Tightly Seal your Sensitive Pointers with PACTight

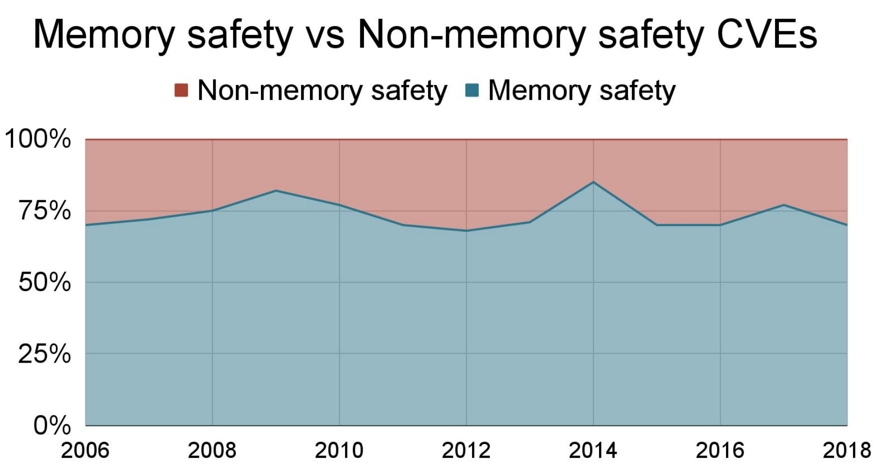
Mohannad Ismail (Virginia Tech), Andrew Quach (Oregon State University), Christopher Jelesnianski (Virginia Tech), Yeongjin Jang (Oregon State University), Changwoo Min (Virginia Tech)

ARM is becoming popular!

- More and more servers, data centers and high-performance computers are using ARM.
- Greater importance to have effective and efficient defenses for ARM in these environments.

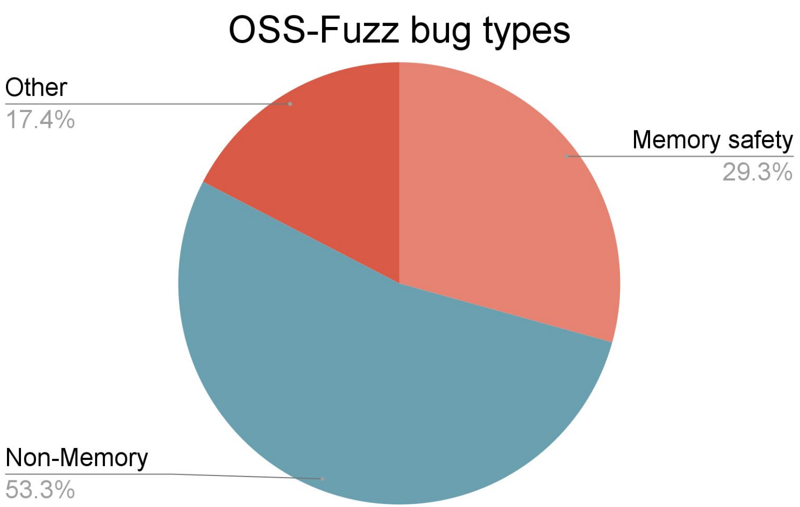
ARM®

Memory safety is a serious problem!



Microsoft Product CVEs

https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/



Google OSS (Open Source Software) Fuzz bugs

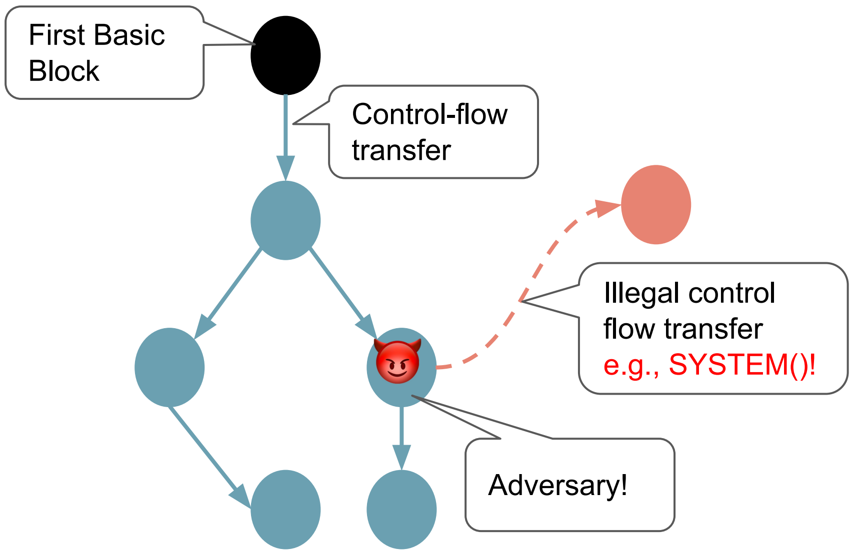
https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

Outline

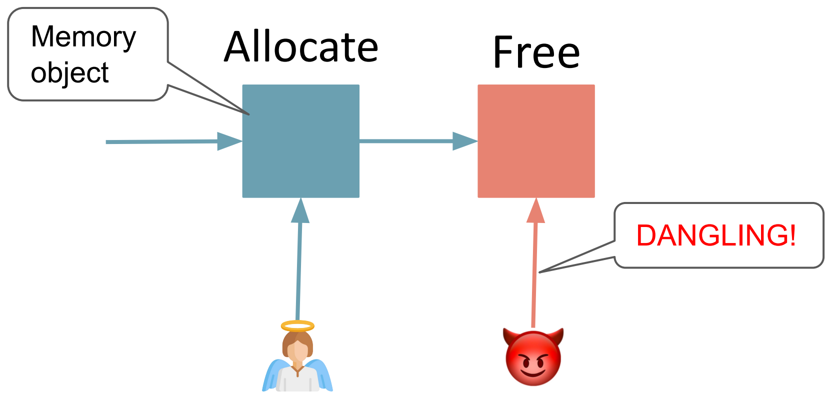
- Introduction
- **Background and related work**
- Introducing PACTight
- PACTight design
- PACTight defense mechanisms
- Evaluation
- Conclusion

Control-flow hijacking and use-after free attacks are critical!

Control-flow hijacking and use-after-free attacks are dangerous memory corruption attacks



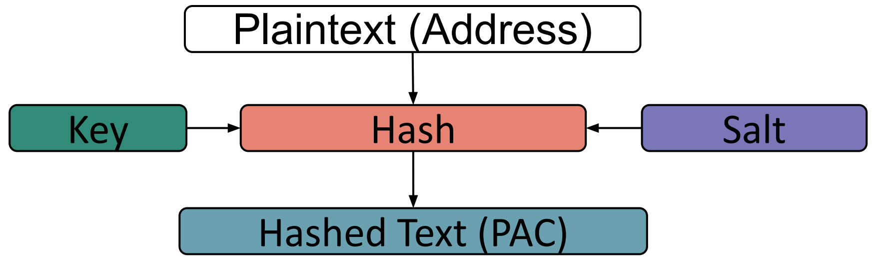
Control-flow hijacking



Use-after free

ARM Pointer Authentication

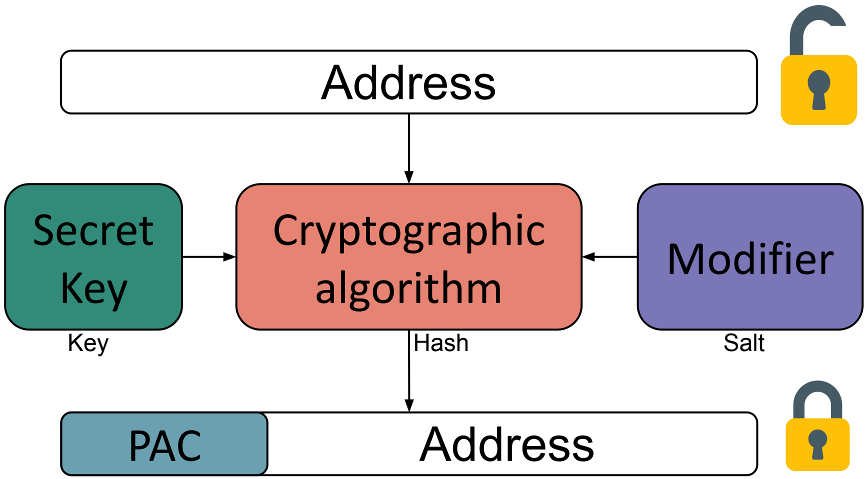
- Pointer Authentication Code (PAC) is generated by a cryptographic hash function.



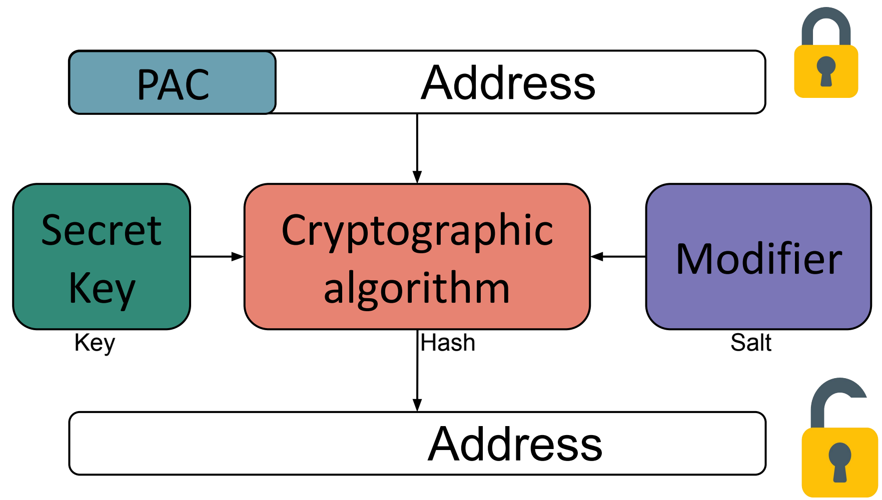
- The PAC is then placed on the unused bits of the 64-bit pointer.

ARM Pointer Authentication

- **<u>PAC signing:</u>** The algorithm takes the pointer and modifier, as well as a key, and generates a PAC.

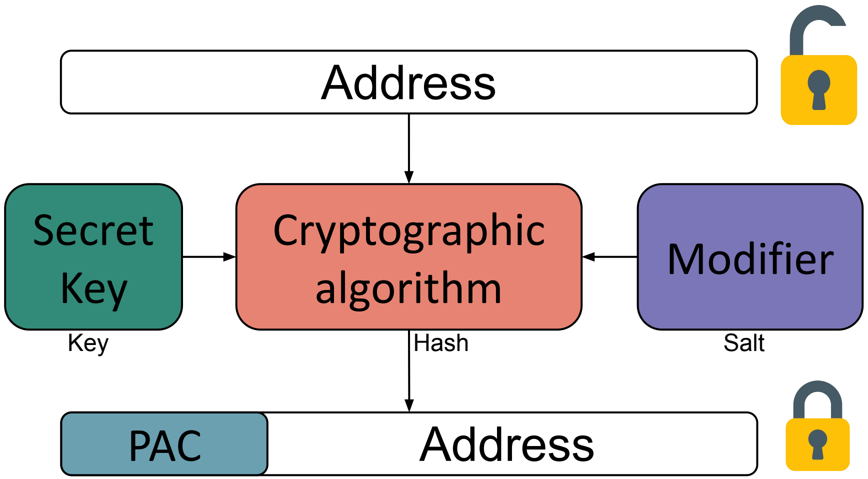


- **<u>PAC authentication:</u>** The algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer.

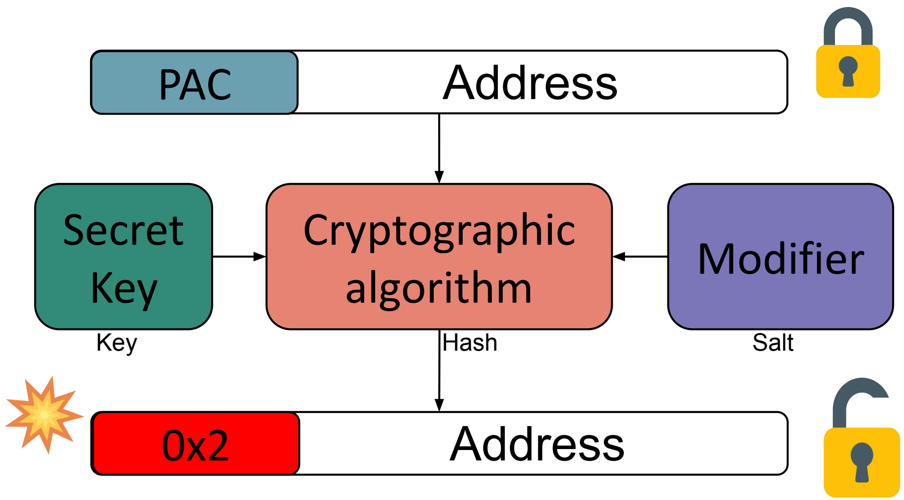


ARM Pointer Authentication

- **PAC signing:** The algorithm takes the pointer and modifier, as well as a key, and generates a PAC.



- **PAC authentication:** The algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer.



Current state-of-the-art PAC techniques

	Protection scope	**PAC modifier**
PARTS-CFI Lilijestrand et. al (USENIX SEC'19)	Return addresses and indirect code pointers	SP (Stack Pointer) + function id for return addresses and type id for indirect code pointers.
PACStack Lilijestrand et. al (USENIX SEC'21)	Return addresses	Previous chained return address on the stack
PTAuth Farkhani et. al (USENIX SEC'21)	Heap allocated objects	A generated object-id

Current state-of-the-art PAC techniques

	Protection scope	PAC modifier
PARTS-CFI Lilijestrand et. al (USENIX SEC'19)	Return addresses and indirect code pointers	SP (Stack Pointer) + function id for return addresses and type id for indirect code pointers.
PACStack Lilijestrand et. al (USENIX SEC'21)	Return addresses	Previous chained return address on the stack
PTAuth Farkhani et. al (USENIX SEC'21)	Heap allocated objects	A generated object-id

Current state-of-the-art PAC techniques

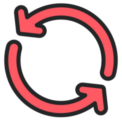
	Protection scope	PAC modifier
PARTS-CFI Lilijestrand et. al (USENIX SEC'19)	Return addresses and indirect code pointers	SP (Stack Pointer) + function id for return addresses and type id for indirect code pointers.
PACStack Lilijestrand et. al (USENIX SEC'21)	Return addresses	Previous chained return address on the stack
PTAuth Farkhani et. al (USENIX SEC'21)	Heap allocated objects	A generated object-id

Current state-of-the-art PAC techniques

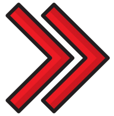
	Protection scope	**PAC modifier**
PARTS-CFI Lilijestrand et. al (USENIX SEC'19)	Return addresses and indirect code pointers	SP (Stack Pointer) + function id for return addresses and type id for indirect code pointers.
PACStack Lilijestrand et. al (USENIX SEC'21)	Return addresses	Previous chained return address on the stack
PTAuth Farkhani et. al (USENIX SEC'21)	Heap allocated objects	A generated object-id

Limitations of state-of-the-art PAC techniques

- Reliance on a modifier that can be **repeated**, thus attackers can **reuse** the PAC generated for one in the context of using the other. [PARTS-CFI SEC'19]



- Reliance on the presence of a **forward-edge CFI technique** with the PAC defense mechanism. [PACStack SEC'21]



- Constrained threat model, defending **only** against attackers with just **arbitrary write**. The defense is not effective **if the attacker has arbitrary read**. [PTAuth SEC'21]

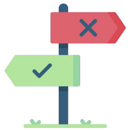
Outline

- Introduction
- Background and related work
- **Introducing PACTight**
- PACTight design
- PACTight defense mechanisms
- Evaluation
- Conclusion

PACTight Overview

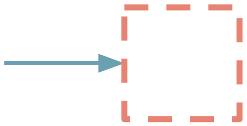
We define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with.

- Unforgeability: A pointer should always point to its legitimate object.



- Non-copyability: A pointer can only be used when it is at its specific legitimate location.

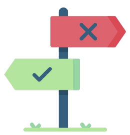
- Non-dangling: A pointer cannot be used after its object has been freed.



PACTight Overview

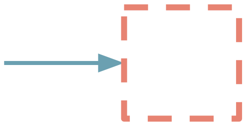
We define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with.

- **Unforgeability**: A pointer should always point to its legitimate object.



- Non-copyability: A pointer can only be used when it is at its specific legitimate location.

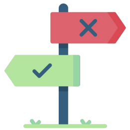
- Non-dangling: A pointer cannot be used after its object has been freed.



PACTight Overview

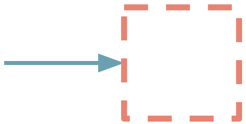
We define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with.

- Unforgeability: A pointer should always point to its legitimate object.



- **<u>Non-copyability</u>**: A pointer can only be used when it is at its specific legitimate location.

- Non-dangling: A pointer cannot be used after its object has been freed.



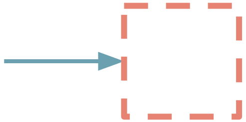
PACTight Overview

We define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with.

- Unforgeability: A pointer should always point to its legitimate object.

- Non-copyability: A pointer can only be used when it is at its specific legitimate location.

- **Non-dangling:** A pointer cannot be used after its object has been freed.

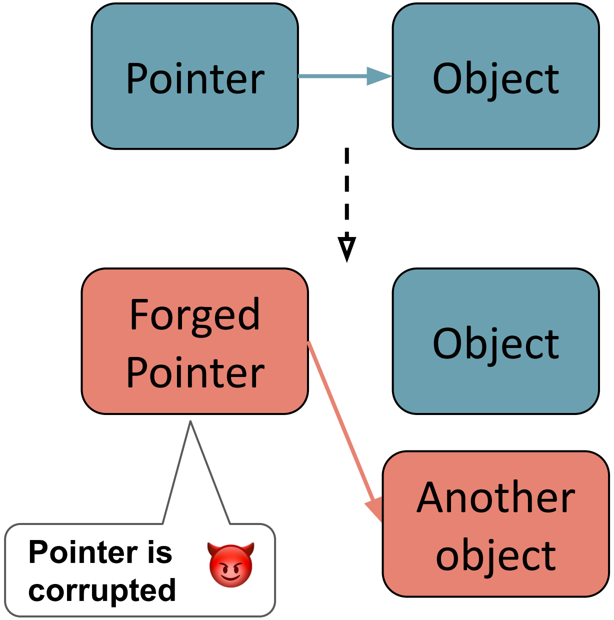


Introducing PACTight

<u>The three properties:</u>

Introducing PACTight

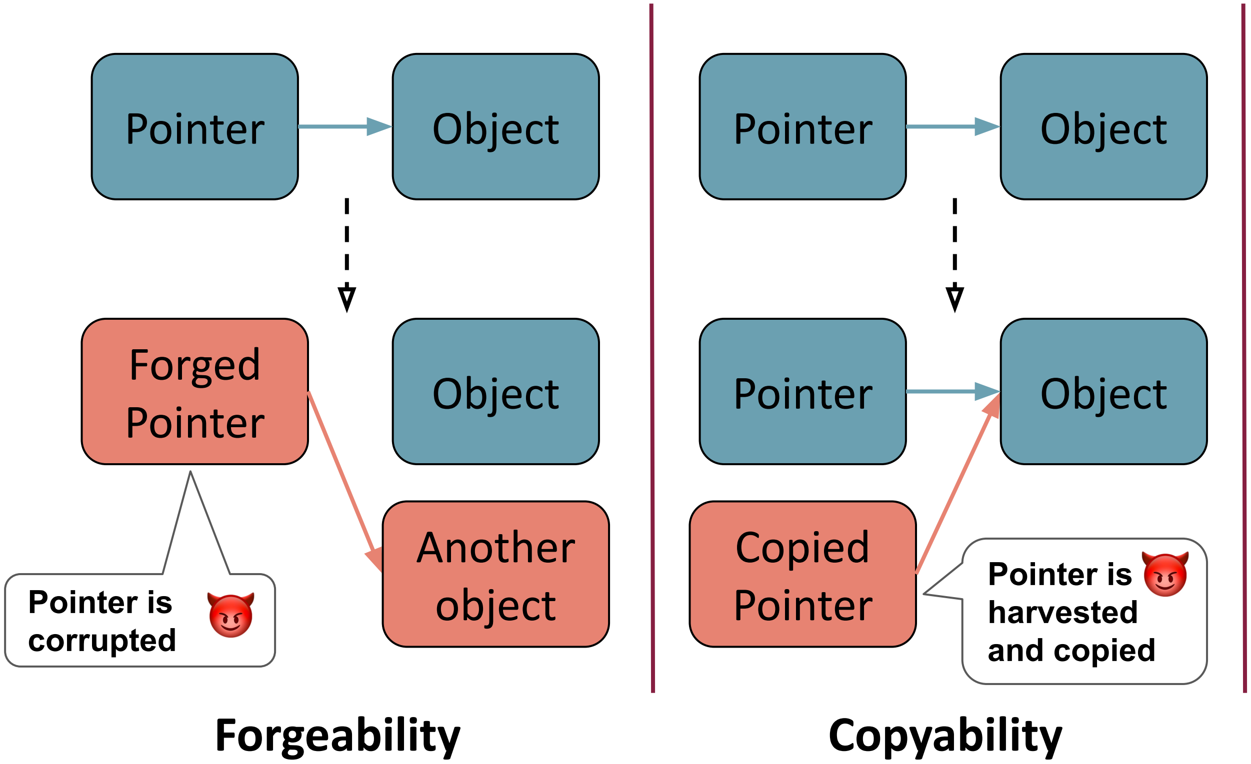
The three properties:



Forgeability

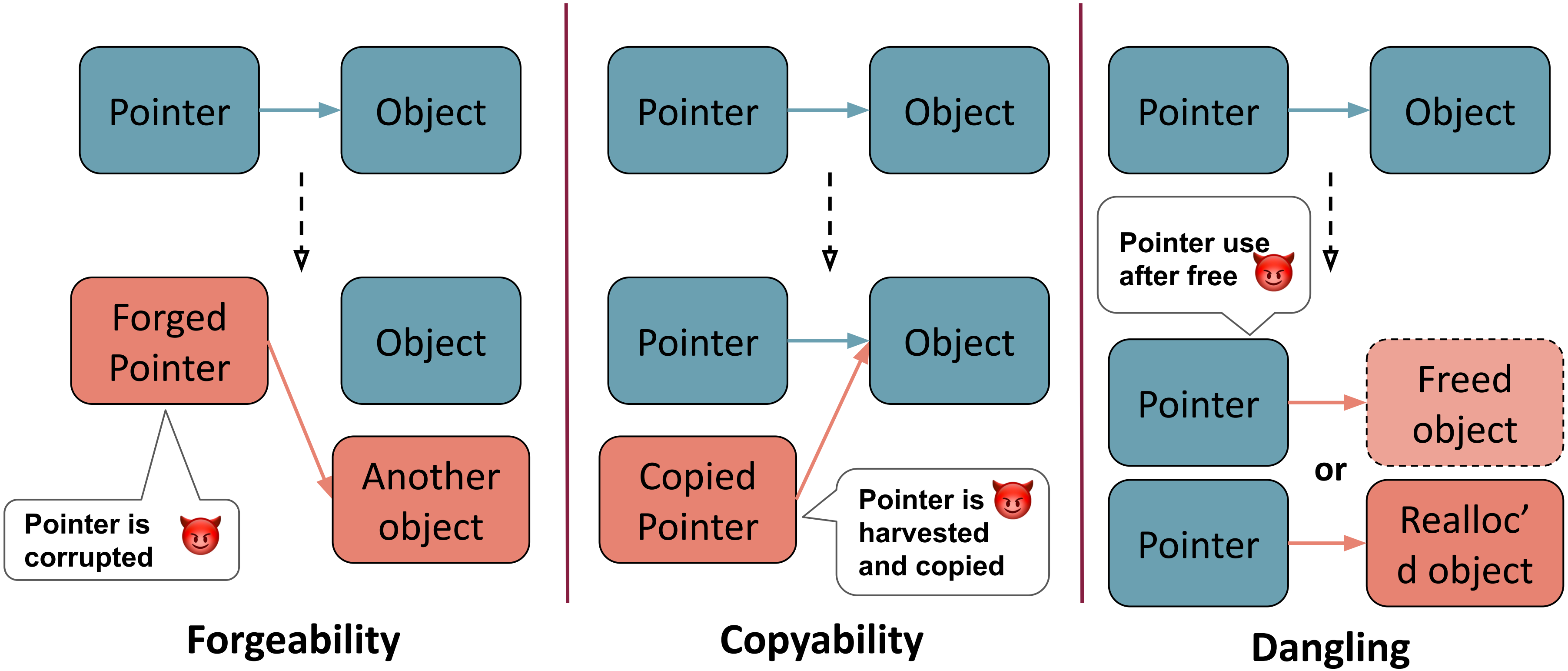
Introducing PACTight

The three properties:



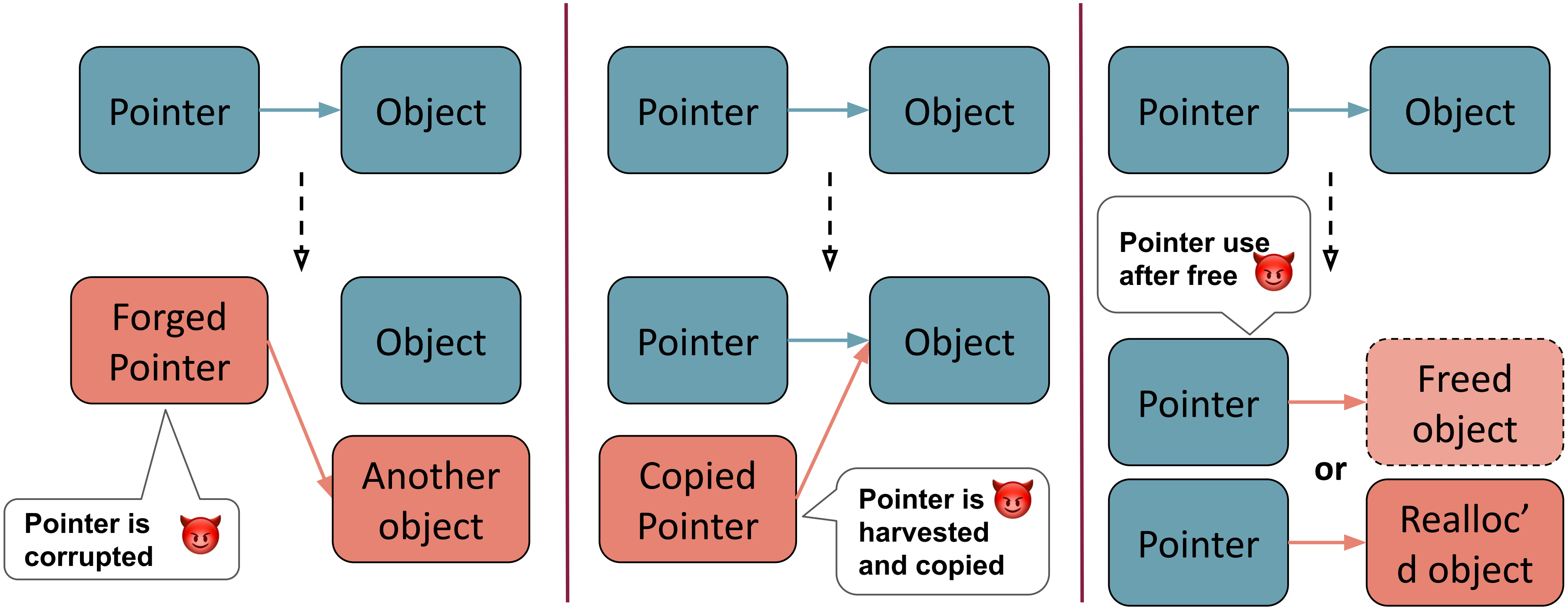
Introducing PACTight

The three properties:



Introducing PACTight

The three properties:



Forgeability -> Generating valid PAC

Copyability -> Reuse valid pointer

Dangling -> Reuse invalid pointer

Introducing PACTight: Goal

- The **importance** of these properties stems from the fact that to hijack control-flow, **at least one** of these properties must be violated.

- PACTight tightly seals pointers and guarantees that a sealed pointer **cannot** be **forged**, **copied**, and is **not dangling**.

- PACTight overcomes the **limitations** of previous approaches:
  - The **non-copyability** property prevents any PAC reuse.
  - PACTight protects all globals, stack variables and heap variables.
  - PACTight assumes a **strong threat model** that has both arbitrary read and write capabilities.

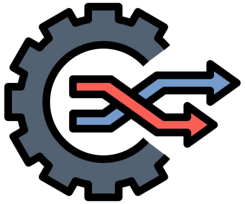
Outline

- Introduction
- Background and related work
- Introducing PACTight
- **PACTight design**
- PACTight defense mechanisms
- Evaluation
- Conclusion

PACTight Design: Enforcing the properties

- In order to enforce the three properties, PACTight relies on the **<u>PAC modifier</u>**.

- Any **<u>changes</u>** in either the **<u>modifier</u>** or the **<u>address</u>** result in a **<u>different PAC</u>**, detecting the violation.

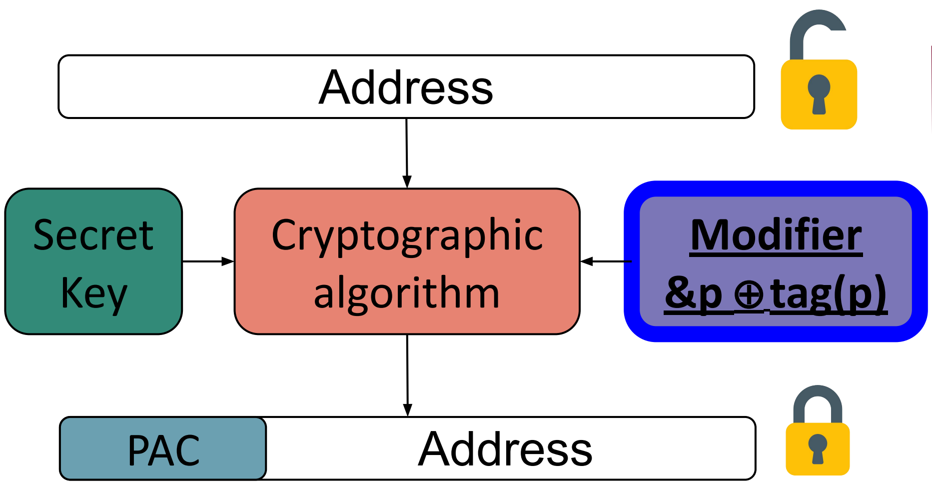


PACTight Design: Enforcing the properties

We propose to blend the **address of a pointer (&p)** and a **random tag associated with a memory object (tag(p))** to efficiently enforce the PACTight pointer integrity property

PACTight Design: Enforcing the properties

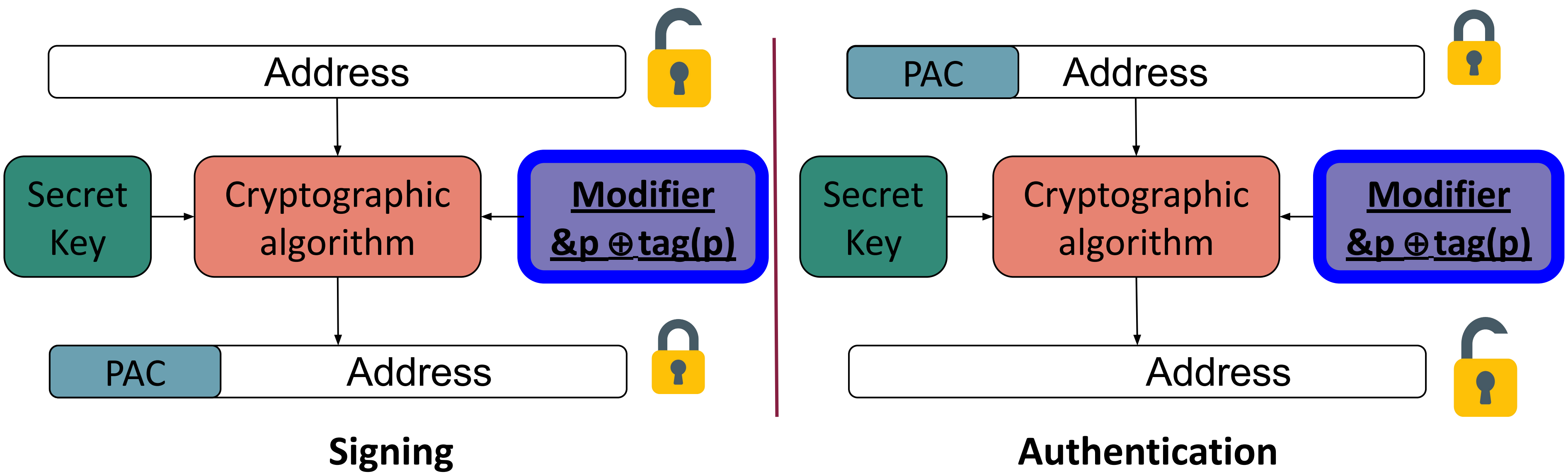
We propose to blend the **address of a pointer (&p)** and a **random tag associated with a memory object (tag(p))** to efficiently enforce the PACTight pointer integrity property



Signing

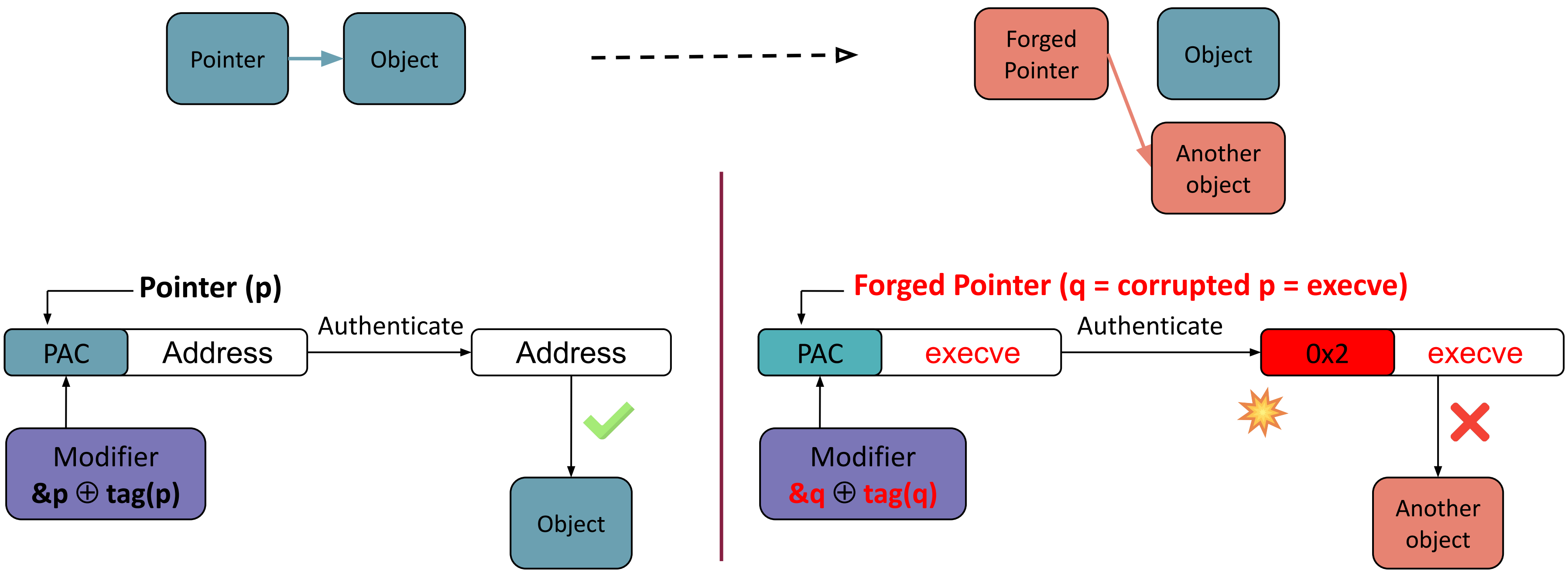
PACTight Design: Enforcing the properties

We propose to blend the **address of a pointer (&p)** and a **random tag associated with a memory object (tag(p))** to efficiently enforce the PACTight pointer integrity property



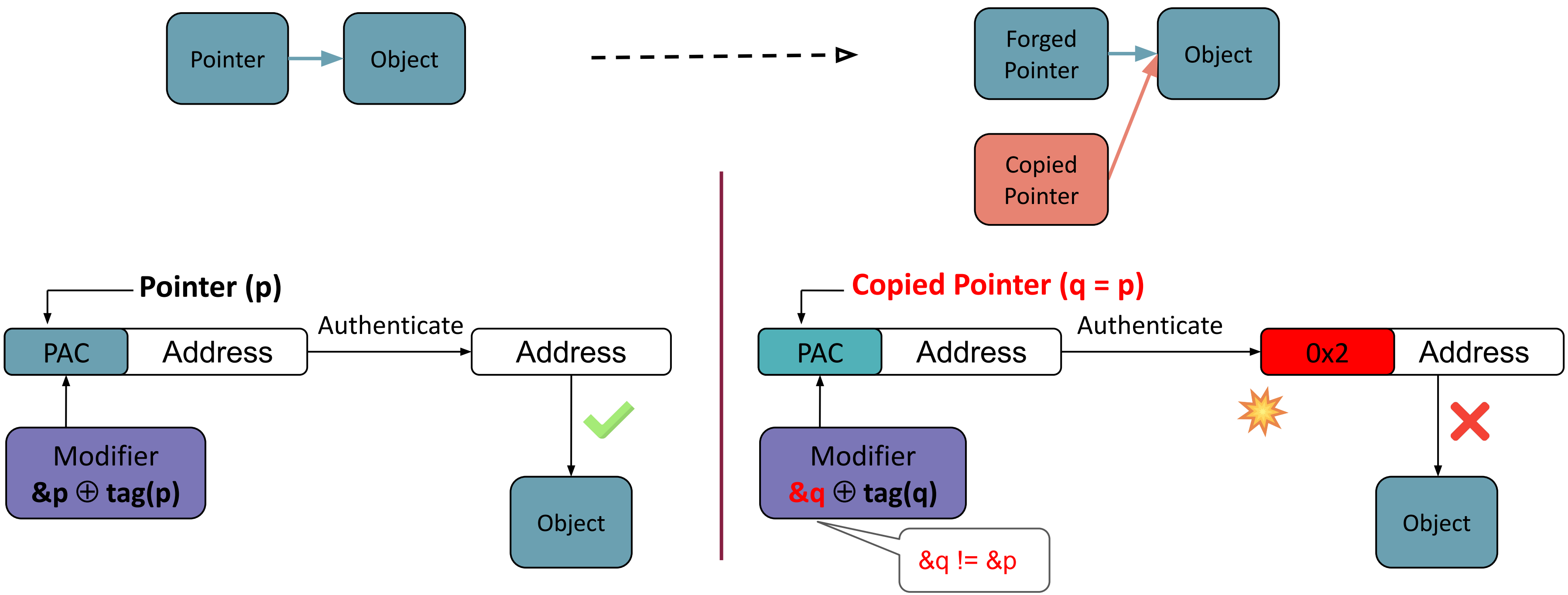
PACTight Design: Enforcing the properties

<u>Unforgeability:</u> The PAC mechanism includes the pointer as one of the inputs to generate the PAC. If the pointer is forged, it will be detected at authentication.



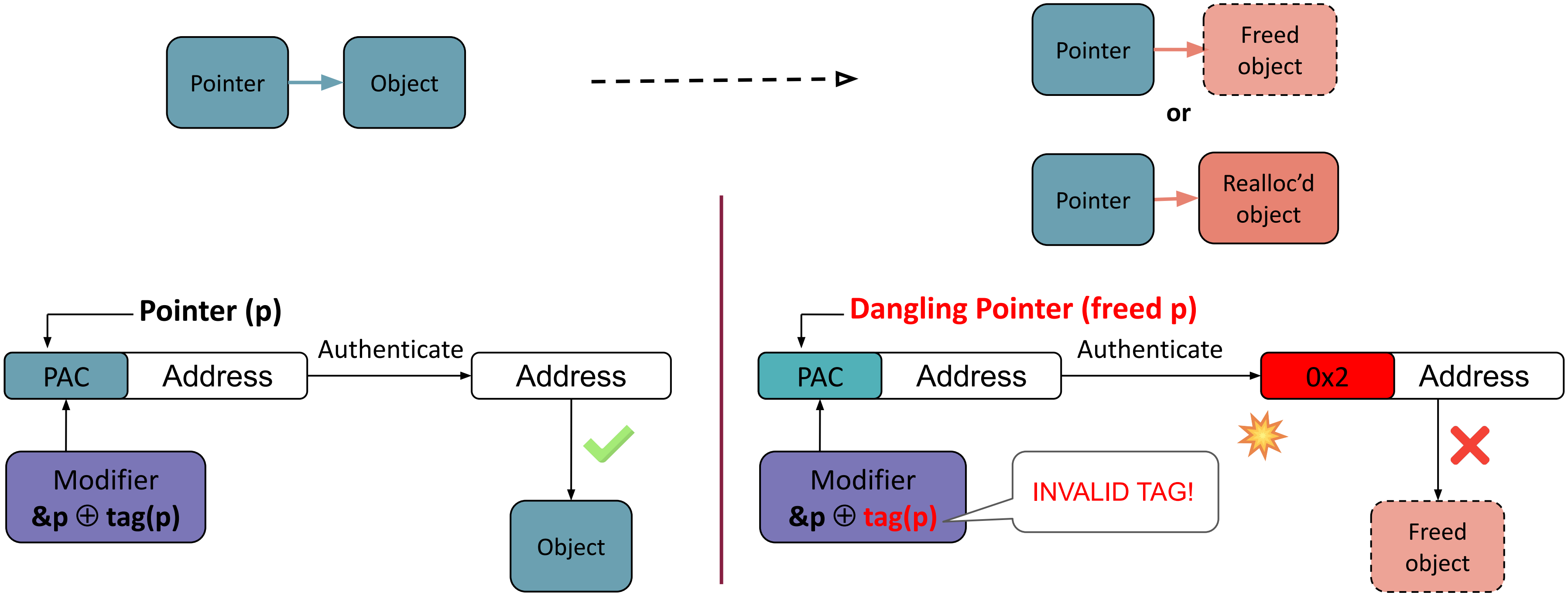
PACTight Design: Enforcing the properties

<u>Non-copyability:</u> PACTight adds the **location of the pointer (&p)** as part of the modifier. Any change in the location by copying the pointer triggers an authentication fault.



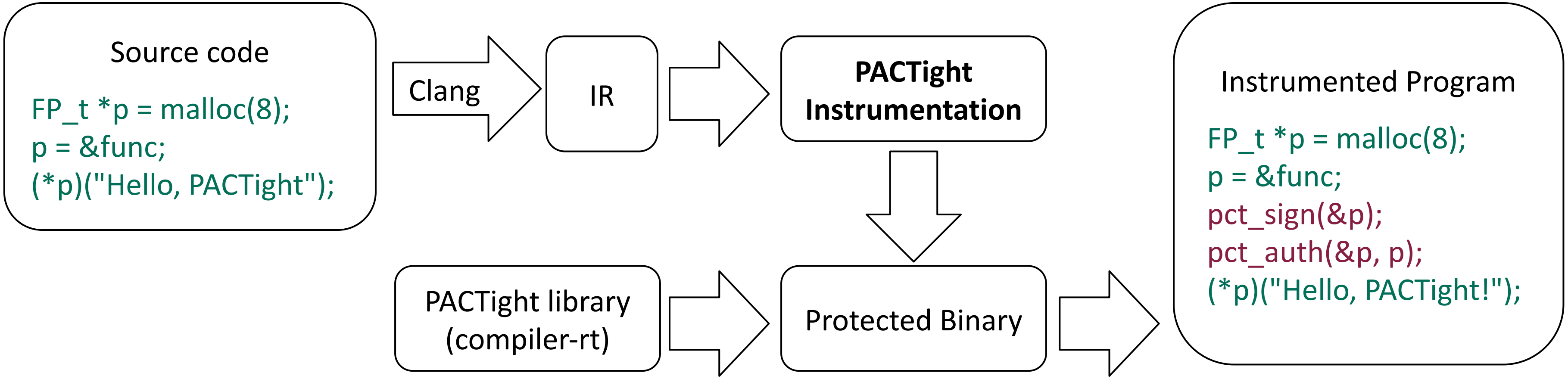
PACTight Design: Enforcing the properties

<u>Non-dangling:</u> PACTight uses a **random tag** to track the lifecycle of a memory object. The lifecycle of a PACTight-sealed pointer is bonded to that of the object.



PACTight structure and overall design

- PACTight instruments programs to guarantee the three properties.
- PACTight automates its instrumentation in four different levels: forward-edge, backward-edge, C++ VTable, and sensitive pointers



Outline

- Introduction
- Background and related work
- Introducing PACTight
- PACTight design
- **PACTight defense mechanisms**
- Evaluation
- Conclusion

PACTight Defense Mechanisms

The PACTight compiler automatically instruments all **globals**, **stack variables** and **heap variables** in a program, inserting the necessary PACTight APIs.

We implement four defense mechanisms:

- Control-Flow Integrity (forward edge protection)
- C++ VTable pointers protection
- Code Pointer Integrity (all sensitive pointer protection) [Kuznetsov et. al, OSDI 2014]
- Return address protection (backward edge protection)

PACTight Defense Mechanisms

The PACTight compiler automatically instruments all **globals**, **stack variables** and **heap variables** in a program, inserting the necessary PACTight APIs.

We implement four defense mechanisms:

- Control-Flow Integrity (forward edge protection)
- C++ VTable pointers protection
- **Code Pointer Integrity (all sensitive pointer protection)** [Kuznetsov et. al, OSDI 2014]
- Return address protection (backward edge protection)

PACTight Defense Mechanisms: PACTight-CPI

- PACTight-CPI guarantees the PACTight pointer integrity properties for **all sensitive pointers**.
- Sensitive pointers are **all code pointers** and **all data pointers that point to code pointers recursively**.
- It authenticates the PAC on a sensitive pointer at **legitimate sensitive sites**. At all other sites, the pointer is **sealed** so it cannot be abused.
- PACTight-CPI identifies all sensitive pointers using LLVM type information. It **recursively** looks through all elements inside a composite type.

Outline

- Introduction
- Background and related work
- Introducing PACTight
- PACTight design
- PACTight defense mechanisms
- **Evaluation**
- Conclusion

Evaluation Questions

- How effectively can PACTight prevent not only synthetic attacks but also real-world attacks by enforcing PACTight pointer integrity properties?

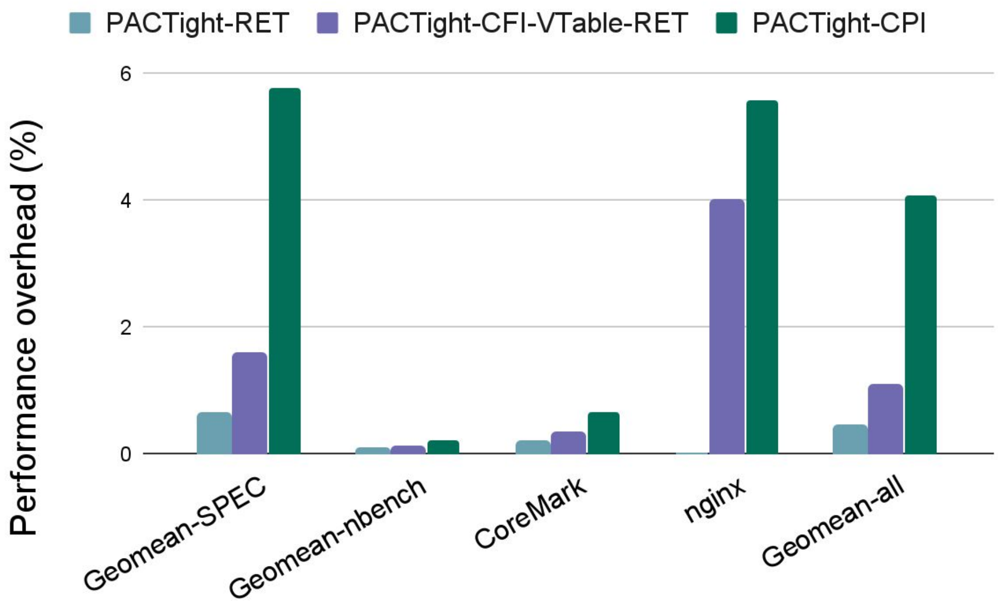
- How much performance and memory overhead does PACTight impose?

Evaluation Questions

- How effectively can PACTight prevent not only synthetic attacks but also real-world attacks by enforcing PACTight pointer integrity properties?

- <u>**How much performance and memory overhead does PACTight impose?**</u>

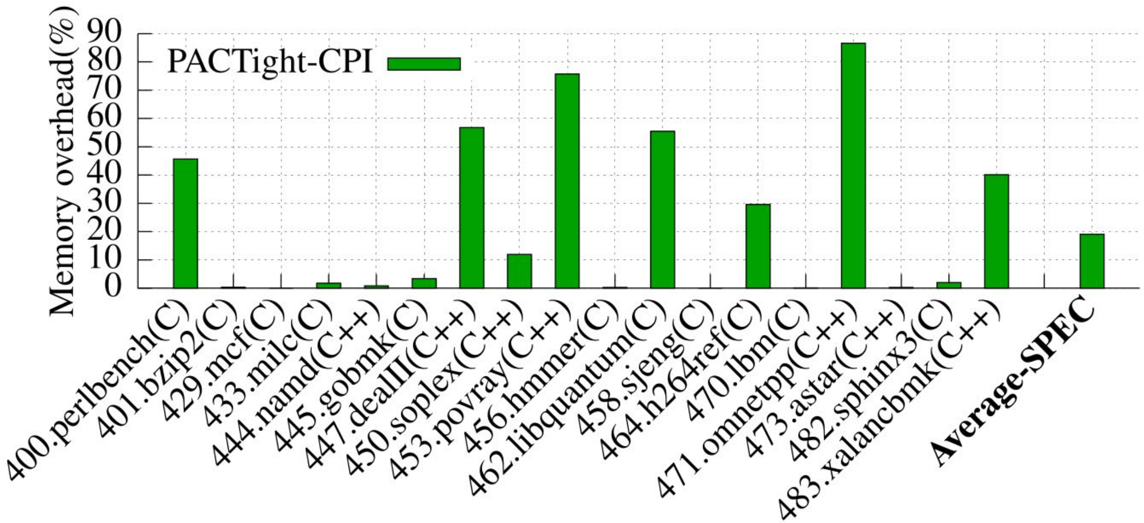
Evaluation: Performance overhead



Geometric mean:

- 0.43% for PACTight-RET
- 1.09% for PACTight-CFI+VTable+RET
- 4.07% for PACTight-CPI

Evaluation: Memory overhead



We ran the SPEC benchmarks with the PACTight-CPI protection:

- 19% memory overhead on average.

Outline

- Introduction
- Background and related work
- Introducing PACTight
- PACTight design
- PACTight defense mechanisms
- Evaluation
- **Conclusion**

Conclusion

- PACTight is an **efficient** and **robust** mechanism utilizing ARM's PA mechanism.

- **Three security properties** that PACTight enforces to ensure pointer integrity.

- We implemented PACTight with four defense mechanisms, protecting **forward-edge**, **backward-edge**, **virtual function pointers**, and **sensitive pointers**.

- PACTight is **secure** against **real** and **synthesized** attacks (more details in the paper) and has **low performance and memory overhead**

Thank you!

Questions?

Mohannad Ismail
imohannad@vt.edu

https://github.com/cosmoss-jigu/pactight

Conclusion

- PACTight is an **efficient** and **robust** mechanism utilizing ARM's PA mechanism.
- **Three security properties** that PACTight enforces to ensure pointer integrity.
- We implemented PACTight with four defense mechanisms, protecting **forward-edge**, **backward-edge**, **virtual function pointers**, and **sensitive pointers**.
- PACTight is **secure** against **real** and **synthesized** attacks (more details in the paper) and has **low performance and memory overhead**

Mohannad Ismail
imohannad@vt.edu

https://github.com/cosmoss-jigu/pactight