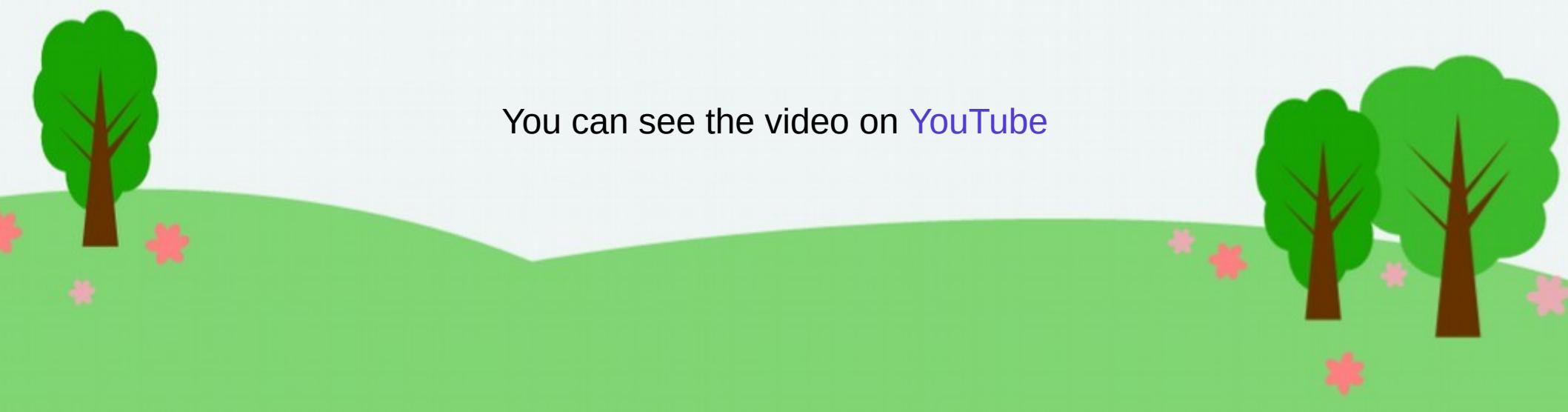




# Using capability to implement memory bounds checking

Wuyang Chung

wy-chung@outlook.com



You can see the video on [YouTube](#)

# Outline

- Object-based vs pointer-based approach
- Capability-based approach
- Temporal safety
- The principle of exclusive thread local data
- Conclusion



# Object-based vs pointer-based

	Object-based	Pointer-based (Fat pointer)
Compatibility	v	Pointer format changed
Cost to get the metadata	A lot of memory or CPU time	Low
Spatial safety	v	v
Temporal safety	Quarantine	Garbage collection Lock-and-key
Principle of intentional use	x	v
Principle of exclusive thread local data	x	?

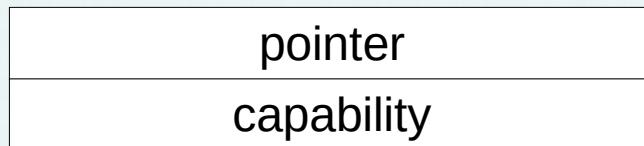


# Capability-based approach

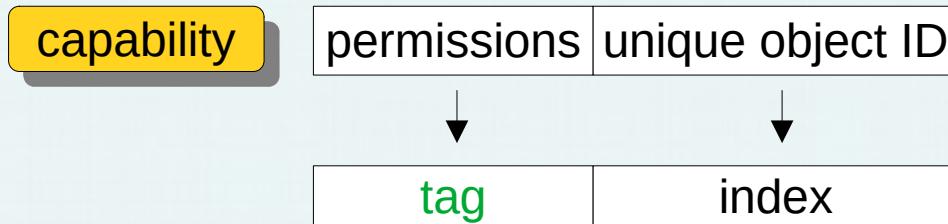
	Object-based	Pointer-based (Fat pointer)	Capability-based
Compatibility	v	Pointer format changed	Pointer format changed
Cost to get the metadata	A lot of memory or CPU time	Low	Low
Spatial safety	v	v	v
Temporal safety	Quarantine	Garbage collection Lock-and-key	Capability revocation
Principle of intentional use	x	v	v
Principle of exclusive thread local data	x	?	v

# Capability-based approach

Pointer format



Object metadata table



tag	start	end

- The most important thing in a capability system is to prevent capability from being forged
  - Tagging
  - Capability is valid only when the tag in the capability is equal to the tag in the metadata
  - The tag must have at least 4 bits
    - The probability of false negative will be lower with more bits in the tag



# Capability-based approach

Pointer format

pointer	
tag	index

Object metadata table

bounds check:

```
cap.tag == meta.tag &&  
pointer >= meta.start && pointer + size <= meta.end
```

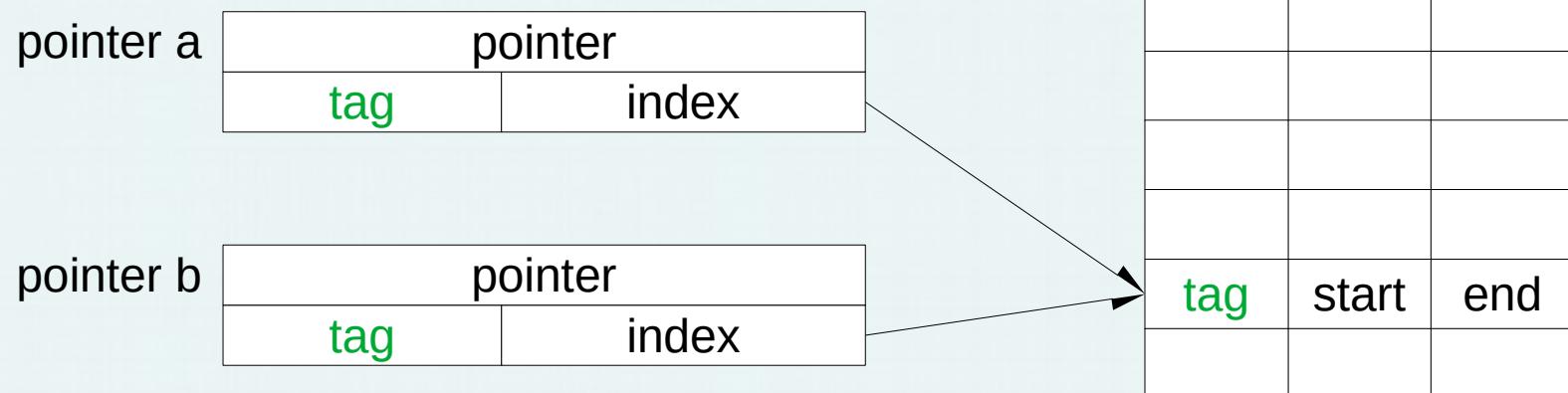
tag	start	end

- The most important thing in a capability system is to prevent capability from being forged
  - Tagging
  - Capability is valid only when the tag in the capability is equal to the tag in the metadata
  - The tag must have at least 4 bits
    - The probability of false negative will be lower with more bits in the tag



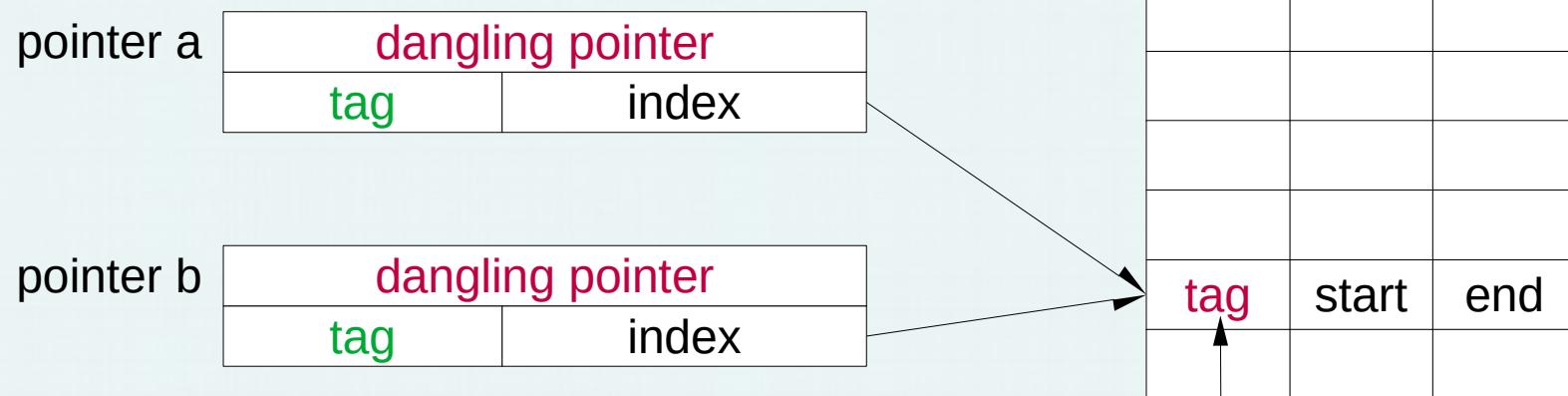
# Temporal safety

- Uses capability revocation to detect
  - use-after-free, double-free, stack-use-after-return and stack-use-after-scope bug



# Temporal safety

- Uses capability revocation to detect
  - use-after-free, double-free, stack-use-after-return and stack-use-after-scope bug



Object metadata table



The tag is changed  
when the object is freed

# Global and local metadata table

- A process has one global metadata table
  - Used to store the metadata for objects on global variable, heap or constant data
  - Need a lock to protect this table
- Each thread in a process has a local metadata table
  - Used to store the metadata for objects on stack or TLS
  - Do not need a lock for this table



# Global and local metadata table

Pointer format

capability

pointer		
tag	G/L	Index

G

L

global  
metadata table

tag	start	end

tag	start	end

local  
metadata table



# Principle of exclusive thread local data

- When thread A tries to dereference a pointer with a capability that points to thread B's local metadata table, it will trigger a fault because the tag will not be equal

capability

pointer		
tag	L	Index

tag	start	end

thread A's local metadata table

tag	start	end

thread B's local metadata table



# Conclusion

- In capability-based approach the pointer is smaller than the fat pointer based approach
- It can get the object metadata with very low overhead
- It can detect temporal safety violations by using capability revocation
- When doing memcpy, there is no need to propagate the metadata for the pointers in the buffer since the metadata is in a separate table



# Conclusion

- To support bounds checking for objects on stack
  - In function prologue, the compiler will insert the metadata of objects on the stack into the local metadata table
  - In function epilogue, the compiler will remove those metadata from the local metadata table



# The end

## Thank you

