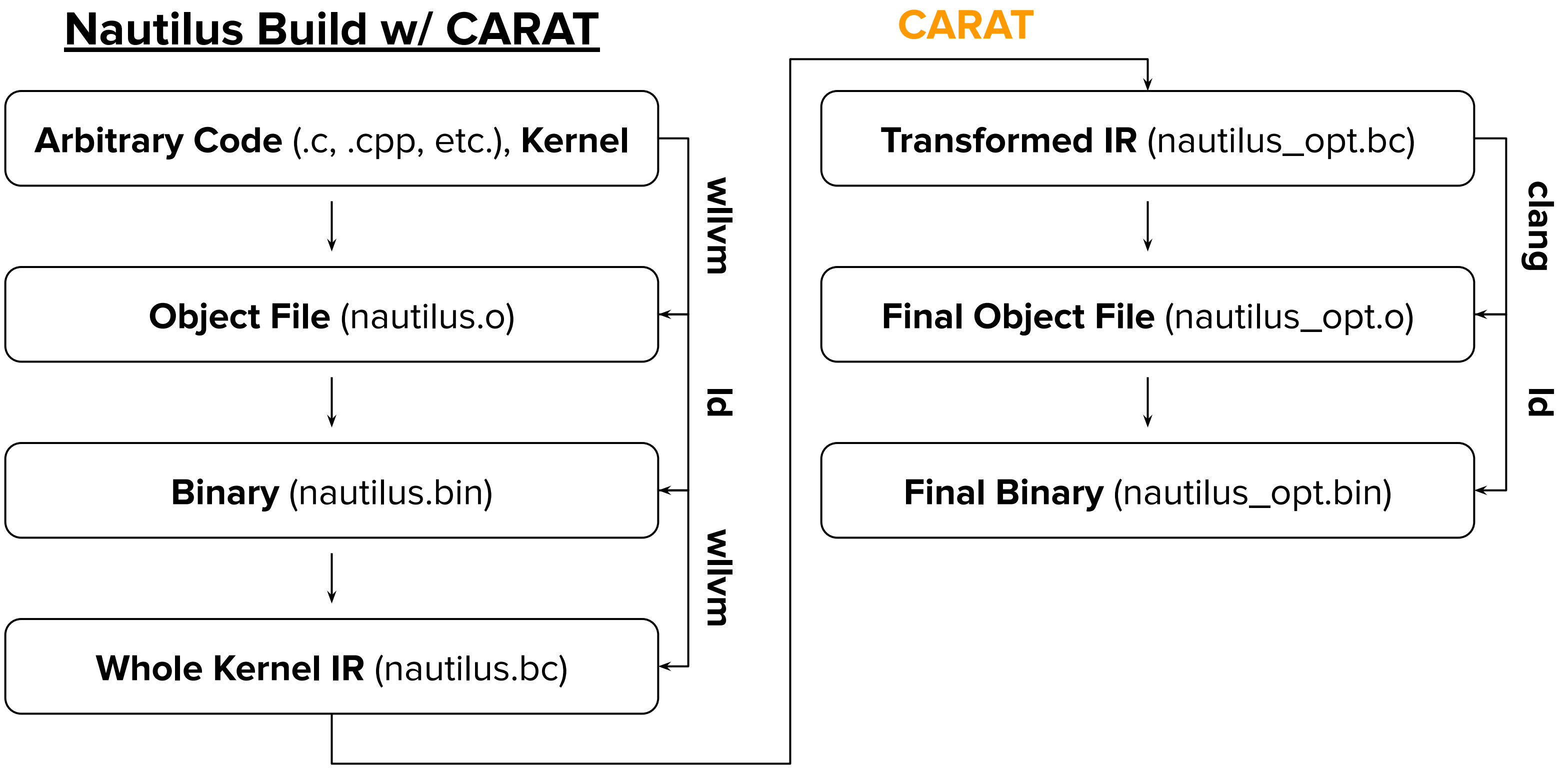
Protections



KARAT: Compiler Mechanics



KARAT: Compiler Mechanics



Outline

- CARAT --- Overview
- **Kernel CARAT --- “KARAT”**
  - CARAT in Nautilus
  - Compiler Mechanics
  - **Runtime Mechanics**
- Next Steps

KARAT: Runtime Mechanics

- Original CARAT runtime has the following features:
  - **Track** memory allocations and escapes
  - Perform **address translation** dynamically
  - Written in **C++**
  - Run at the **user level** and for **user** programs
- KARAT will be using a simplified and more accurate runtime:
  - Performs tracking via kernel **allocation** and **escapes table**
  - Essentially **no** address translation b/c of Nautilus' simple paging
  - Run at the **kernel level** and for **kernel** programs

KARAT: Runtime Mechanics

- What is the runtime?
  - **Compiler transforms** inject calls to the runtime
  - The **runtime** is built into the **kernel** itself
- **What does this look like?**
  - In reality, these will be injections in the LLVM-IR (**middle-end**)
  - We'll show an example of what the transformations and runtime invocations look like at the **source-code** level --- for simplicity

KARAT: Runtime Mechanics

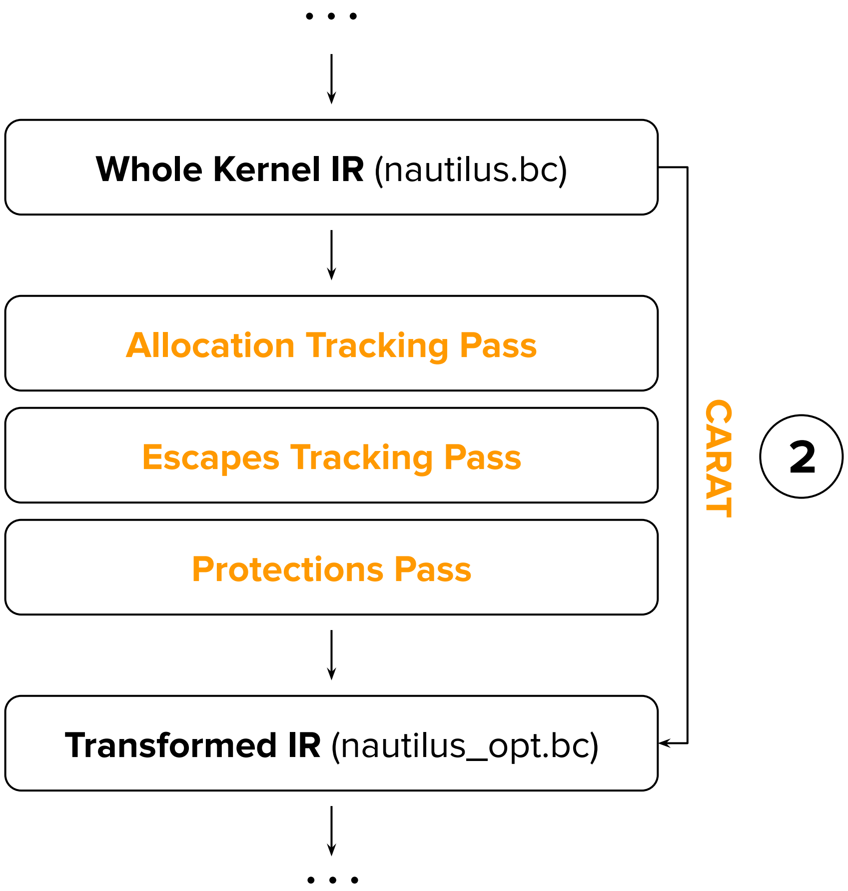
- Example **user code** before any compiler transformation

```c
int main() {
    // Array of 100 ints
    int* x = (int*) malloc(sizeof(int)*100);

    x[2] = 2;

    int* y = &x[2];

    x[3] = 4;
    return 0;
}
```


. . .
Whole Kernel IR (nautilus.bc)
Allocation Tracking Pass
Escapes Tracking Pass
Protections Pass
Transformed IR (nautilus_opt.bc)
. . .
CARAT
2

KARAT: Runtime Mechanics

- Example **user code** before any compiler transformation

```
int main() {
    // Array of 100 ints
    int* x = (int*) malloc(sizeof(int)*100);

    x[2] = 2;

    int* y = &x[2];

    x[3] = 4;
    return 0;
}
```

KARAT: Runtime Mechanics

- Example **user code** with **allocation/escapes** tracking transforms

```c
int main() {
    // Array of 100 ints
    int* x = (int*) malloc(sizeof(int)*100);
    AddToAllocationTable(x, sizeof(int)*100);
    x[2] = 2;

    int* y = &x[2];
    AddToEscapeTable(y);
    x[3] = 4;
    return 0;
}
```

Track **allocation** stored in **x**

Track the **escape** that points to the allocation

KARAT: Runtime Mechanics

- Example **user code** with **protections** transforms

```
int main() {
    // Array of 100 ints
    int* x = (int*) malloc(sizeof(int)*100);
    AddToAllocationTable(x, sizeof(int)*100);
    if(&x[0] < LowerBound || &x[99] > UpperBound){
        abort();
    }
    x[2] = 2;
    // Variable y is stored on our stack and we know it doesn't go over
    int* y = &x[2];
    AddToEscapeTable(y);
    //We know that the entire x array is fine from the first check we performed
    x[3] = 4;
    return 0;
}
```

Check if accessed memory location is a part of the **aspace**

KARAT: Runtime Mechanics --- Put Altogether

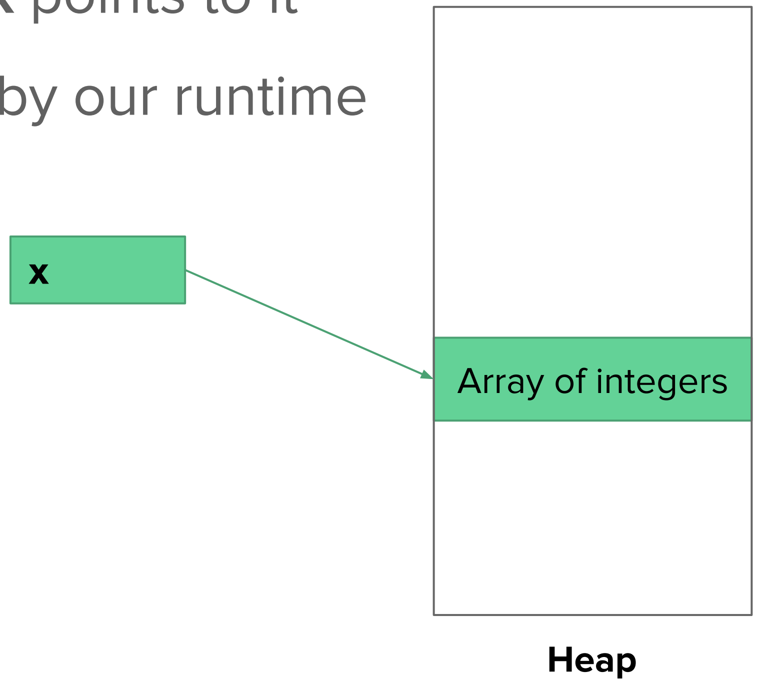
- Why do we track **allocations** and **escapes**?
  - To have a precise picture of memory at every execution step
  - To allocate and deallocate memory without corrupting the already allocated stuff
- This is where **patching** comes in
  - When the kernel needs to move physical memory
  - We can no longer rely on page tables to ensure pointers are correct
  - Pointers need to be updated by the kernel using information we tracked

KARAT: Runtime Mechanics --- Allocations

- An array is initialized, variable **x** points to it
- **x** is an **allocation** --- is tracked by our runtime

x	Length = 100 Escapes = { }
...	...

Allocation Map

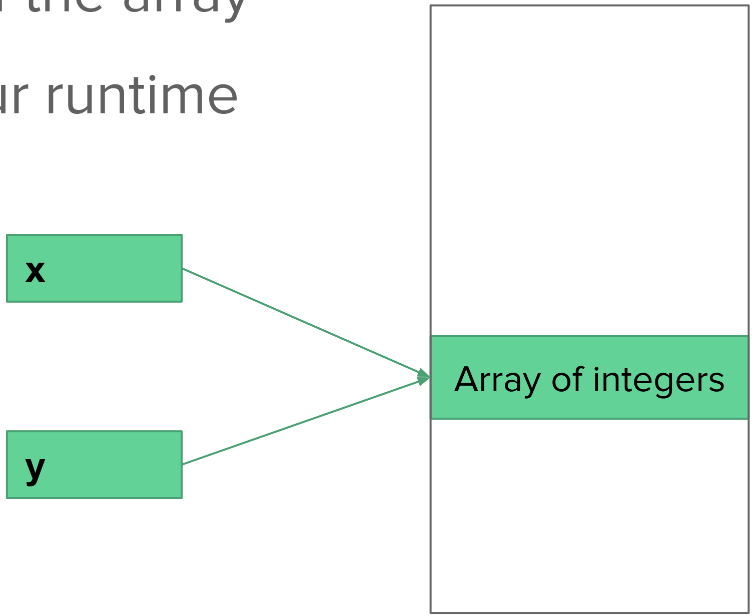


KARAT: Runtime Mechanics --- Escapes

- Variable **y** is a pointer to part of the array
- **y** is an **escape** --- tracked by our runtime

x	Length = 100 Escapes = { **y** }
…	…

Allocation Map



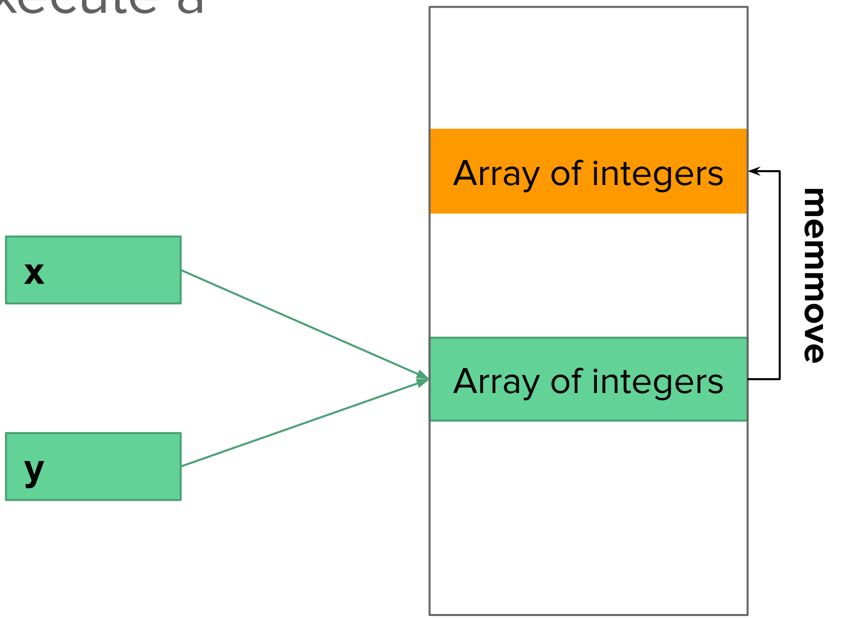
Heap

KARAT: Runtime Mechanics --- Moves and Patches

- Let's say the kernel needs to execute a **move** for the array of integers

x	Length = 100 Escapes = { **y** }
...	...

Allocation Map



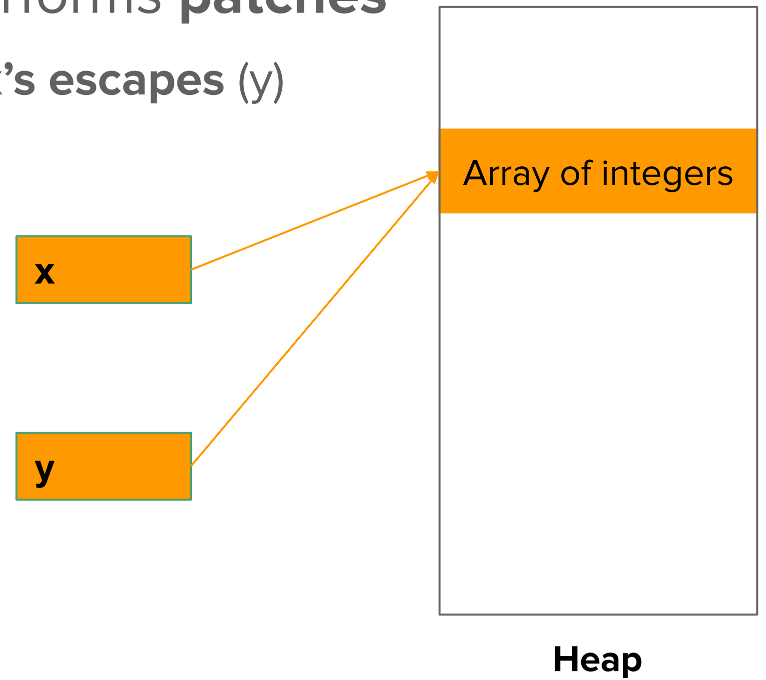
Heap

KARAT: Runtime Mechanics --- Moves and Patches

- After the move, the runtime performs **patches**
  - What **x** is pointing to AND what **x's escapes** (y) point to

x	Length = 100 Escapes = { **y** }
...	...

Allocation Map



KARAT: Runtime Mechanics

- Porting from C++ to C is complicated
- A lot of data structures that exist in the C++ STL **don’t exist** in C or in Nautilus
  - We built them ourselves --- **sets** and **maps** --- using **skiplists**
- We also have to confirm that the runtime **does not conflict** with Nautilus’ buddy allocator system

Similar methods exist for **`realloc`**, **`calloc`**, etc.

`AddToAllocationTable`

`RemoveFromAllocationTable`

`AddToEscapeTable`

Runtime Invocations

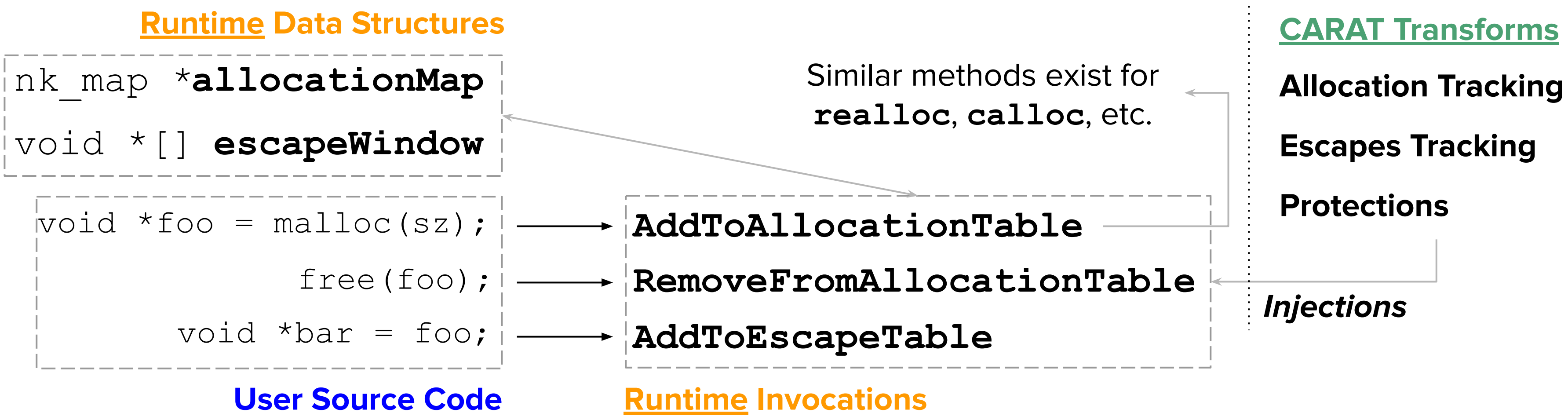
CARAT Transforms

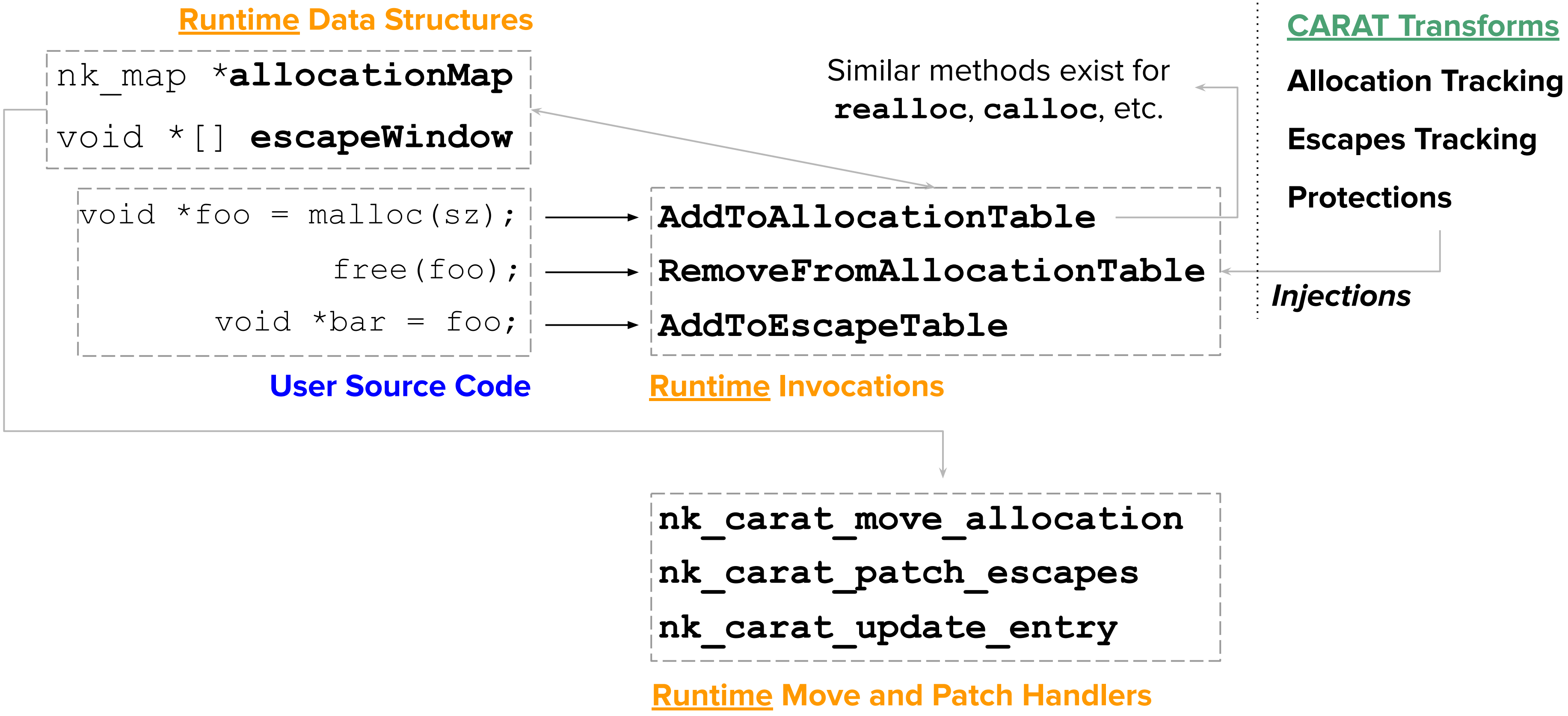
Allocation Tracking

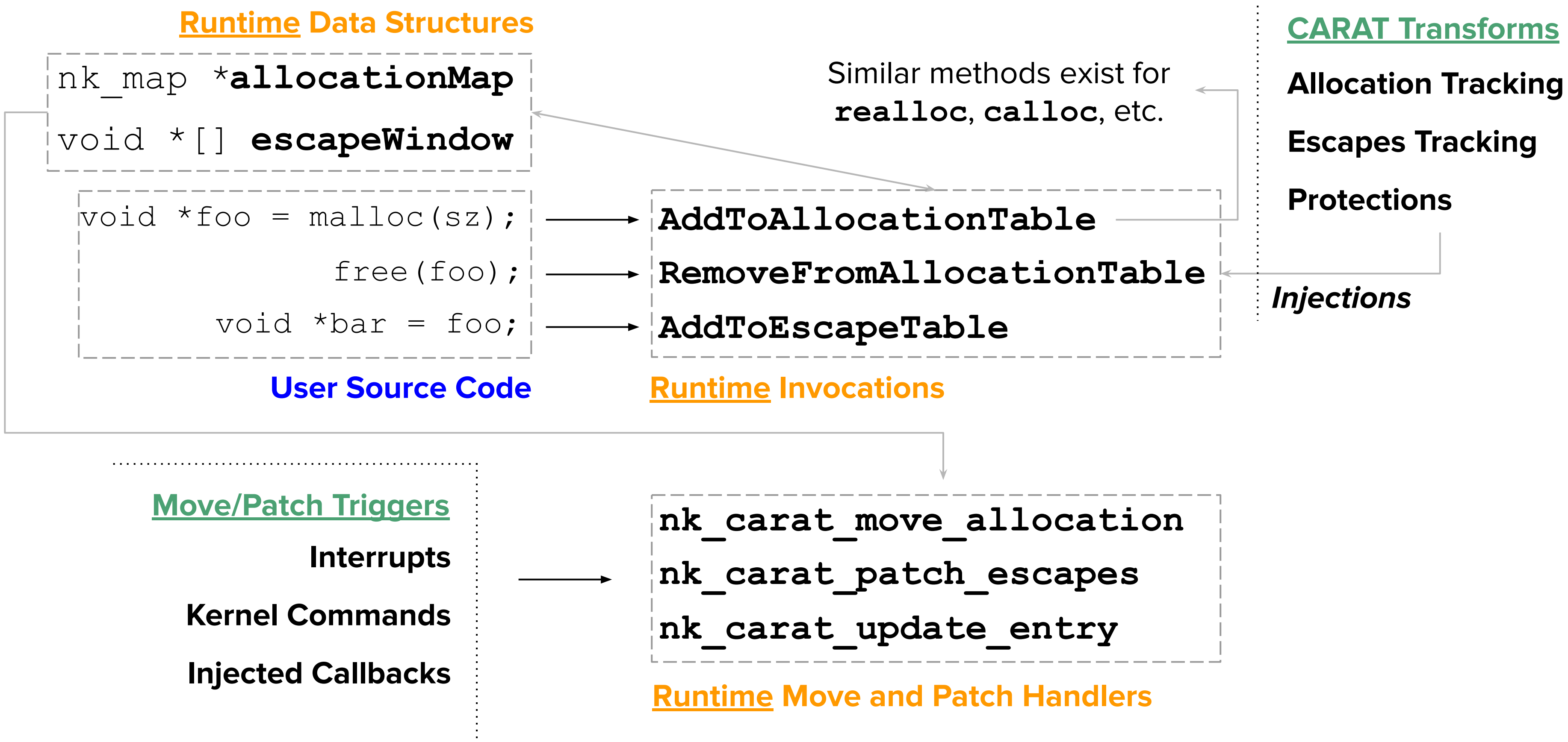
Escapes Tracking

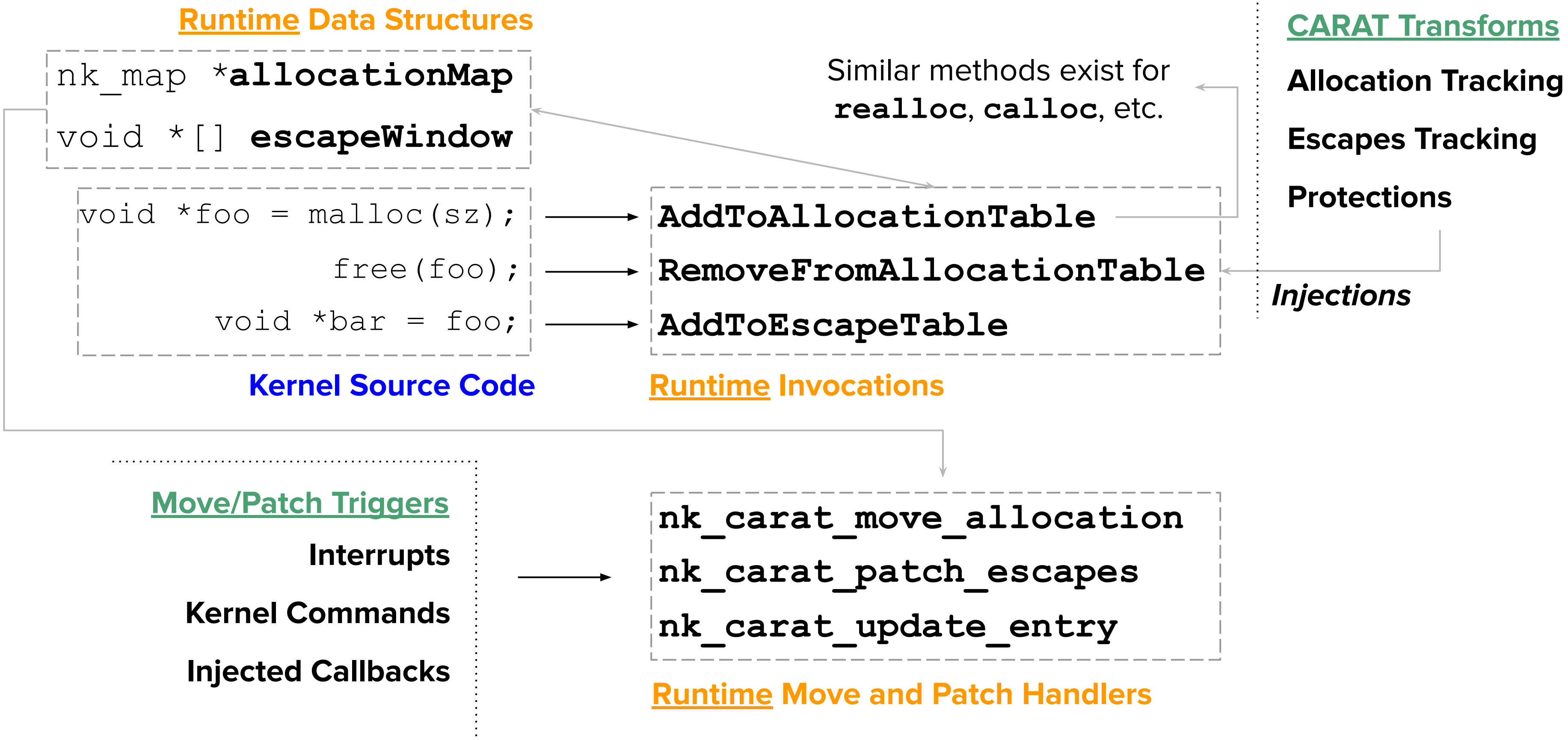
Protections

Injections


Runtime Data Structures
nk_map *allocationMap
void *[] escapeWindow
Similar methods exist for
realloc, calloc, etc.
CARAT Transforms
Allocation Tracking
Escapes Tracking
Protections
void *foo = malloc(sz);
free(foo);
void *bar = foo;
AddToAllocationTable
RemoveFromAllocationTable
AddToEscapeTable
Injections
User Source Code
Runtime Invocations


Runtime Data Structures
nk_map *allocationMap
void *[] escapeWindow
void *foo = malloc(sz);
free(foo);
void *bar = foo;
User Source Code
AddToAllocationTable
RemoveFromAllocationTable
AddToEscapeTable
Runtime Invocations
Similar methods exist for
realloc, calloc, etc.
CARAT Transforms
Allocation Tracking
Escapes Tracking
Protections
Injections
nk_carat_move_allocation
nk_carat_patch_escapes
nk_carat_update_entry
Runtime Move and Patch Handlers


Runtime Data Structures
nk_map *allocationMap
void *[] escapeWindow
void *foo = malloc(sz);
free(foo);
void *bar = foo;
User Source Code
AddToAllocationTable
RemoveFromAllocationTable
AddToEscapeTable
Runtime Invocations
Similar methods exist for
realloc, calloc, etc.
CARAT Transforms
Allocation Tracking
Escapes Tracking
Protections
Injections
Move/Patch Triggers
Interrupts
Kernel Commands
Injected Callbacks
nk_carat_move_allocation
nk_carat_patch_escapes
nk_carat_update_entry
Runtime Move and Patch Handlers


Runtime Data Structures
nk_map *allocationMap
void *[] escapeWindow
Similar methods exist for
realloc, calloc, etc.
CARAT Transforms
Allocation Tracking
Escapes Tracking
Protections
void *foo = malloc(sz);
free(foo);
void *bar = foo;
AddToAllocationTable
RemoveFromAllocationTable
AddToEscapeTable
Injections
Kernel Source Code
Runtime Invocations
Move/Patch Triggers
Interrupts
Kernel Commands
Injected Callbacks
nk_carat_move_allocation
nk_carat_patch_escapes
nk_carat_update_entry
Runtime Move and Patch Handlers

Outline

- CARAT --- Overview
- Kernel CARAT --- “KARAT”
    - CARAT in Nautilus
    - Compiler Mechanics
    - Runtime Mechanics
- **Next Steps**

Next Steps

- Compilation
  - Finish up the **protections pass**
- Runtime
  - Build a more realistic policy for Nautilus to move memory and test with KARAT --- involving **interrupts** or **injected callbacks**
- Write benchmarks in Nautilus to test a working version of KARAT
- Run Nautilus with KARAT on bare metal (KNL Phi, R415, etc.)