Decker: Attack Surface Reduction via On-Demand Code Mapping

Chris Porter, Sharjeel Khan, Santosh Pande

ASPLOS 2023

CREATING THE NEXT®

Problem: Code reuse attacks

- Software continues to be susceptible to existing and new code reuse attacks
- **Code reuse attacks** are software attacks that leverage existing code in programs to perform some malicious action
- They're commonly built today with **gadgets**
- **Gadgets** are code snippets that can be stitched together at runtime to form **gadget chains** that execute malicious behavior

ROP chain example

ROP gadgets within the .text section

.text

```
0x500000
     ...
0x5001f9  pop rcx
     ...  ret

0x52e1fe  mov qword ptr [rcx],
     ...  rax
          ret
0x55720a
     ...  pop rax
          ret
0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9  pop rcx
     ...  ret

0x52e1fe  mov qword ptr [rcx],
     ...  rax
          ret
0x55720a
     ...  pop rax
          ret
0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret
0x55720a
     ...   pop rax
           ret
0x600000
```

gadget 1 address -> pop rax; ret;

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret

0x55720a
     ...   pop rax
           ret

0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret

0x55720a
     ...   pop rax
           ret

0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret

0x55720a
     ...   pop rax
           ret

0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret

0x55720a
     ...   pop rax
           ret

0x600000
```

ROP chain example

Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe
```

.text

0x500000	
...	
0x5001f9	pop rcx ret
...	
0x52e1fe	mov qword ptr [rcx], rax ret
...	
0x55720a	pop rax ret
...	
0x600000	

ROP chain example

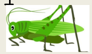
Stack payload

```
0x55720a
0x2a
0x5001f9
0x800000
0x52e1fe   <-
```

.text

```
0x500000
     ...
0x5001f9   pop rcx
     ...   ret

0x52e1fe   mov qword ptr [rcx],
     ...   rax
           ret
0x55720a
     ...   pop rax
           ret
0x600000
```



ROP chain example

- Summary: gadget chaining works by leveraging multiple snippets across the existing code.

Debloating as defense

- We know that software is bloated with unused code
  - The unneeded code contains gadgets which could be chained.
  - Why not remove it?
- Debloating is a proposed defense against code reuse attacks
- Shortcomings with current approaches
  - **Too conservative**, leaving too much code available to attackers
  - Or they **compromise soundness** by specializing the application for only certain inputs or features, which can lead to crashes or incorrect output

Soundness and may-use code

<u>Definition</u>

sound transformation: a program transformation that does not change the semantics of a program. Program transformations that induce crashes or cause incorrect output are unsound.

<u>Definition</u>

may-use code: code that may be used by the program under certain inputs or execution conditions.

Properties of debloating frameworks

	Piece-wise	Chisel	Razor	BlankIt
Works on application		✔	✔	
Works on library	✔		✔	✔
Works on binary			✔	✔
No user input needed	✔			✔
No training needed	✔		✔	
Is sound	✔			✔
Can debloat may-use code				✔

Properties of debloating frameworks

Georgia Tech

	Piece-wise	Chisel	Razor	BlankIt
Works on application		✔	✔	
Works on library	✔		✔	✔
Works on binary			✔	✔
No user input needed	✔			✔
No training needed	✔		✔	
Is sound	✔			✔
Can debloat may-use code				✔

Properties of debloating frameworks

	Piece-wise	Chisel	Razor	BlankIt
Works on application		✔	✔	✗
Works on library	✔		✔	✔
Works on binary			✔	✔
No user input needed	✔			✔
No training needed	✔		✔	
Is sound	✔			✔
Can debloat may-use code				✔

Motivation

Georgia Tech

To the best of our knowledge, there is no general technique today that:

1. Works on the applications as a whole instead of libraries
2. Is sound
3. Can effectively debloat may-use code using dynamic contexts

Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion

Decker overview

- Decker is a ***constructive*** attack surface reduction technique
- **“Decks”** are active sections of code that the program can effectively stand on. When a deck is unneeded, it can be removed.
- Map only code that is currently needed by a running program
- Disable all other code so accesses trigger a runtime exception
- Granularity: implement with system code pages (4KB)
- Compiler and runtime component to it

Threat model

- **Focus is on attack surface reduction**
  - Assume the attacker can initiate and propagate the attack
  - Decker’s main goal
    - Incrementally expose the executable surface of a program by following its interprocedural control flow.
    - Breaks chains of gadgets, because all the gadgets that compose a chain are never dynamically exposed at the same time.

- **Integrity of indirect call targets is out of scope** (orthogonal schemes like CFI and CPI are designed specifically to tackle it)

Decker Example

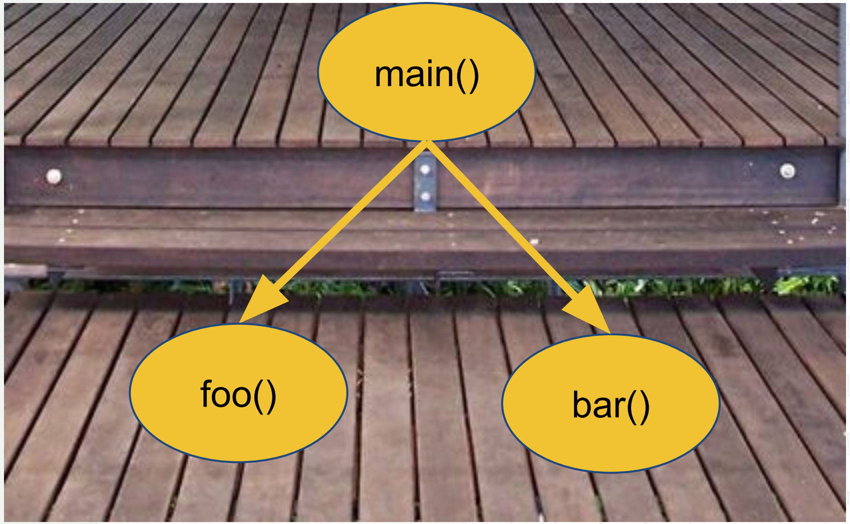
Image source: Wikipedia (deck)

Decker Example

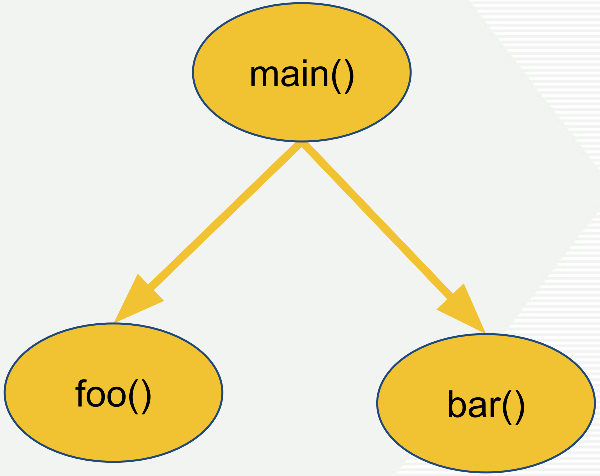
A deck

Image source: Wikipedia (deck)

Decker Example



Decker Example

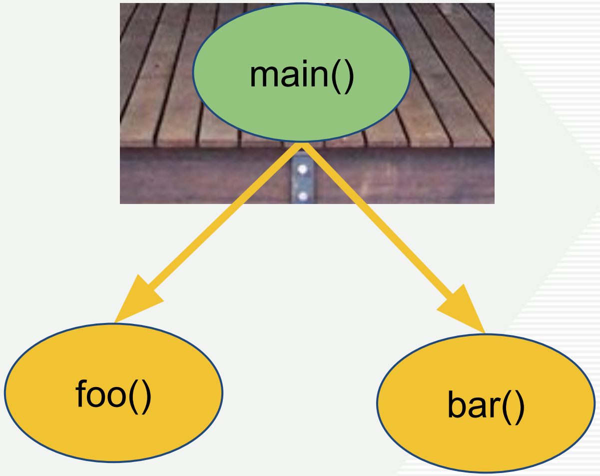



```
main()
   ...
   if(x)
      foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example



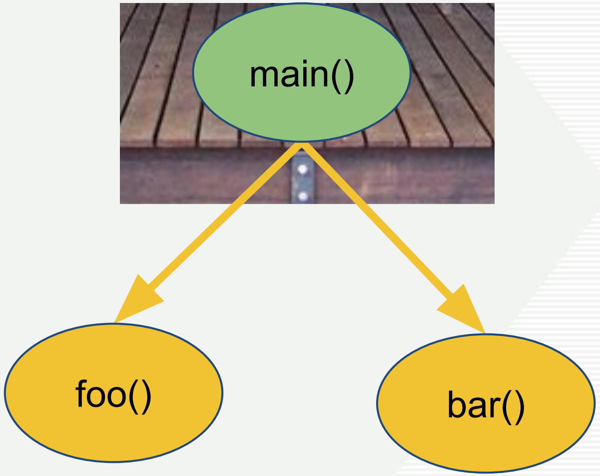
Program counter →

```
main()
  ...
  if(x)
    foo()
  bar()

foo()
  ...

bar()
  ...
```

Decker Example

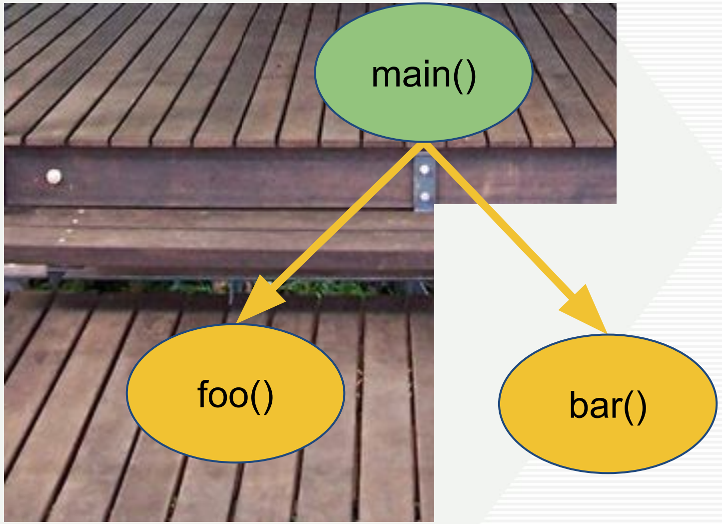



```
main()
   ...
   if(x)
     foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

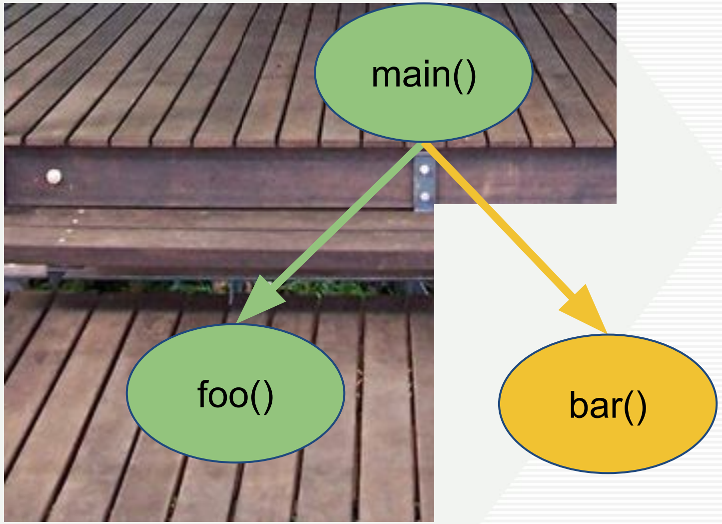



```
main()
   ...
   if(x)
     foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

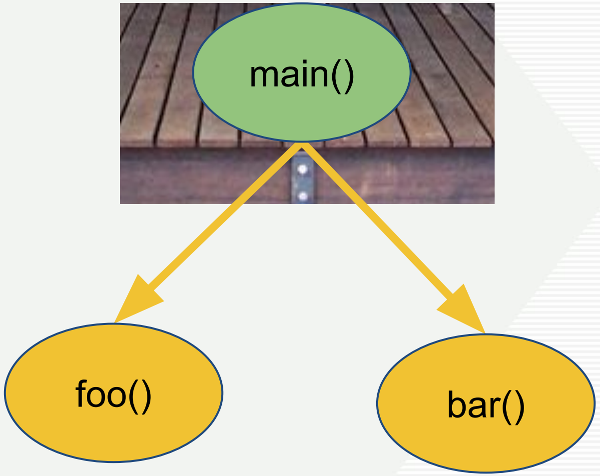



```
main()
   ...
   if(x)
      foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

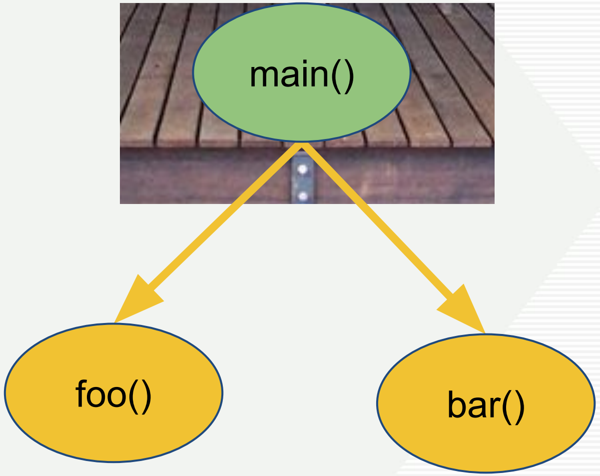



```
main()
   ...
   if(x)
     foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

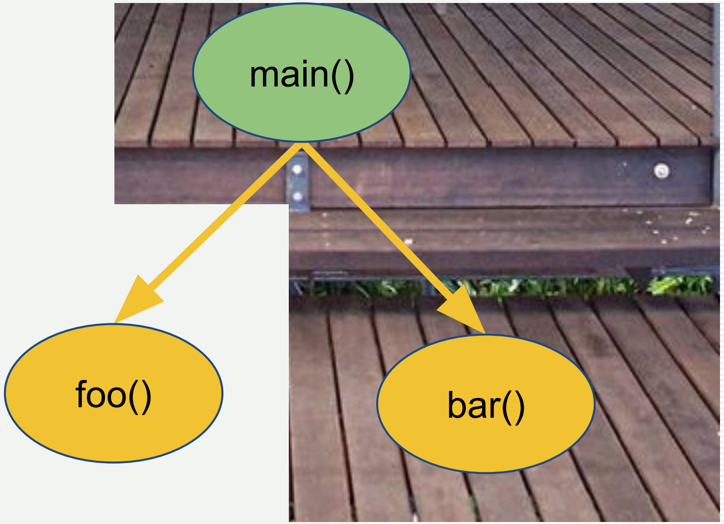



```
main()
   ...
   if(x)
      foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

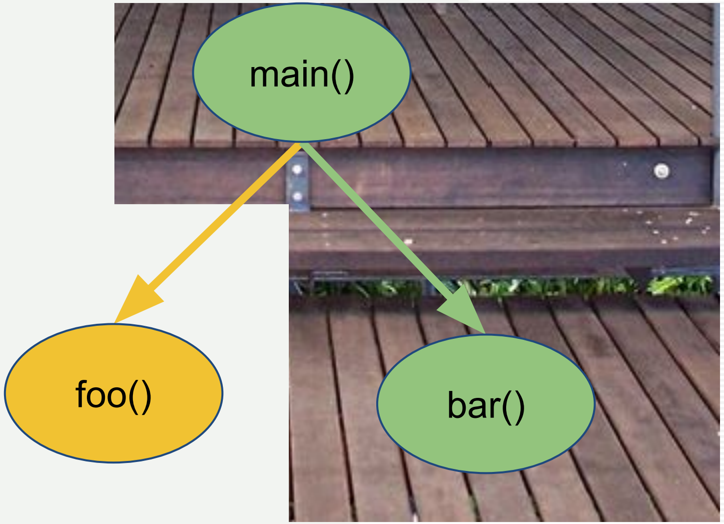



```
main()
   ...
   if(x)
      foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

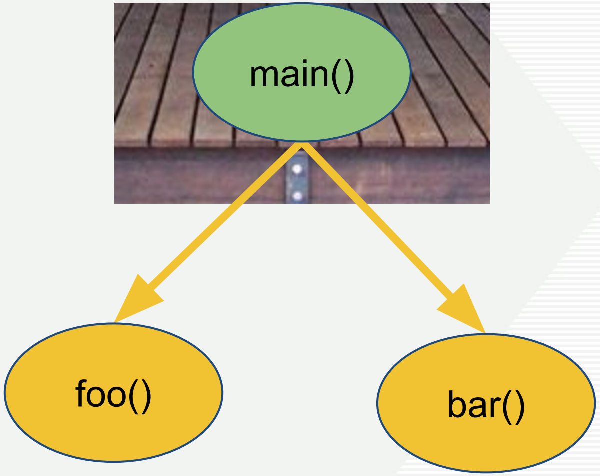



```
main()
   ...
   if(x)
     foo()
   bar()

foo()
   ...

bar()
   ...
```

Decker Example

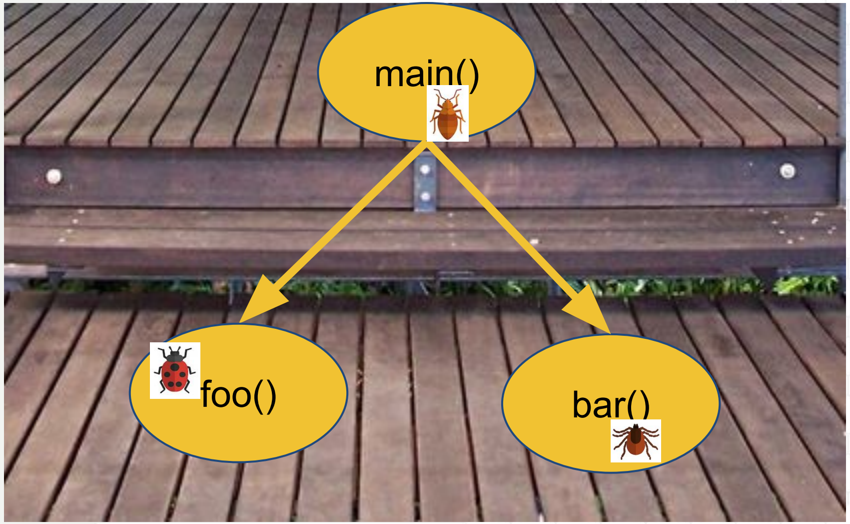



```
main()
   ...
   if(x)
     foo()
   bar()

foo()
   ...

bar()
   ...
```

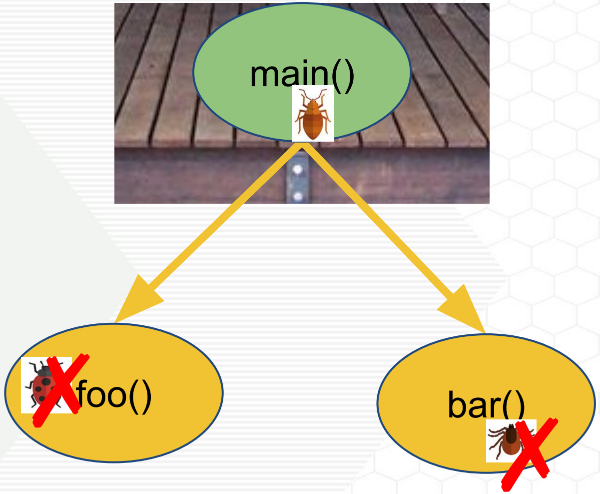
Example (Unprotected Program)



Gadget chain components: 

Gadget chain possible? ✔

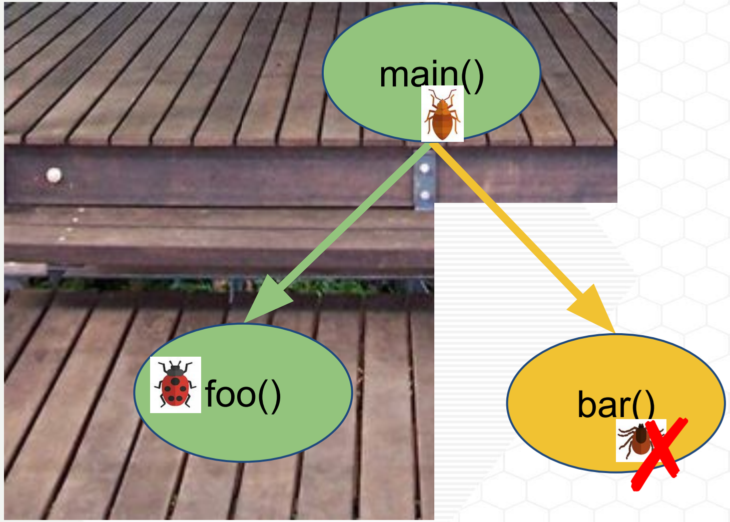
Example (Decker-Protected Program)



Gadget chain components:

Gadget chain possible?

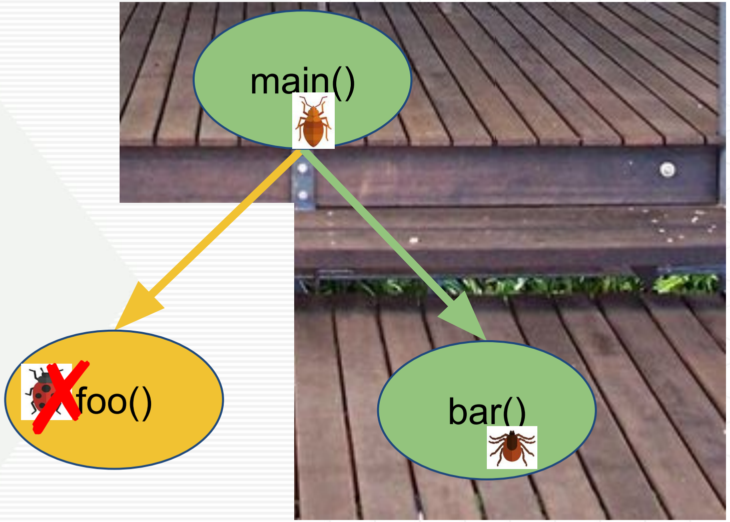
Example (Decker-Protected Program)



Gadget chain components:

Gadget chain possible?

Example (Decker-Protected Program)



Gadget chain components:

Gadget chain possible?

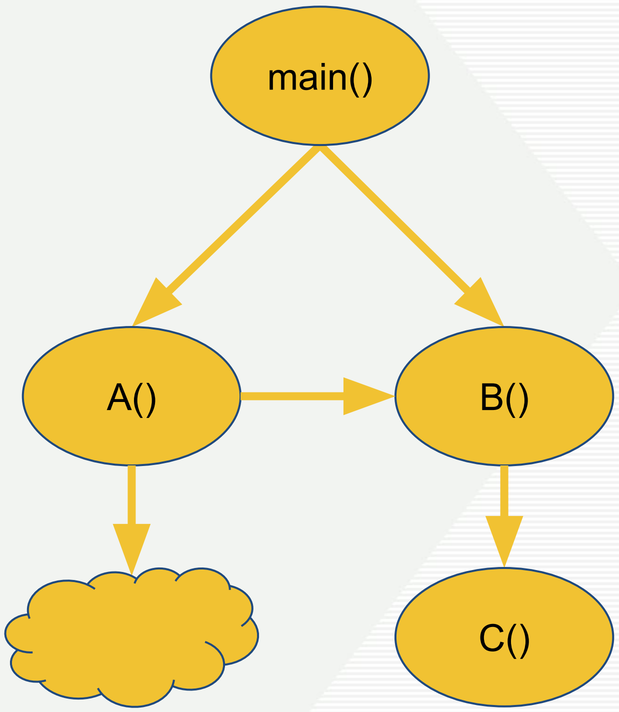
Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion

Program structure and performance

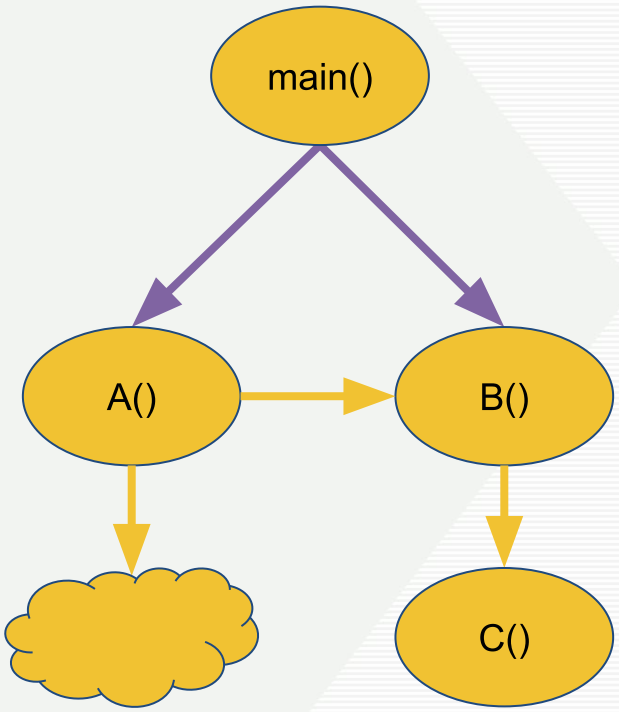
- Ideally, decking should be done at the entrances and exits of callsites
  - But such instrumentation can cause substantial runtime overhead
- *Loops* in particular are problematic
  - Instrumentation for functions invoked inside of loops will execute repeatedly and kill performance
- Leads us to 4 types of decks

Example



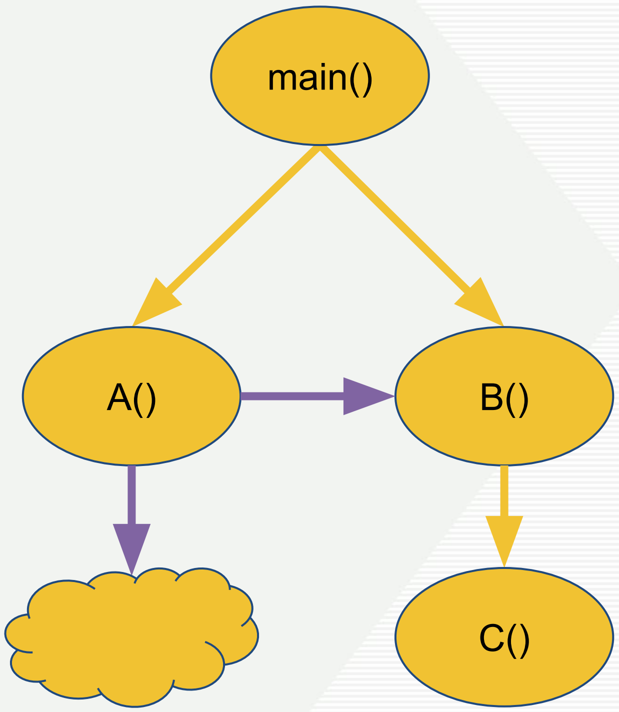

```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

Example



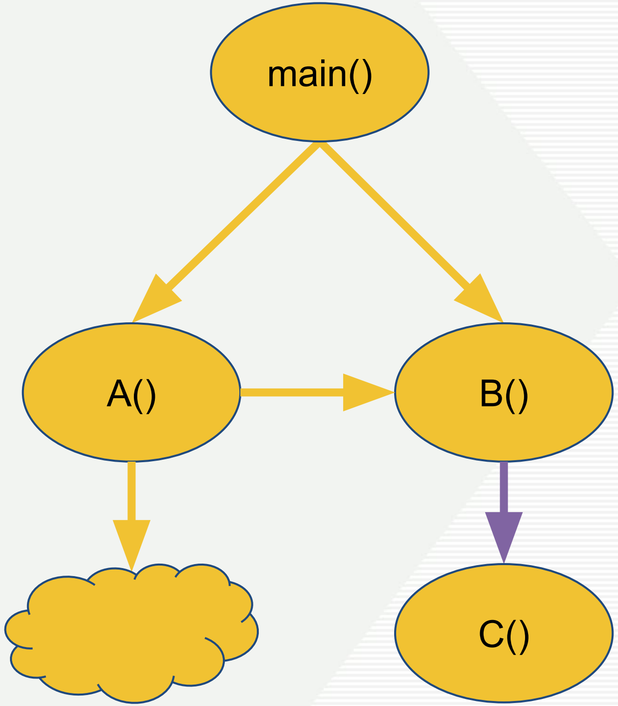

```
main()
    A(func_ptr)
    while {
      B()
    }
A(func_ptr)
    B()
    func_ptr()
B()
    C()
```

Example



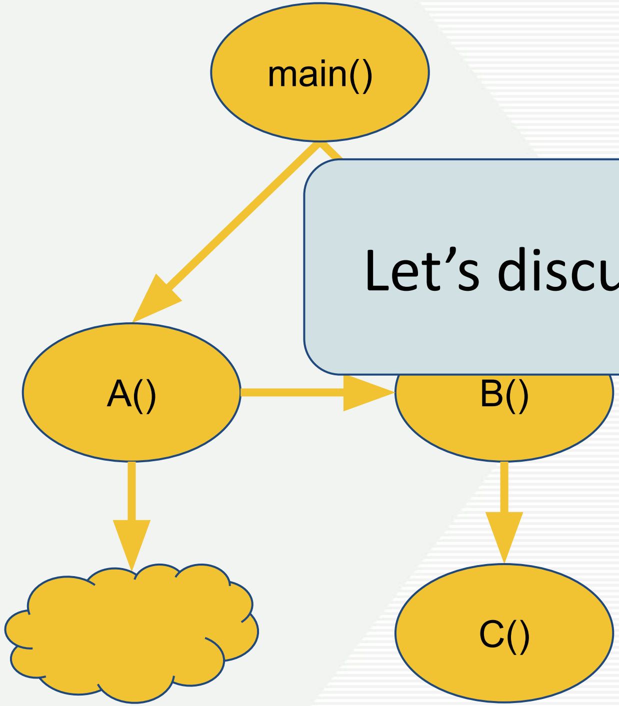

```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

Example




```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

Example



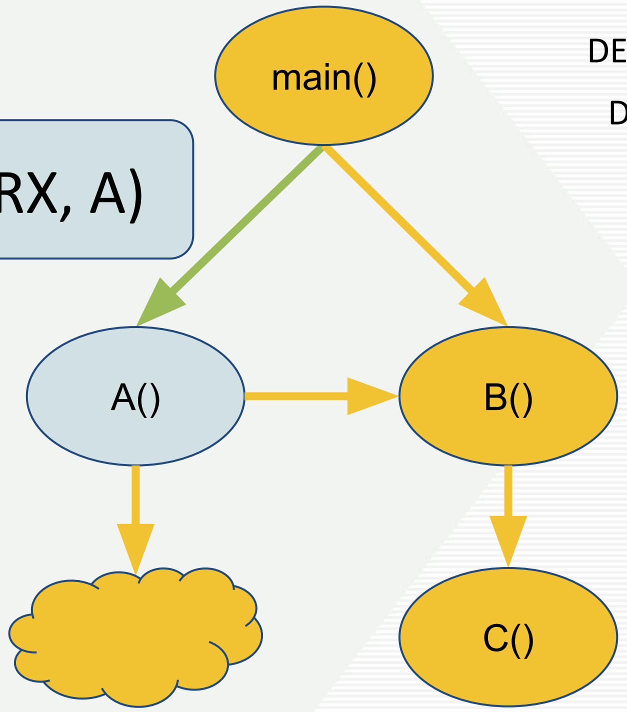

```
main()
   A(func_ptr)
   while {
```

Let's discuss the 4 deck types!

```
   B()
   func_ptr()
B()
   C()
```

1. Single deck - Occurs in a non-loop region

map(RX, A)

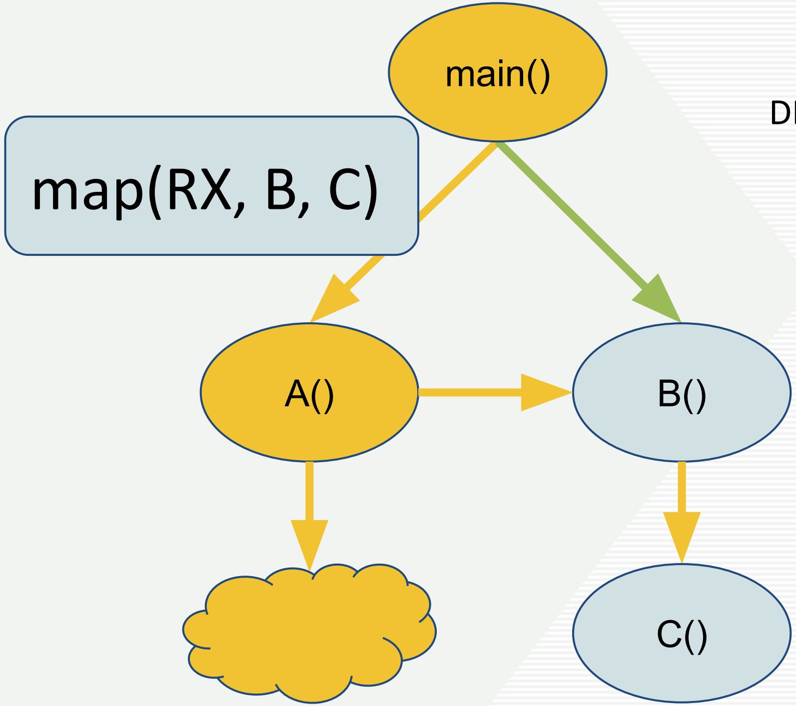


DECK-START

DECK-END

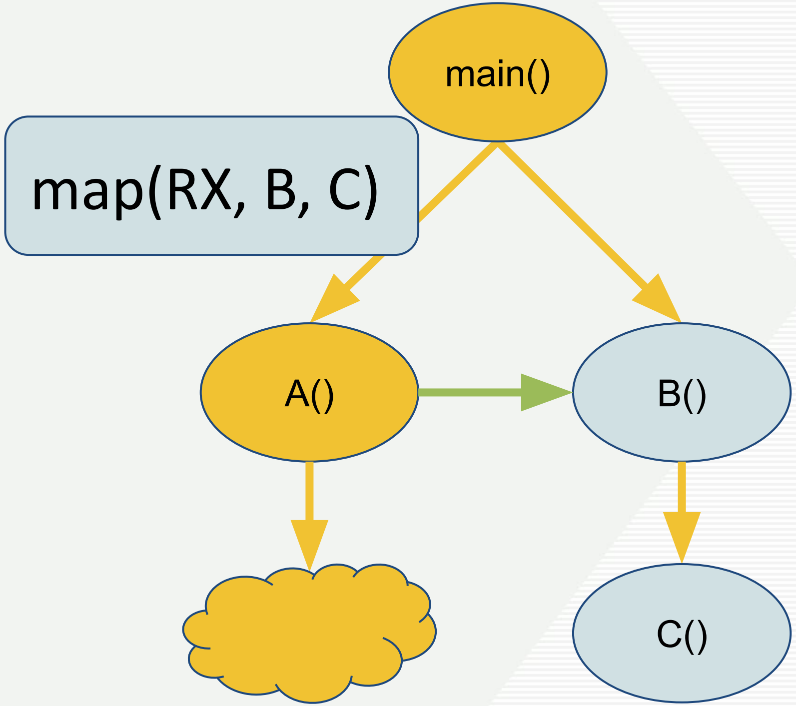
```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

2. Loop deck - Entrance to loops




```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

3. Reachable deck - Enter loop region via non-loop

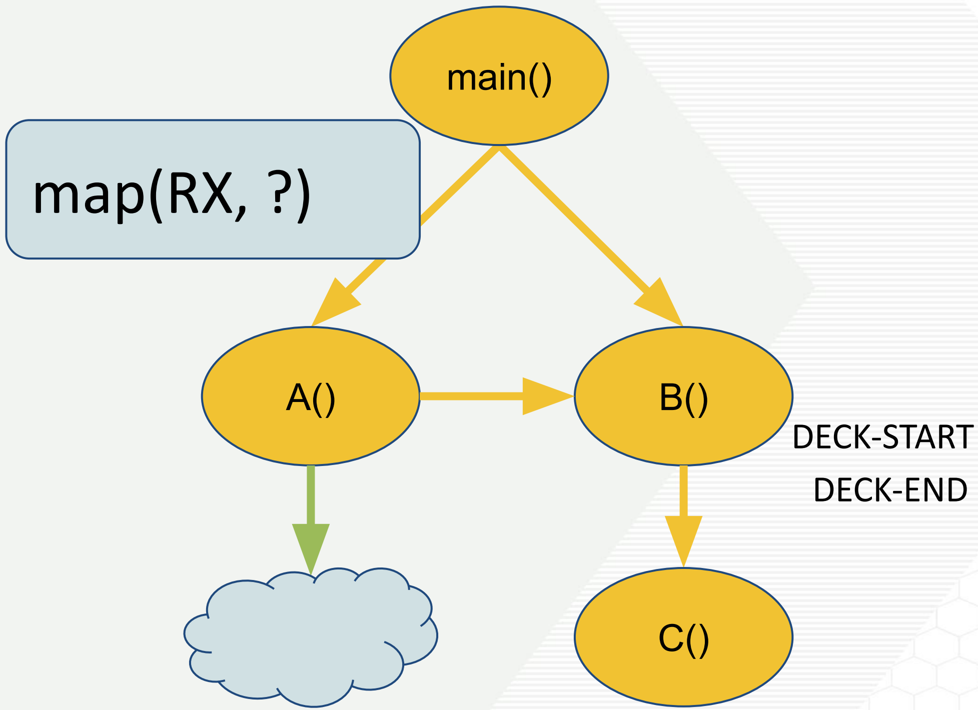



```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

DECK-START → B()

DECK-END → func_ptr()

4. Indirect deck - Occurs at indirect calls




```
main()
   A(func_ptr)
   while {
     B()
   }
A(func_ptr)
   B()
   func_ptr()
B()
   C()
```

Indirect deck

- Static function pointer analysis?
  - Narrows possible targets but is an overapproximation
  - Limits attack surface reduction
- Instead, resolve this at runtime by
  - Passing function pointer to Decker runtime
  - Mapping the appropriate pages
- We will violate the rule of thumb and instrument inside of loops!
  - Caching optimization needed

Linker

- Problem: Functions in one deck can occupy the same code page as functions in another deck, and we do not want to inadvertently include gadgets from other functions when we map a code page as active.
- Solution: Separate deck sets into disjoint sets (separate pages) and use a linker script to enforce it.

Outline

1. Introduction
2. Overview
3. Decking
4. Evaluation
5. Conclusion

Evaluation goals

- What is the performance slowdown due to Decker?
- What is the gadget reduction for applications that use Decker?
- Can Decker
  - break real gadget chains in the benchmarks and real-world applications to be able to stop gadget-based attacks?
  - render JOP gadgets ineffective in practical scenarios, including Windows?

Evaluation summary

- Performance
  - ~5% average overhead for SPEC 2017, GNU coreutils, and nginx
- Security
  - 70-87% total gadget reduction
  - Equals or (in many cases) improves on comparable prior work
  - Achieves this without compromising soundness

Evaluation summary

- Gadget chain-breaking
  - An unexplored metric that we introduce and report for Decker
  - Decker fully breaks a popular syscall ROP chain in all cases
  - Decker removes functionality which is critical for JOP chains to be successful from nginx in both Linux and Windows

Evaluation details

- Please see paper for detailed results and analysis!

Conclusion

- Decker
  - an attack surface reduction technique for applications
  - is sound and enables may-use code on-demand
  - requires zero training, user inputs, or specifications
- Acceptable performance slowdowns and strong total gadget reductions across two benchmark suites and nginx
- Gadget chain-breaking study
  - Breaks automatically generated syscall ROP chains
  - Strong evidence that useful JOP chains are substantially hampered or impossible to build for both Linux and Windows

Thank you!

- Check out our artifact!
  - https://zenodo.org/record/7319957
  - Link is available in the paper, too

- **Questions?**