Effective and Practical Defenses Against Memory Corruption and Transient Execution Attacks with Hardware Security Features

Mohannad Ismail

September 15, 2022

Committee:
Changwoo Min (Co-chair), Wenjie Xiong (Co-chair)
Danfeng Yao, Michael Hsiao, & Haining Wang

Outline

- Motivation
- Background
- Present Contributions
- Future Work
- Summary

Outline

- **Motivation**
- Background
- Present Contributions
- Future Work
- Summary

Memory safety is a serious problem!

CABLE HAUNT —

Exploit that gives remote access affects ~200 million cable modems

Cable Haunt lets attackers take complete control when targets visit booby-trapped sites.

DAN GOODIN - 1/13/2020, 5:00 PM

Computing Sep 6

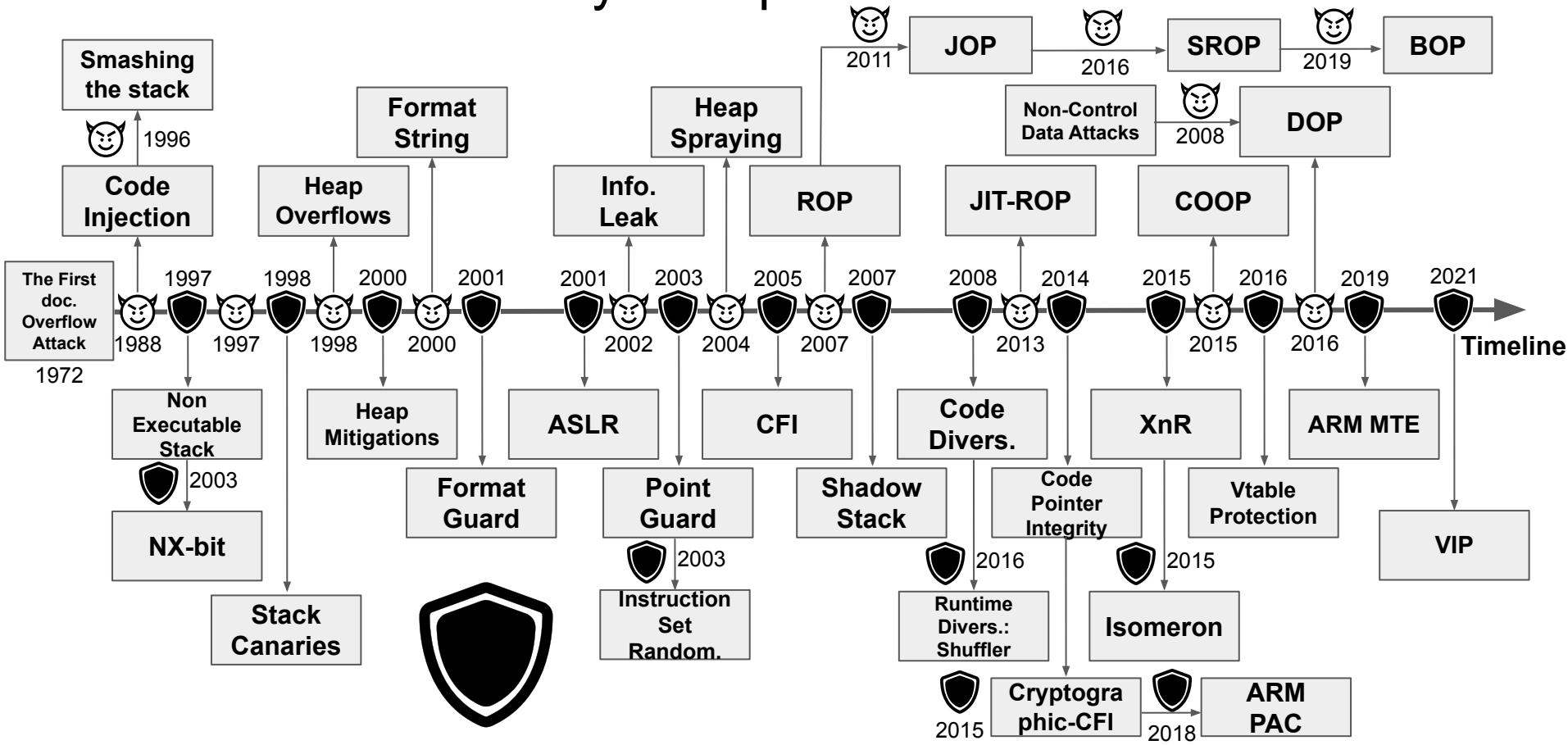
Apple says China's Uighur Muslims were targeted in the recent iPhone hacking campaign

The tech giant gave a rare statement that bristled at Google's analysis of the novel hacking operation.

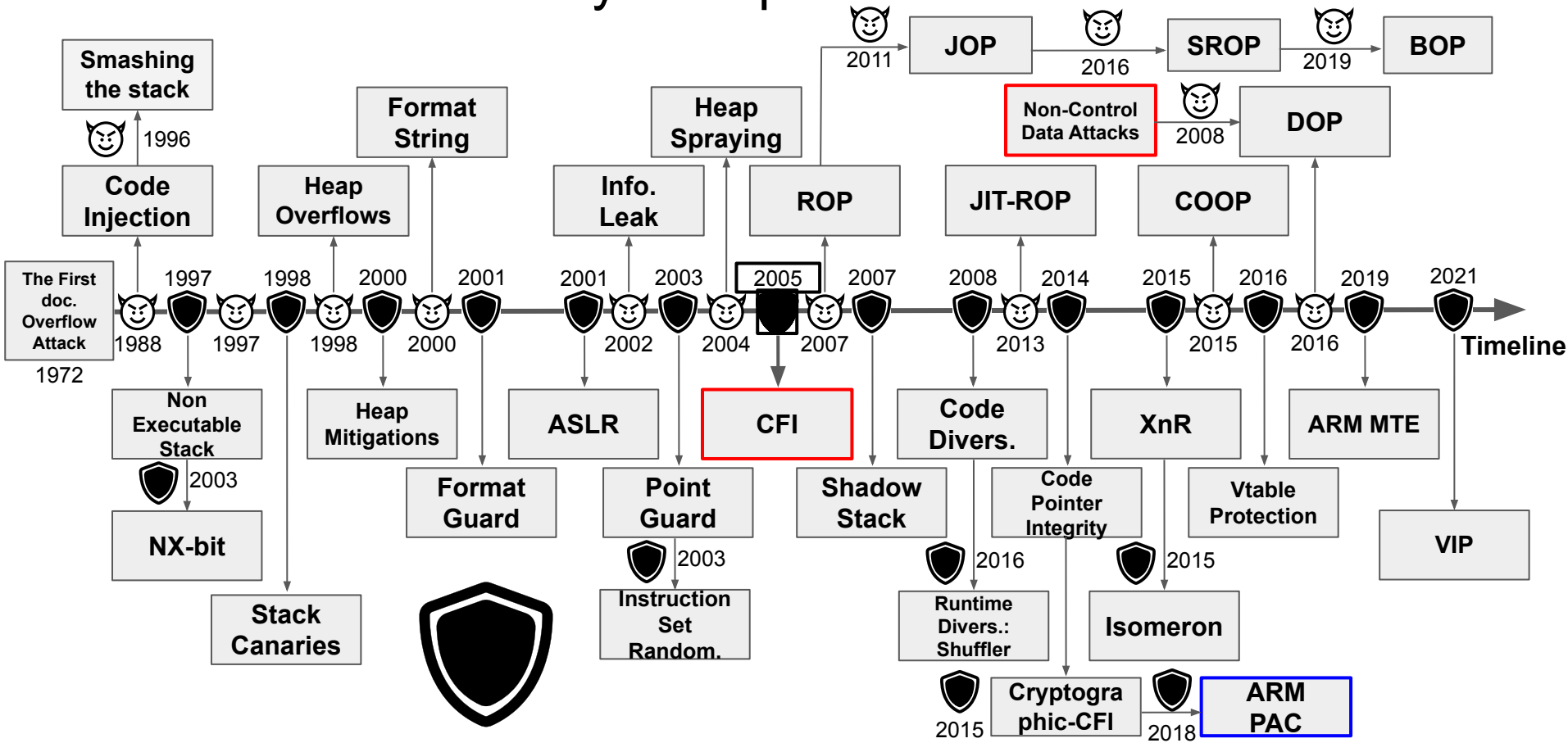
EDITOR'S PICK | 42,742 views | Nov 21, 2018, 07:00am

Exclusive: Saudi Dissidents Hit With Stealth iPhone Spyware Before Khashoggi's Murder

State-of-the-art memory corruption defenses



State-of-the-art memory corruption defenses



Challenges in the current state-of-the-art

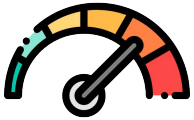
Why are efficient and practical memory safety defense mechanisms hard to achieve?

Low coverage

- Many moving parts make it **difficult** to cover all avenues

Less security guarantees

- Modern attacks require **more code components** to be **guarded**



High Performance overhead

- A defense mechanism needs to be **sufficiently fast** to be used in the real-world

Compatibility

- Defense mechanisms must be **compatible** with current **legacy** code

Thesis Question

Is it possible to design an **effective** and **practical** defense mechanism to defend against **memory corruption attacks** by addressing the challenges above? If so, how?

Thesis Statement

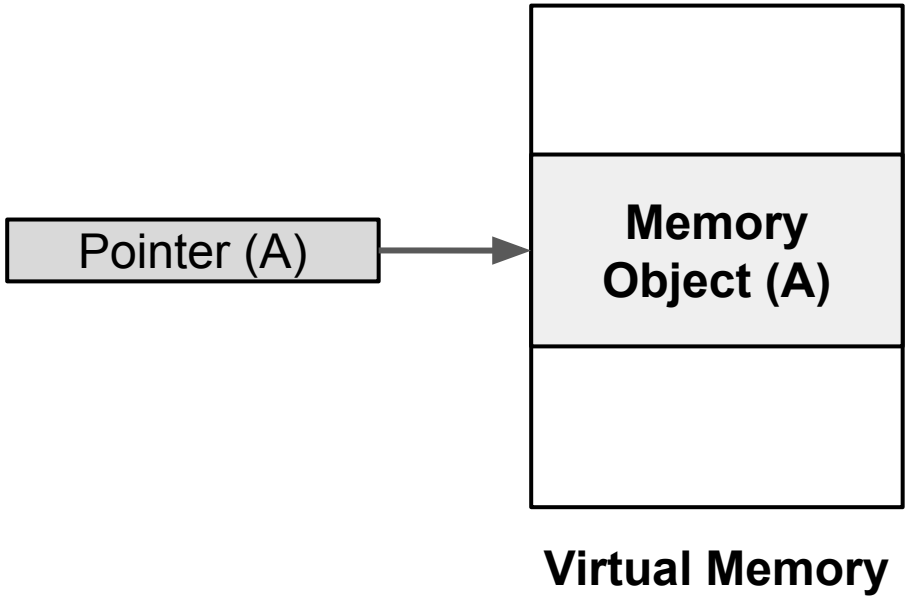
Defense mechanisms against **memory corruption attacks** can be **practical**, **efficient** and have **more coverage** by utilizing hardware security and software codesign.

Outline

- Motivation
- **Background**
  - **Memory safety basics**
  - **Memory safety attacks**
  - **ARM Pointer Authentication**
  - **Past work: PACTight**
- Present Contributions
- Future Work
- Summary

Memory Safety definition

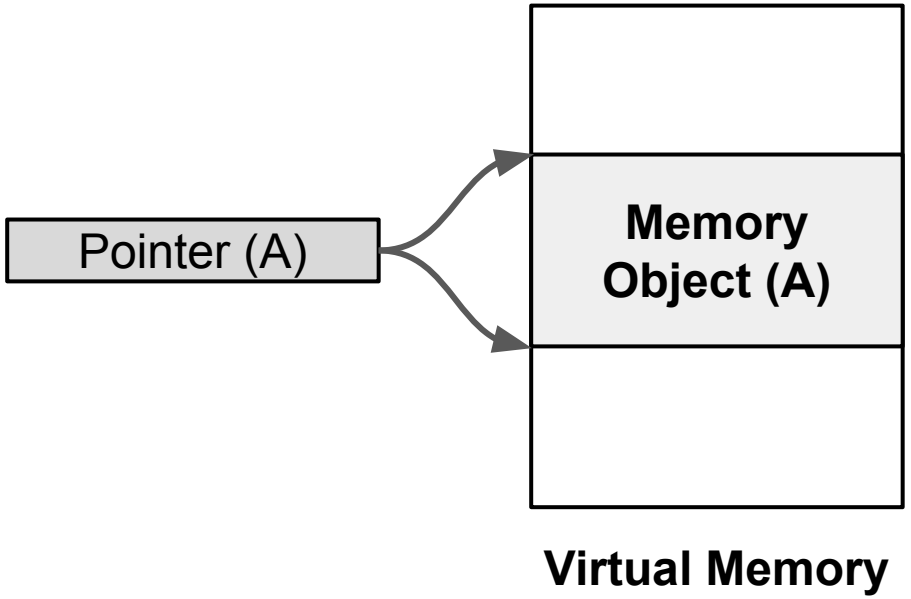
Memory safety means that memory objects are guaranteed to be accessed only:



Memory Safety definition

Memory safety means that memory objects are guaranteed to be accessed only:

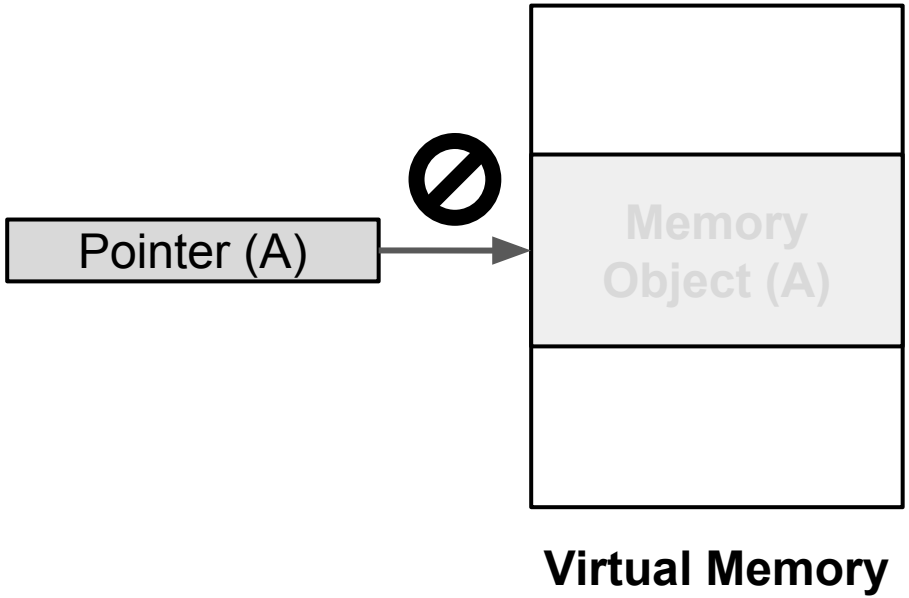
- Between their intended bounds



Memory Safety definition

Memory safety means that memory objects are guaranteed to be accessed only:

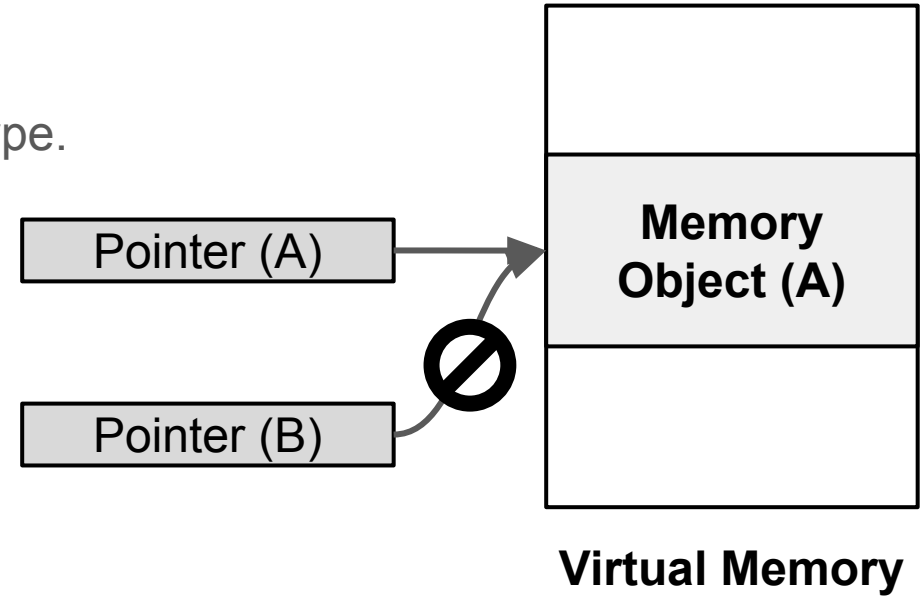
- Between their intended bounds,
- During their lifetime,



Memory Safety definition

Memory safety means that memory objects are guaranteed to be accessed only

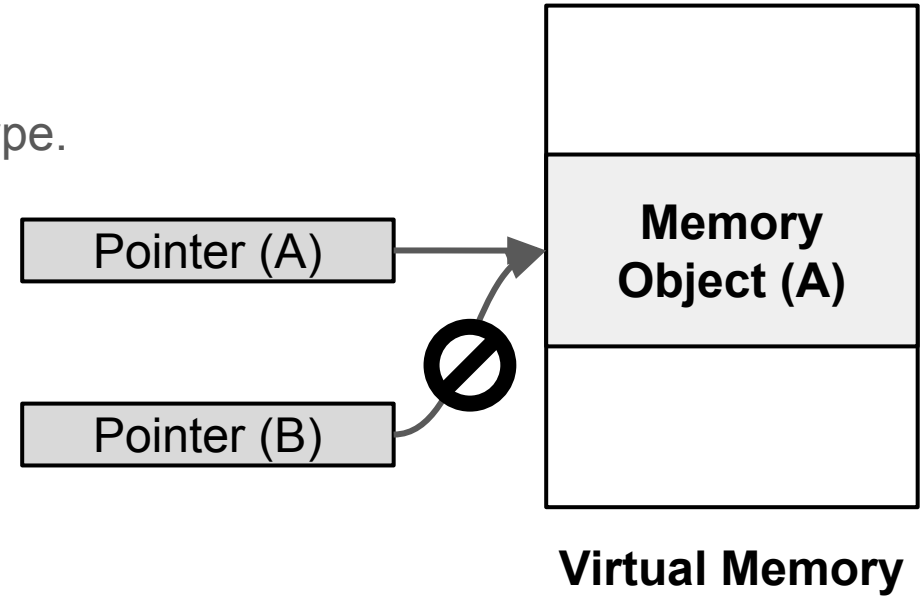
- Between their intended bounds,
- During their lifetime,
- Given their original (or compatible) type.



Memory Safety ~~definition~~ Violations

Memory safety means that memory objects are guaranteed to be accessed only:

- Between their intended bounds,
- During their lifetime,
- Given their original (or compatible) type.



Memory Safety ~~definition~~ Violations

Memory safety means that memory objects are guaranteed to be accessed only:

- ~~Between their intended bounds~~, Buffer overflow
- During their lifetime,
- Given their original (or compatible) type.

Memory Safety ~~definition~~ Violations

Memory safety means that memory objects are guaranteed to be accessed only:

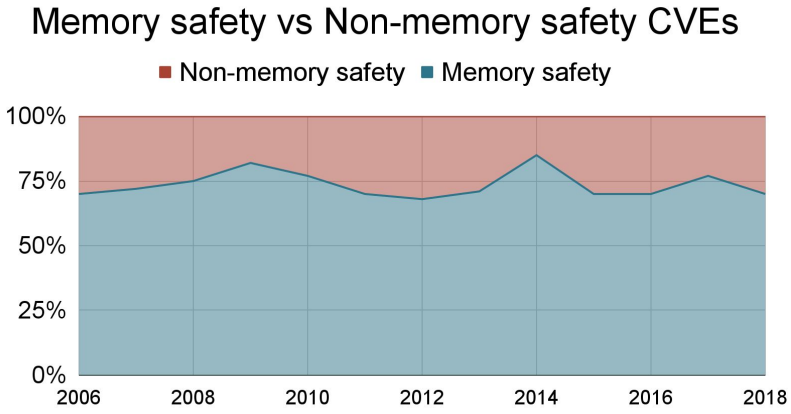
- ~~Between their intended bounds~~, Buffer overflow
- ~~During their lifetime~~, Use-after-free
- Given their original (or compatible) type.

Memory Safety ~~definition~~ Violations

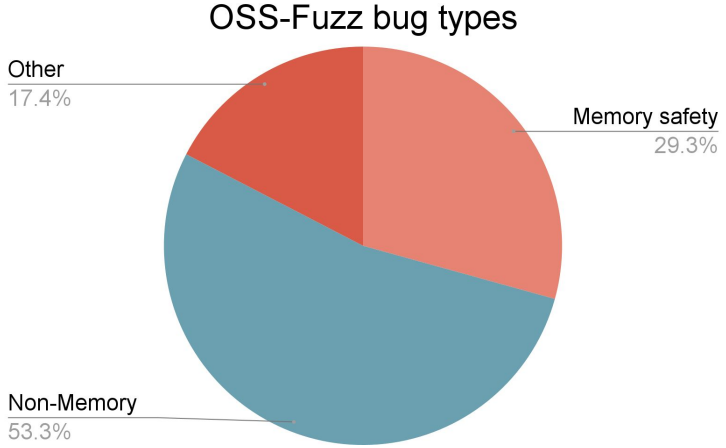
Memory safety means that memory objects are guaranteed to be accessed only:

- ~~Between their intended bounds~~, Buffer overflow
- ~~During their lifetime~~, Use-after-free
- ~~Given their original (or compatible) type~~. Type confusion

Memory safety is a serious problem!



Microsoft Product CVEs



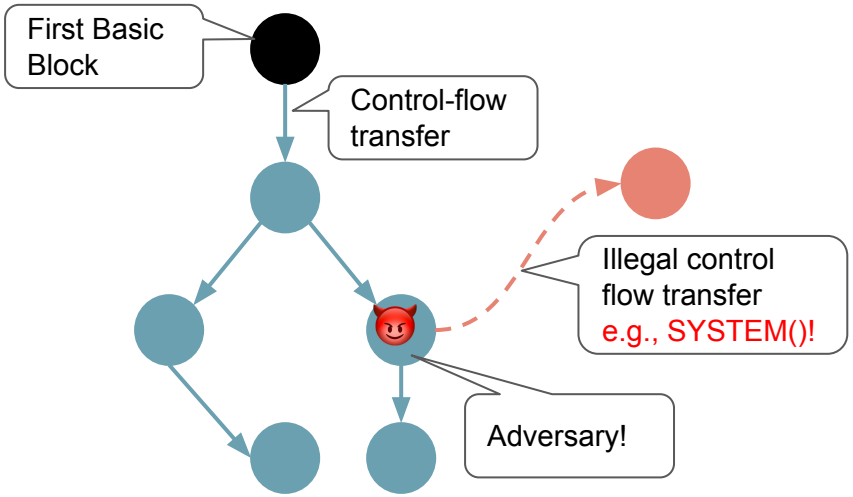
Google OSS (Open Source Software) Fuzz bugs

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

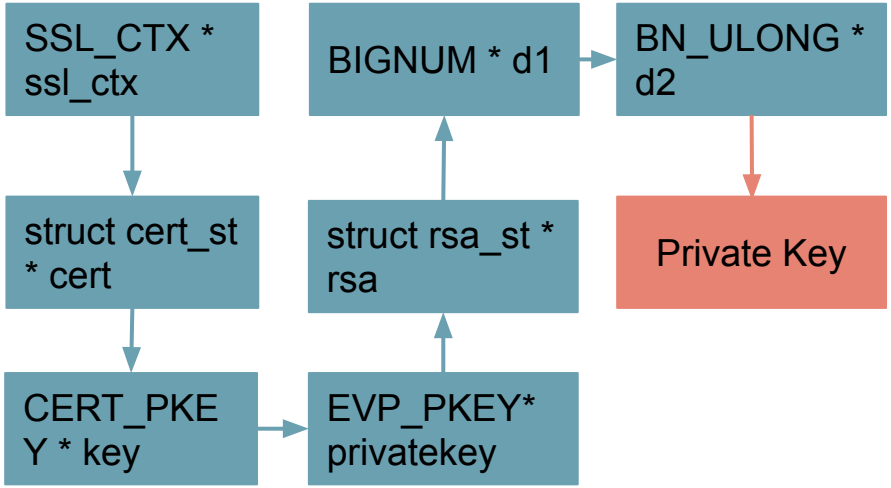
https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

Control-flow hijacking and data-oriented attacks are critical!

Control-flow hijacking and data-oriented attacks are dangerous memory corruption attacks

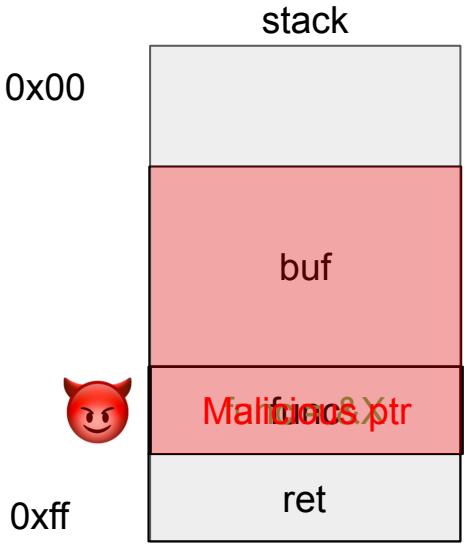


Control-flow hijacking



Data-oriented attack

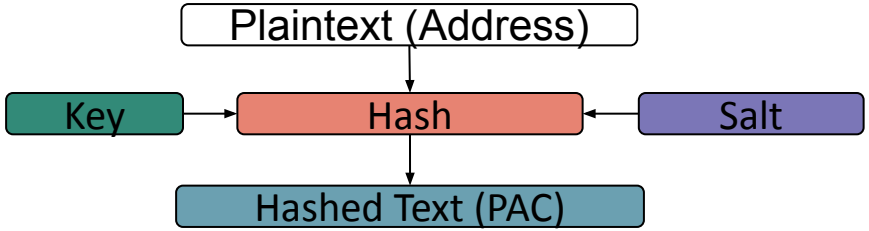
Vulnerable Control Data Example



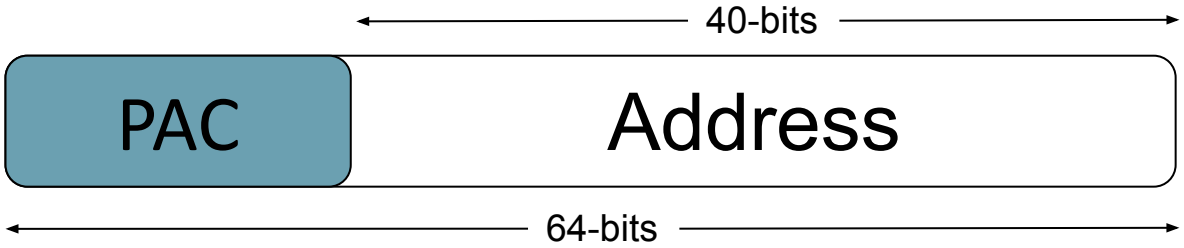

```
1  /** == An example of a code pointer corruption attack ========= */
2  void X(char *); void Y(char *); void Z(char *);
3
4  typedef void (*FP)(char *);
5  static const FP arr[2] = {&X, &Y};
6
7  void handle_req(int uid, char * input) {
8    FP func; // control data to be corrupted!
9    char buf[20]; // buffer that may overflow
10
11   if (uid<0 || uid>1) return; // only allows uid == 0 or 1
12
13   func = arr[uid]; // func pointer assignment, either X or Y.
14
15   strcpy(buf, input); // stack overflow corrupting a code pointer!!!
16
17   (*func)(buf); // func is corrupted!
18
19 }
20 // END
```

ARM Pointer Authentication

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

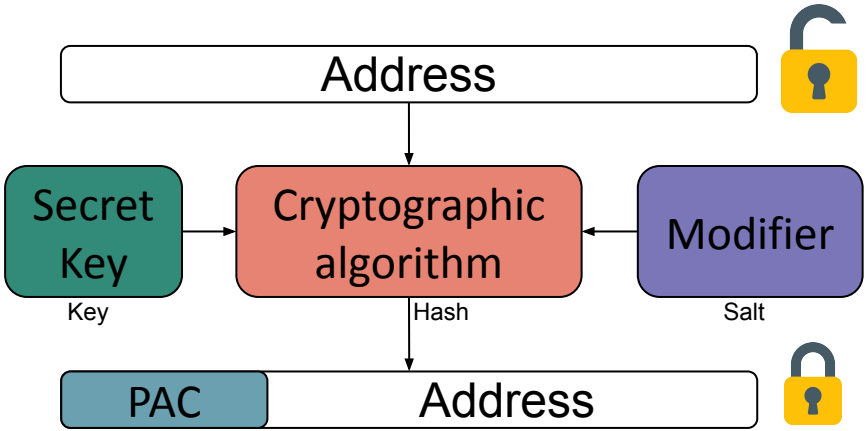


- The PAC is then placed on the unused bits of the 64-bit pointer.

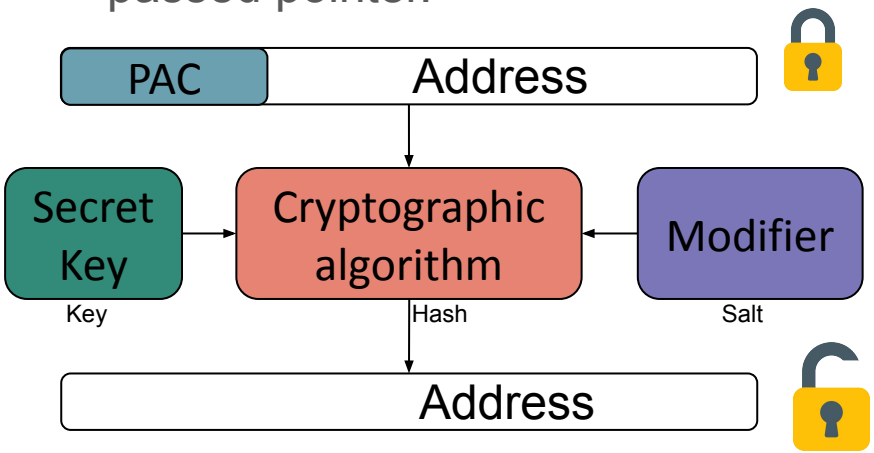


ARM Pointer Authentication

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

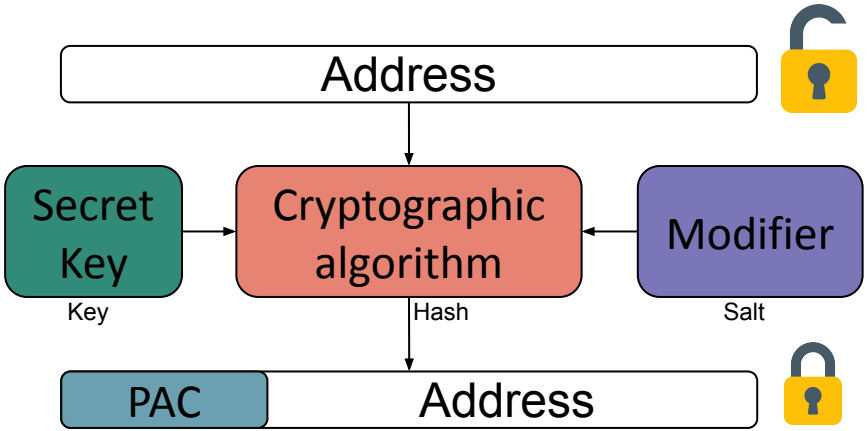


- **PAC authentication:** The algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer.

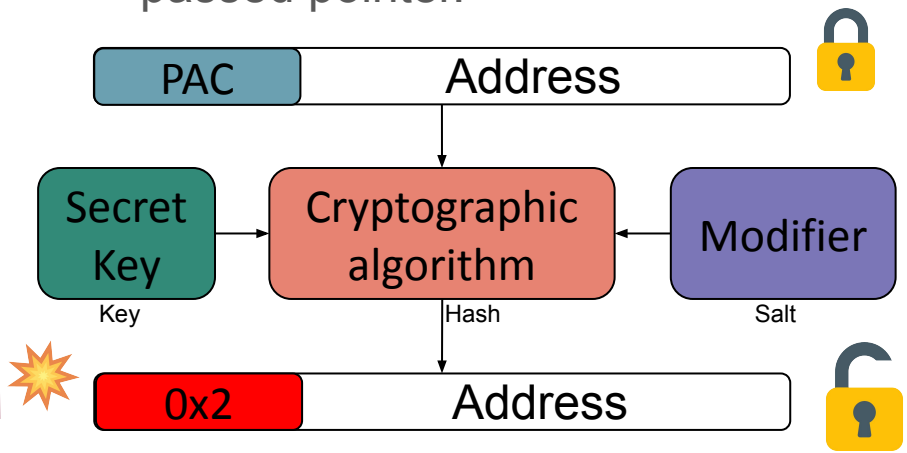


ARM Pointer Authentication

- **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.



Outline

- Motivation
- **Background**
  - Memory safety basics
  - Memory safety attacks
  - ARM Pointer Authentication
  - **Past work: PACTight**
- Present Contributions
- Future Work
- Summary

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]

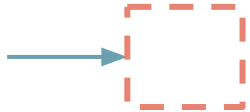
PACTight Overview

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.



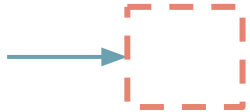
PACTight Overview

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.



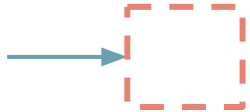
PACTight Overview

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.

- **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.



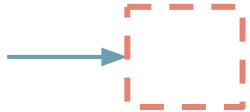
PACTight Overview

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.

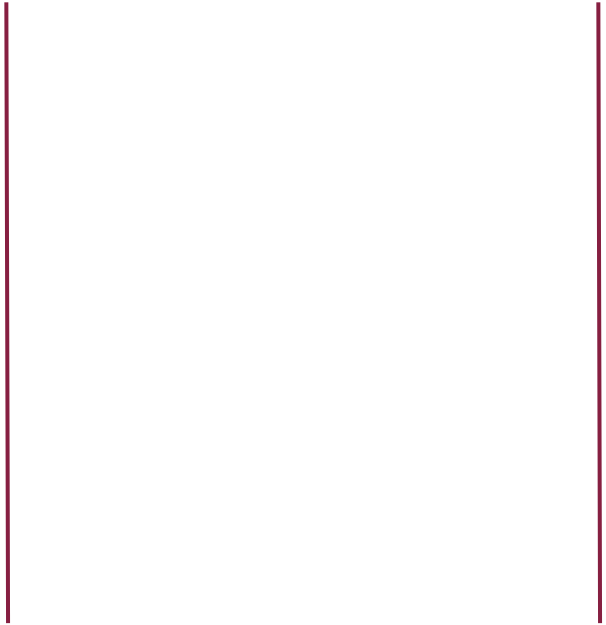
- Non-copyability: A pointer can only be used when it is at its specific legitimate location.

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



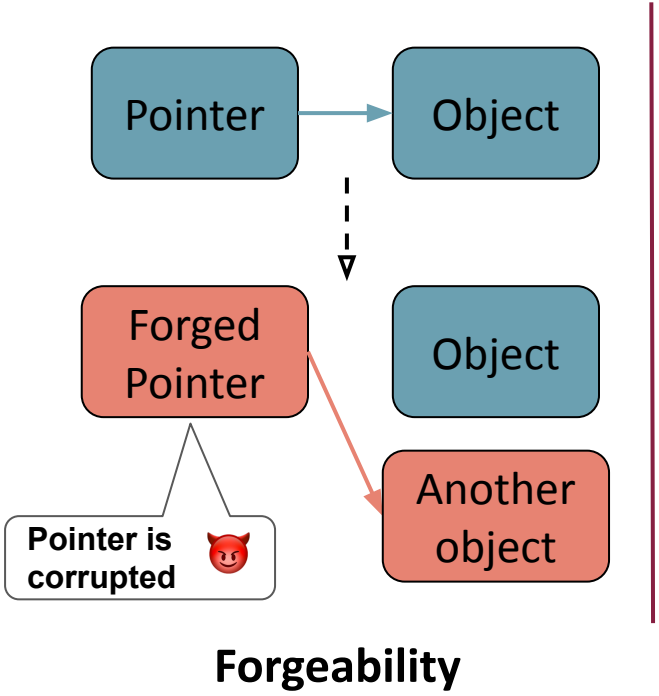
Introducing PACTight

The three properties:



Introducing PACTight

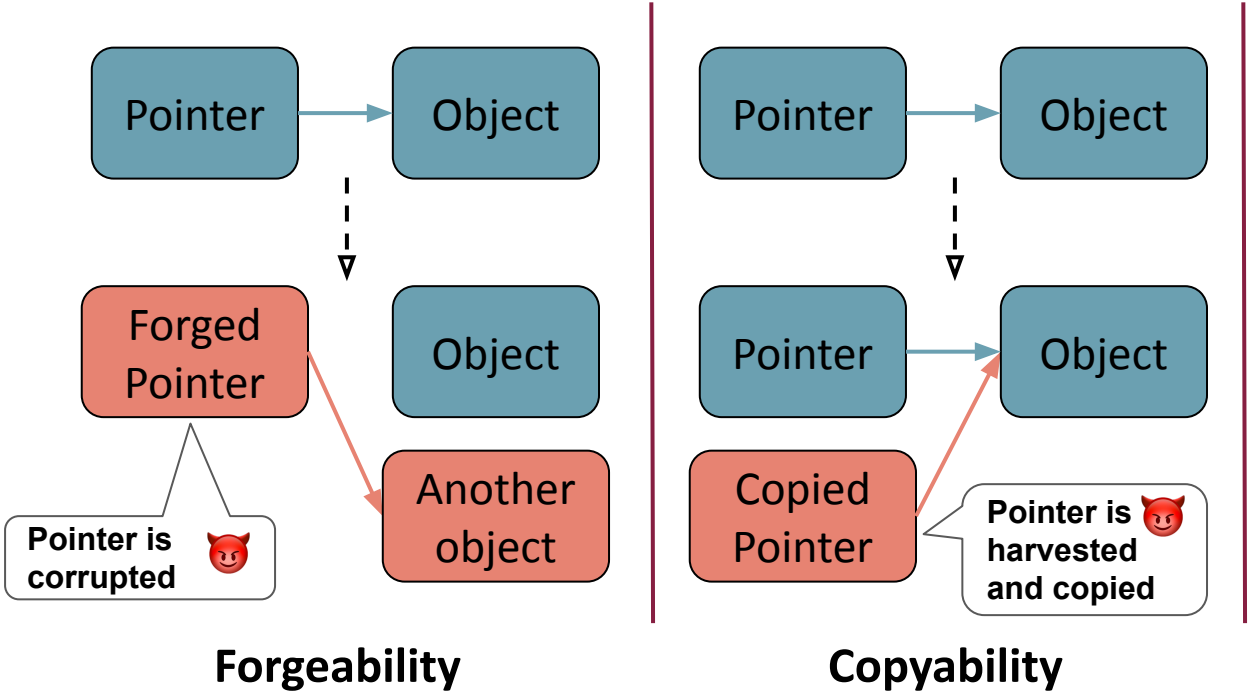
The three properties:



Forgeability

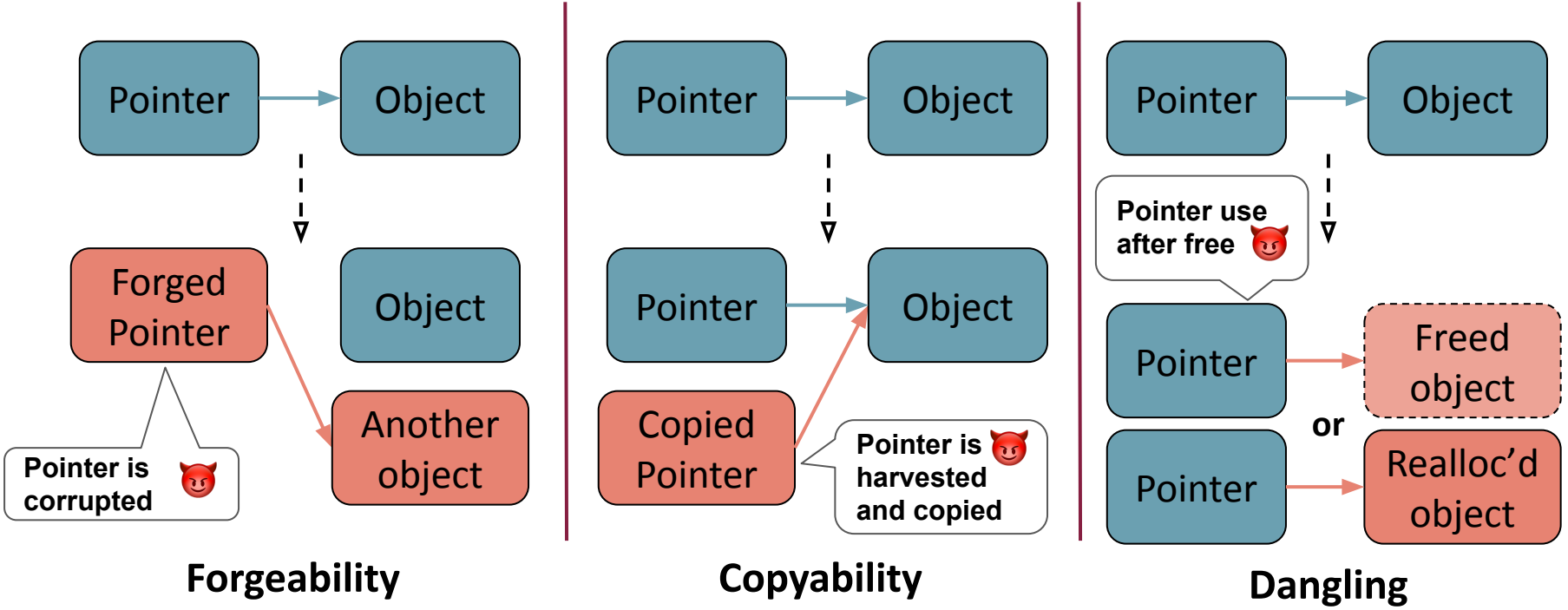
Introducing PACTight

The three properties:



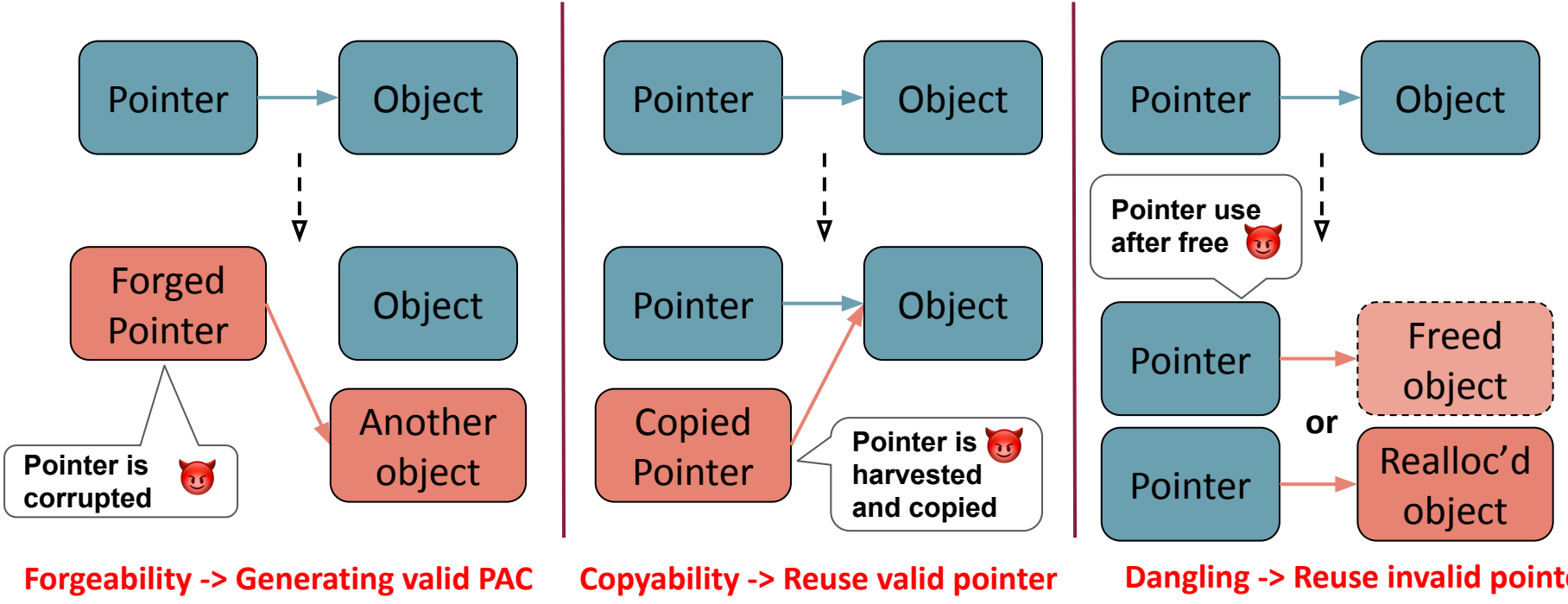
Introducing PACTight

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.

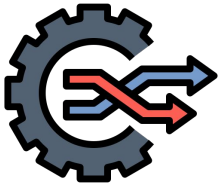
- PACTight tightly seals pointers and guarantees that a sealed pointer **cannot** be **forged**, **copied**, and is **not dangling**.

- 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.

PACTight Design: Enforcing the properties

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

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

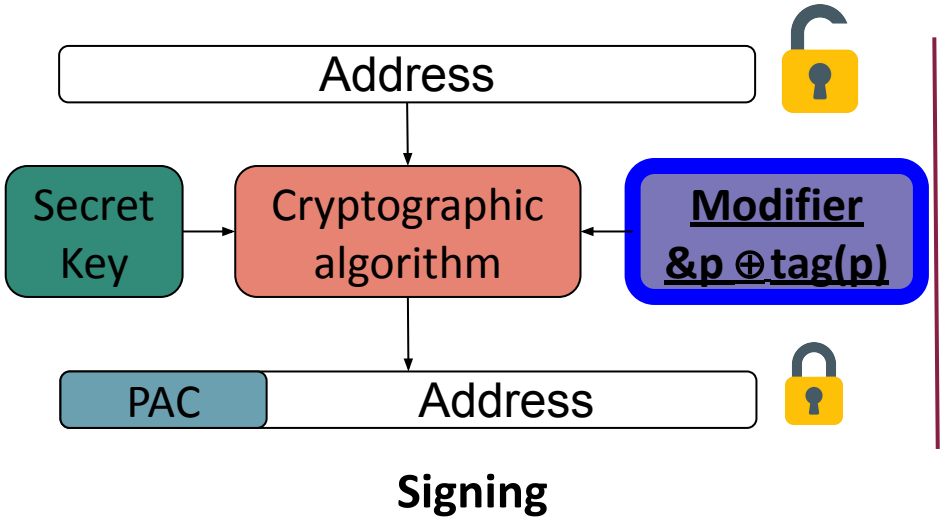


PACTight Design: Enforcing the properties

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

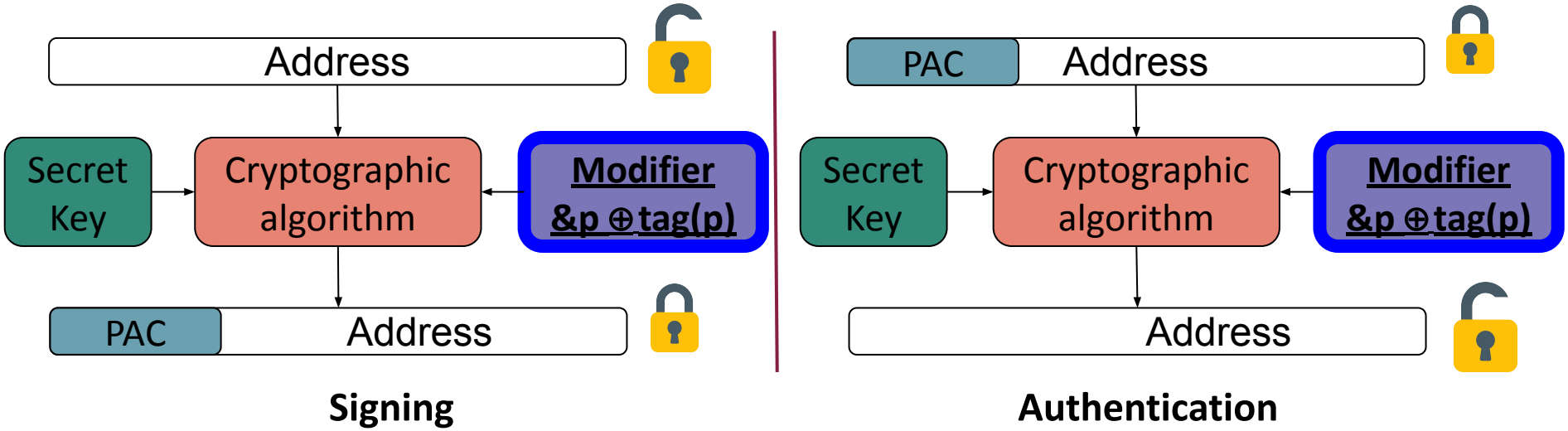
PACTight Design: Enforcing the properties

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



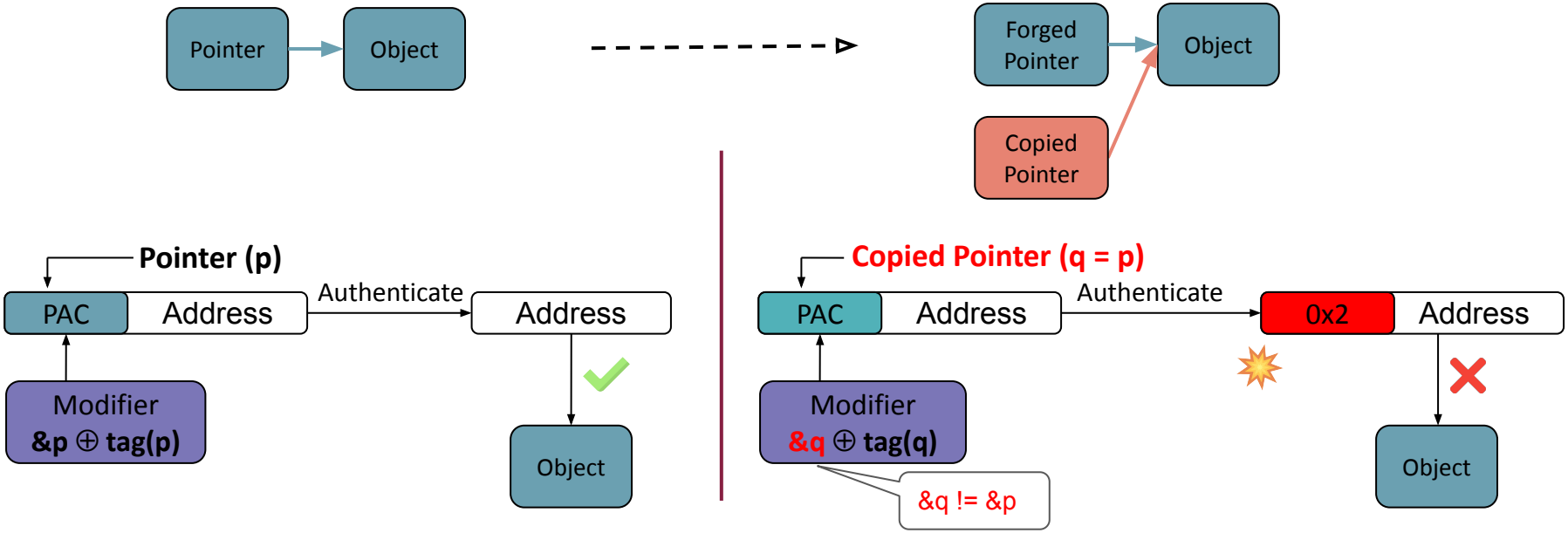
PACTight Design: Enforcing the properties

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

Non-copyability: 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.



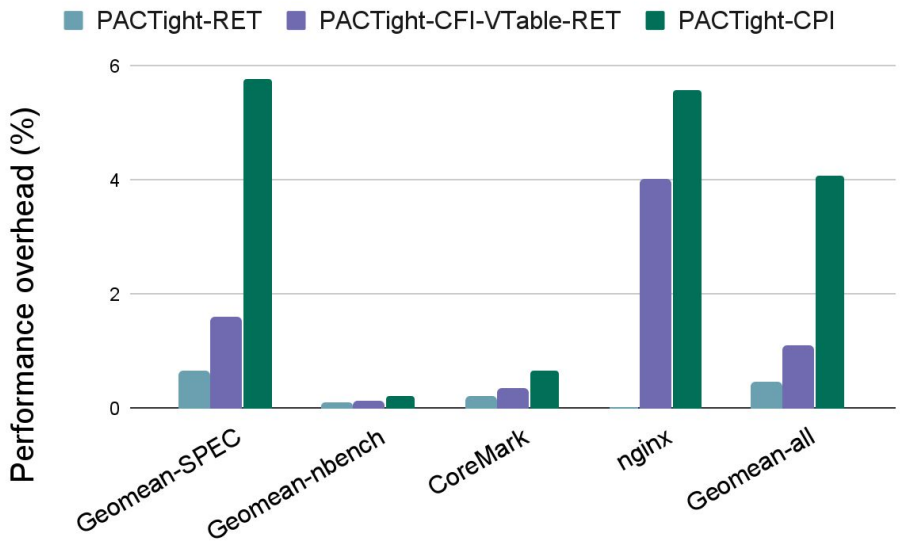
PACTight Defense Mechanisms

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)

Evaluation: Performance overhead



Geometric mean:

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

Summary and limitations

- PACTight is an **efficient** and **robust** mechanism utilizing ARM's PA mechanism.
- **Three security properties** that PACTight enforces to ensure pointer integrity.
- PACTight is **secure** against **real** and **synthesized** attacks (more details in the paper) and has **low performance and memory overhead**
- Published in USENIX Security 2022
- **Limitations:**
  - Does not protect data pointers.
  - The existence of the external metadata adds overhead and possible security risks.

Summary and limitations

- 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**
- Published in USENIX Security 2022
- **Limitations:**
  - Does not protect data pointers.
  - The existence of the external metadata adds overhead and possible security risks.

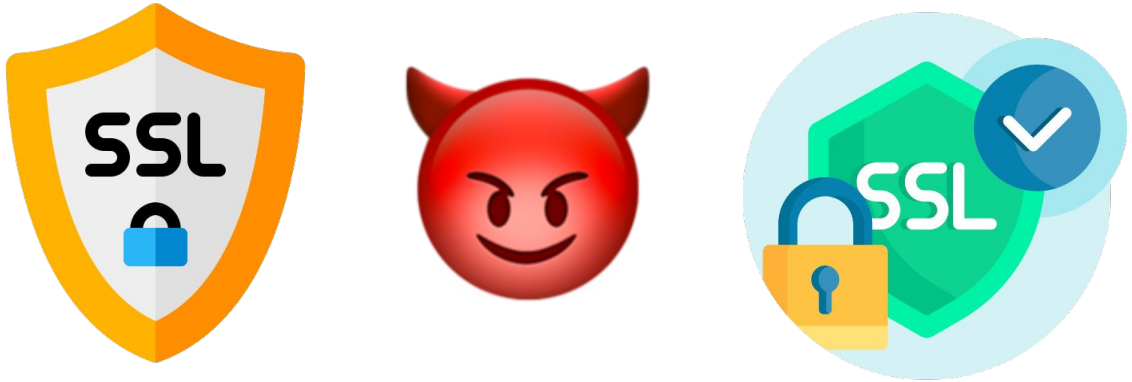
Outline

- Motivation
- Background
- **Present Contributions**
  - **Runtime Scope Type Integrity (RSTI)**
- Future Work
- Summary

Data oriented attacks are becoming more sophisticated!

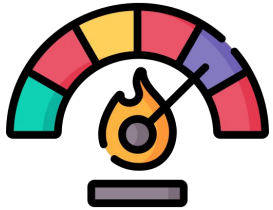
- A new breed of attacks has emerged in recent years that doesn't violate control-flow.

- Attacks such as DOP and NEWTON exploit data pointers to leak sensitive data, such as SSL keys, or achieve arbitrary code execution.



Current data-oriented defense mechanisms are limited!

- Data Flow Integrity (DFI) makes sure that a variable can only be written by its legitimate write instruction.
  - Complete DFI incurs around 103% runtime overhead on average and 50% memory overhead, which is very high.

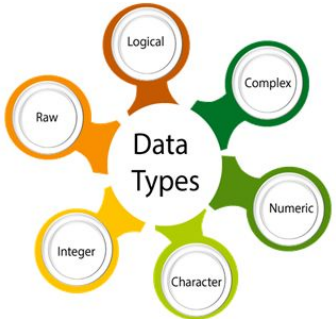


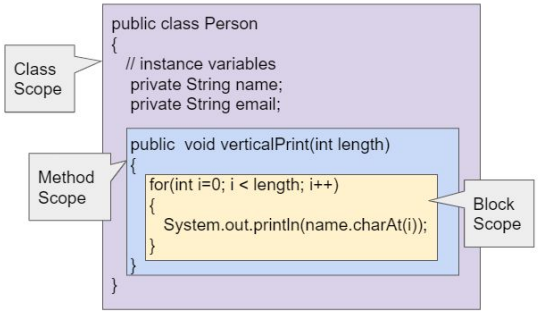
- Hardware assisted Data Flow Isolation (HDFI) provides instruction-level isolation by relying on tags.
  - However, HDFI relies on a 1-bit tag, and thus only two protection domains.

- YARRA is an extension of the C language that relies on programmer annotations to protect critical data types marked by the developer.
  - Performance overhead is variable, but when applied to the whole program, the runtime overhead is in the order of more than 400%. In addition to that, coverage is limited.

The programmer expresses intent when writing a program

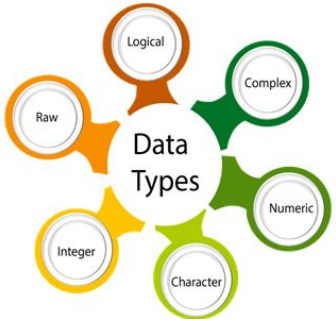
- When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:
  - Basic Type: Each variable must have a specific type, e.g., char, int*.
  - Scope: Scope defines where the variable will be used.
  - Permission: We refer to permissions as whether a variable is defined as read or read/write.

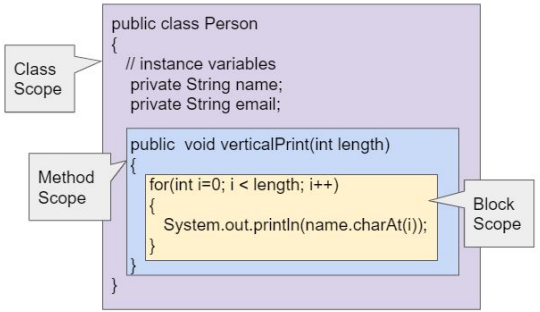




The programmer expresses intent when writing a program

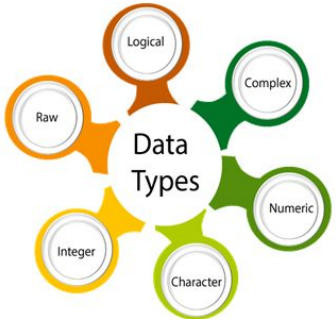
- When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:
  - **Basic Type:** Each variable must have a specific type, e.g., char, int*.
  - Scope: Scope defines where the variable will be used.
  - Permission: We refer to permissions as whether a variable is defined as read or read/write.

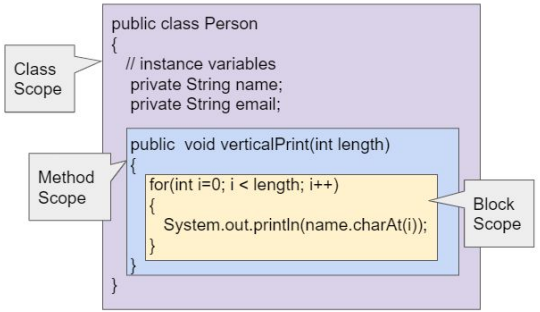




The programmer expresses intent when writing a program

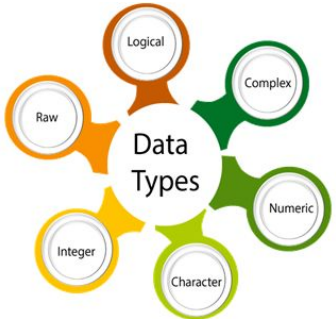
- When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:
  - Basic Type: Each variable must have a specific type, e.g., char, int*.
  - **Scope:** Scope defines where the variable will be used.
  - Permission: We refer to permissions as whether a variable is defined as read or read/write.

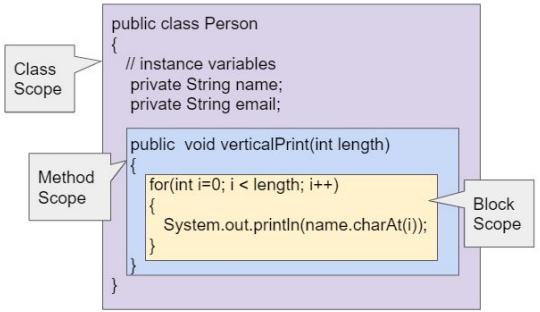




The programmer expresses intent when writing a program

- When a programmer writes a C/C++ program, each defined variable has a few properties. Some of these are:
  - Basic Type: Each variable must have a specific type, e.g., char, int*.
  - Scope: Scope defines where the variable will be used.
  - **Permission:** We refer to permissions as whether a variable is defined as read or read/write.





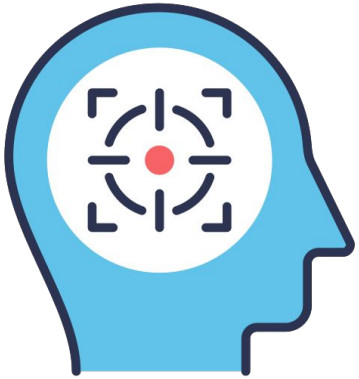
The programmer expresses intent when writing a program

- Programmer **intent** gets lost at runtime!

- What if we **bring back** the programmer's intent to the **runtime**?

Scope-Type Integrity (STI)

- A new defense policy that enforces pointers to conform to the **programmer's intended manner**, by utilizing **scope**, **type**, and **permission** information.



- This allows STI to defeat advanced pointer attacks since these attacks typically violate either the scope, type, or permission.

Runtime Scope-Type Integrity (RSTI)

- RSTI leverages ARM Pointer Authentication (PA) to generate Pointer Authentication Codes (PACs), based on the information from STI, and place these PACs at the top bits of the pointer.

- At runtime, the PACs are then checked to ensure pointer usage complies with STI.

RSTI: Design goals

- The main goal of RSTI is to protect **all pointers** in a program from memory corruption attacks.

- In summary, our main goals are:
  - **Completeness:** Protection of all pointers in a program.
  - **Little reliance on external metadata:** Eliminate metadata lookup overhead and attacks on the metadata.
  - **High performance:** Keep runtime overhead as low as possible.
  - **Compatibility:** Allow protection of legacy (C/C++) programs without needing programmer annotations.

RSTI: Extracting scope, type and permission

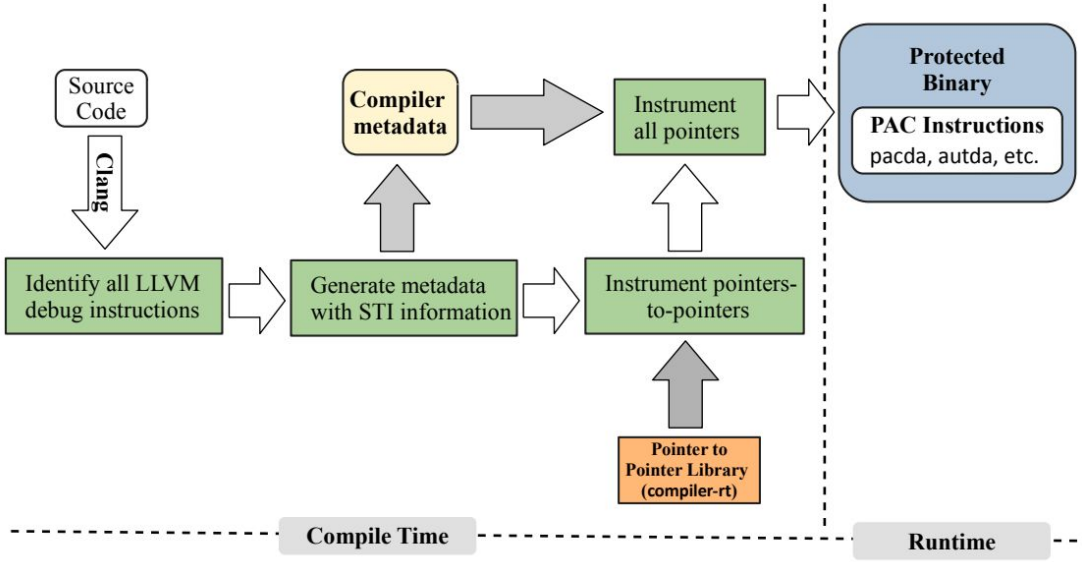
- LLVM Intermediate Representation (IR)
- LLVM debug metadata

```
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

RSTI: Overview




```c
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

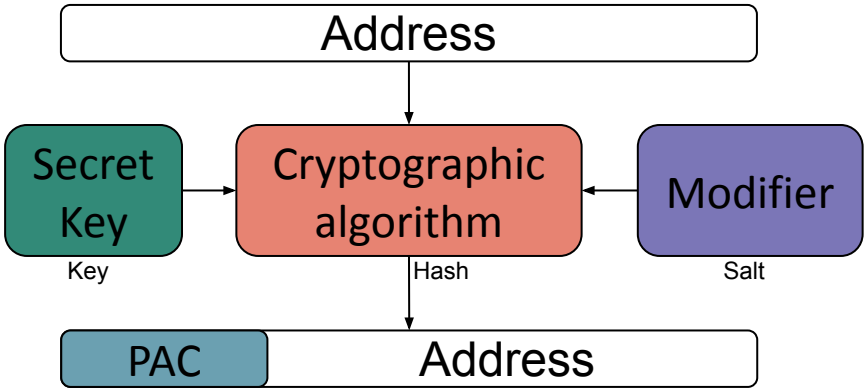
```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

RSTI Type

- A pointer type which restricts a pointer to conform to the scope-type information. If a pointer does not have the intended RSTI-type, RSTI will detect that the pointer has been tampered with.

- RSTI is enforced with ARM's Pointer authentication by adding the RSTI-type to the modifier.

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3



Enforcing RSTI-type

- Three RSTI-types from two basic types.
- Distinguish based on scope, type and permission.
- Referred to as RSTI-STWC (Scope-Type Without Combining).
- Authenticate and resign at casts.

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                              int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                    void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

RSTI Enforcement details

- **LLVM pointer authentication intrinsics**
  - RSTI instruments all pointer loads/stores with PA instructions (pac/aut), leveraging LLVM's pointer authentication intrinsics (llvm.ptrauth)

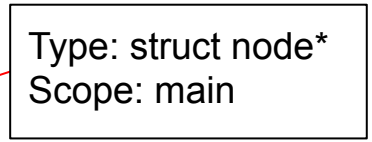
- **On-Store Pointer Signing**
  - RSTI instruments all pointer stores in a program. They are all signed with their respective RSTI-type as the modifier.

- **On-Load Pointer Authentication**
  - RSTI authenticates pointers as they are loaded from memory, using the same RSTI-type that was used to sign them on-store.

RSTI Enforcement details

- **Field sensitive analysis:** RSTI handles members of a composite type and assigns the scope and type appropriately.

```
1  void hello_func(){printf("Hello!"); }
2  struct node {
3     int key;
4     int (*fp)();
5     struct node *next; };
6  int main(void){
7      struct node* ptr = (struct node*)
8                              malloc(sizeof(struct node));
9      ptr->fp = hello_func;
10     ptr->fp(); }
```



RSTI Enforcement details

- **Field sensitive analysis:** RSTI handles members of a composite type and assigns the scope and type appropriately.

```
1  void hello_func(){printf("Hello!"); }
2  struct node {
3     int key;
4     int (*fp)();
5     struct node *next; };
6  int main(void){
7      struct node* ptr = (struct node*)
8                              malloc(sizeof(struct node));
9      ptr->fp = hello_func;
10     ptr->fp(); }
```

Type: int()*
Scope: main, struct node*

RSTI Enforcement details: Pointer-to-pointer handling

- Due to the reliance on the RSTI-type when signing/authenticating pointers, we must make sure that the correct types are being used and propagated. For single pointers, this is not an issue.

- However, for a pointer-to-pointer, this can be a possible issue, as the original type of the pointer can be lost, especially when the double pointer is casted and passed as a function argument.

- Thus, we need to find a way to preserve the original type of the pointer in order to make sure that it does not violate the programmer's intent.

RSTI Enforcement details: Pointer-to-pointer handling

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                          malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```

RSTI Enforcement details: Pointer-to-pointer handling

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                          malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```

RSTI Enforcement details: Pointer-to-pointer handling

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                          malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```

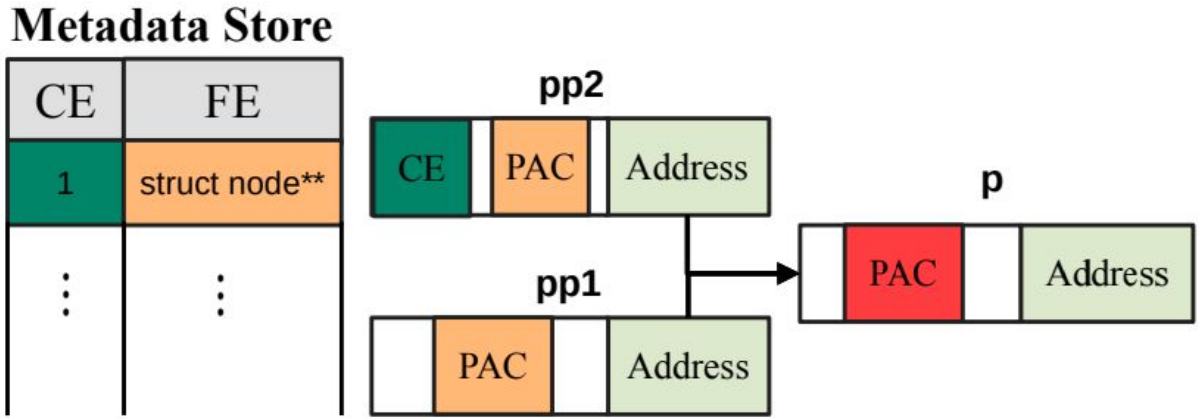
RSTI Enforcement details: Pointer-to-pointer handling

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                          malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```

RSTI Enforcement details: Pointer-to-pointer handling

- Store the original type on the pointer itself.
- RSTI makes use of an ARMv8 hardware feature called Top Byte Ignore (TBI) that ignores the top 8 bits of a virtual address.
- The tag and the full original type are referred to as the Compact Equivalent (CE) and Full Equivalent (FE), respectively.

```
1  void foo1(struct node** pp1){... }
2  void foo2(void** pp2){
3    //pp_auth(pp2, pp2_CE);
4    ...}
5  int main(){
6    struct node* p = (struct node*)
7                        malloc(sizeof(struct node));
8    foo1(&p); //Not instrument with pointer-to-pointer mechanism
9    //pp_add(&p,pp2_CE);
10   //pp_sign(&p, pp2_CE);
11   //pp_add_tbi(&p, pp2_CE);
12   foo2(&p); //Instrument with pointer-to-pointer mechanism
13   ...  }
```



RSTI Enforcement details: Pointer-to-pointer handling

- Not all pointer-to-pointer types are covered. Only ones that are lost when the pointer type goes out of scope.

- The IR provides sufficient information to handle pointer-to-pointer cases that do not fall into this category.

Other RSTI defense mechanisms

- Three RSTI defense mechanisms, each with its own distinct view of the RSTI-type:

1. **RSTI-STWC (Scope-Type Without Combining)** authenticates and re-signs pointers when casts happen.

2. **RSTI-STC (Scope-Type with Combining)** combines compatible types together so that it wouldn't need to re-sign pointers when pointer casts happen.

3. **RSTI-STL (Scope-Type with Location)** combines the location, i.e., address, of the pointer (&p) with the scope-type information.

Other RSTI defense mechanisms

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

(a) RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M2);
19
20
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*, void*	main,foo2	R/W	M1
void*	main	R	M2

(b) RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const = malloc(sizeof(
17                                  void))
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const, 2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   foo2((void*) c);
22   ...}
```

Type	Scope	Permission	RSTI Type
ctx*	main	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3
ctx*	foo	R/W	M4
ctx*	bar	R/W	M5

(c) RSTI-STL

RSTI Enforcement details

Merging of Compatible Types for Casting

- Depending on which mechanism is used and the pointer casts, different RSTI types can be combined or merged together.
- This is showcased below.

```
1 void foo(){
2     void *p1, *p2;
3     int* p3;
4     ...
5     p1 = (void*) p3;
6     ...}
```

	RSTI-STWC	RSTI-STC	RSTI-STL
p1 and p2	Merges RSTI-type of p1 and p2	Merges RSTI-type of p1 and p2	Doesn't merge
p1 and p3	Doesn't merge	Merges RSTI-type of p3 with p1 and p2	Doesn't merge

RSTI Implementation

LLVM Compiler pass

Custom compiler pass for static analysis and instrumentation (3017 LoC)

AArch64 Backend

LLVM pass is executed in the AArch 64 Backend

RSTI P2P library

Library for pointer-to-pointer primitives (487 LoC)

+Additional support

Link Time Optimization (LTO)

RSTI Evaluation

We evaluate RSTI by answering the following questions:

- How effective is RSTI in preventing state-of-the-art attacks as well as real-world attacks?
- How secure is RSTI against abusing pointer-to-pointer metadata and pointers with the same RSTI-type?
- How much performance overhead does RSTI impose in benchmarks and real-world programs?

RSTI Security evaluation

RSTI can defend against both control-flow hijacking and data-oriented attacks, as well as real-world exploits.

Attack Category & Type		Pointers being abused	Original scope-type information	Corrupted scope-type information
Control flow hijacking	NEWTON CsCFI attack [67]	**Corrupted:** `c->send_chain` **Target:** `malloc`	**Type:** `ngx_send_chain_pt` **Scope:** `ngx_http_write_filter`	**Type:** `void* (size_t size)` **Scope:** `libc`
	CVE-2014-8668 - libtiff v4.0.6	**Corrupted:** `tif->tif_encoderow` **Target:** Arbitrary pointer	**Type:** `TIFFCodeMethod` **Scope:** `_TIFFSetDefaultCompression`, `TIFFWriteScanline`, `TIFFOpen`, `main`	Unknown, since this is a CVE and not an attack
Data oriented attack	DOP ProFTPd Attack [36]	**Corrupted:** `&ServerName` **Target:** `resp_buf`, `ssl_ctx`	**Type:** `const char*` **Scope:** `core_display_file`	**Type:** `char*` **Scope:** `pr_response_send_raw`

RSTI Security evaluation

RSTI can defend against both control-flow hijacking and data-oriented attacks, as well as real-world exploits.

Attack Category & Type		Pointers being abused	Original scope-type information	Corrupted scope-type information
Control flow hijacking	NEWTON CsCFI attack [67]	**Corrupted:** `c->send_chain` **Target:** `malloc`	**Type:** `ngx_send_chain_pt` **Scope:** `ngx_http_write_filter`	**Type:** `void* (size_t size)` **Scope:** `libc`
	CVE-2014-8668 - libtiff v4.0.6	**Corrupted:** `tif->tif_encoderow` **Target:** Arbitrary pointer	**Type:** `TIFFCodeMethod` **Scope:** `_TIFFSetDefaultCompression`, `TIFFWriteScanline`, `TIFFOpen`, `main`	Unknown, since this is a CVE and not an attack
Data oriented attack	DOP ProFTPd Attack [36]	**Corrupted:** `&ServerName` **Target:** `resp_buf`, `ssl_ctx`	**Type:** `const char*` **Scope:** `core_display_file`	**Type:** `char*` **Scope:** `pr_response_send_raw`

Analysis on RSTI instrumentation: Equivalence Class

- **Equivalence Class of Type (EC_T)**: We refer to the number of basic types in each RSTI-type as Equivalence Class of Type (EC_T).

- **Equivalence Class of Variable (EC_V)**: We refer to the number of variables in each RSTI-type as Equivalence Class of Variable (EC_V).

Analysis on RSTI instrumentation

Pointer-to-pointer data from SPEC 2006

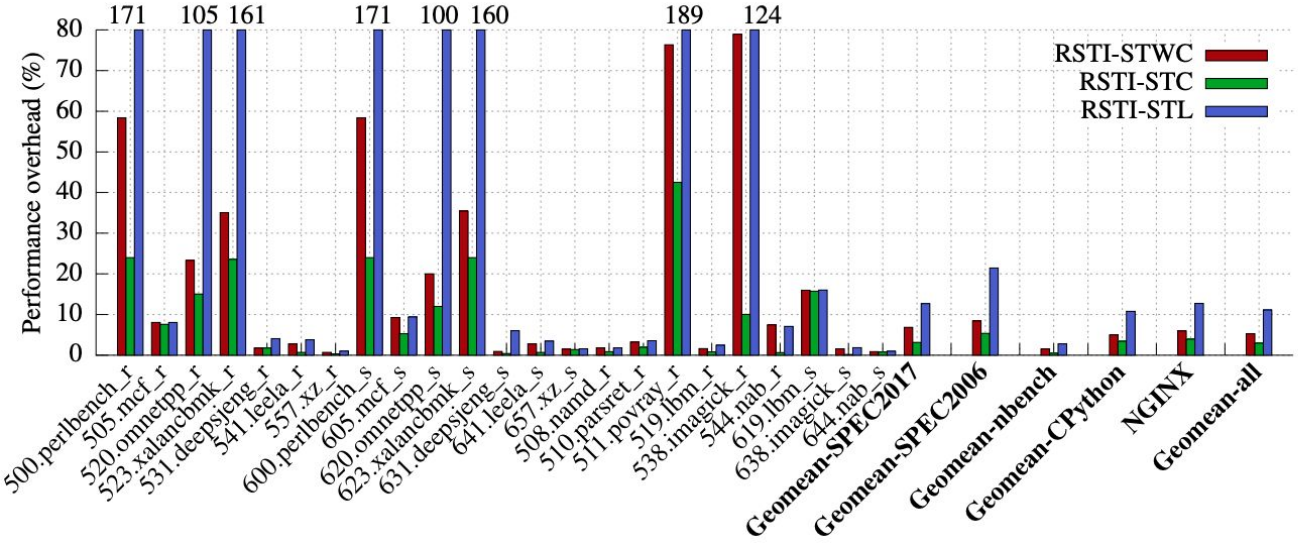
- The intuition was that this exact case of losing the original type of the pointer when a pointer-to-pointer is casted and passed as an argument to a function is rare.

- And this was confirmed by the analysis of SPEC 2006. There is a total of **7489 sites** across the benchmarks where a pointer-to-pointer is passed or loaded. Of those, only **25** meet the special criteria where the original type could be lost.

RSTI Performance evaluation

- Evaluations were run on the Apple Mac Mini M1 which supports the ARMv8.4 architecture and ARM PA. It has 4 small cores, 4 big cores, and 8GB DRAM.
- The prototype was implemented on Apple's LLVM fork. All applications were compiled with LTO and O2 for fair comparison.
- All C programs were run with real PA instructions.
- For C++ programs:
  - A correctness test was done on ARM's Fixed Virtual Platform (FVP).
  - Seven XOR (eor) instructions were used as an equivalent to one PA instruction.
  - This has been measured and confirmed in previous works

RSTI Performance evaluation



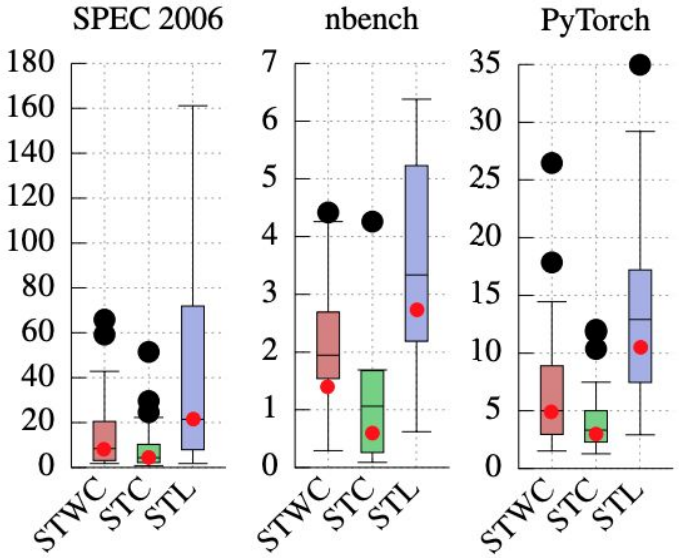
- A wide variety of benchmarks to showcase RSTI's versatility, namely: SPEC CPU 2006, SPEC CPU 2017, and nbench, in addition to PyTorch and NGINX
- The total geometric mean across all the benchmarks and applications is **5.29%**, **2.97%** and **11.12%** for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

RSTI Performance evaluation

- CPython3.9 PyTorch geometric mean is **5.01%**, **3.44%** and **10.80%** for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

- SPEC 2006 geometric mean is **8.42%**, **5.36%** and **21.47%** for RSTI-STWC, RSTI-STC and RSTI-STL, respectively.

- Pearson correlation factor of 0.75 - 0.8.



Discussion

- **Matching scope-type information.**
- **Is the PAC length enough?**
- **Protecting return addresses.**

Conclusion

- **Scope-Type Integrity (STI)** is a new defense policy that enforces a pointer to conform to the **programmer's intended usage** by utilizing **scope**, **type** and **permission** information and bringing that information back to leverage during runtime.

- **Runtime Scope-Type Integrity (RSTI)** is a robust and efficient security mechanism that protects all pointers in a program by leveraging **ARM Pointer Authentication** to enforce STI.

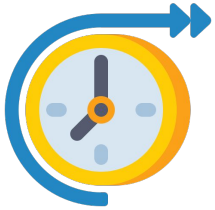
- Three RSTI defense mechanisms with varying levels of security granularity, enforcing **control-flow** and **data-flow integrity**.

- Showcased the security of RSTI against state-of-the-art attacks, and demonstrated its **low performance overhead** across a variety of benchmarks and real-life applications.

Outline

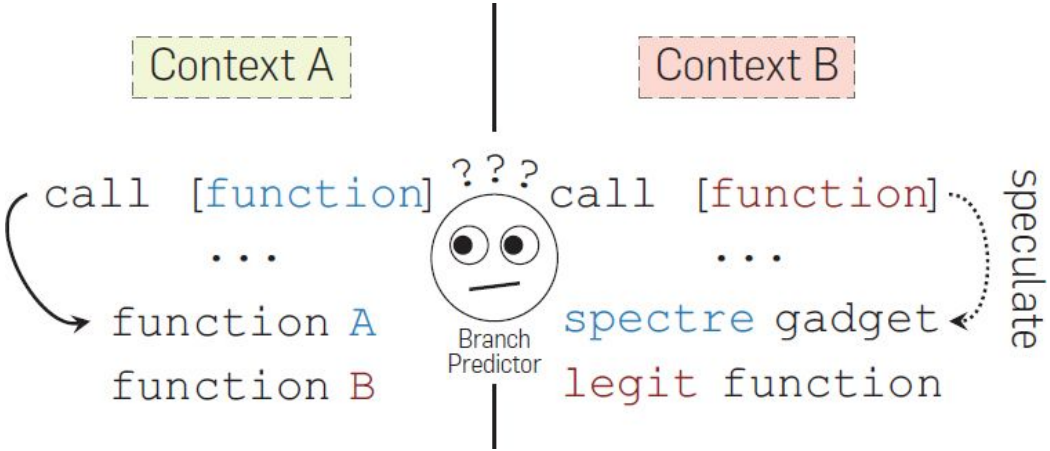
- Motivation
- Background
- Present Contributions
- **Future Work**
  - **Speculative execution memory corruption attacks**
  - **Pointer Authentication based Spectre defense**
  - **Security and Memory Safety in Energy Harvesting**
- Summary

Future work

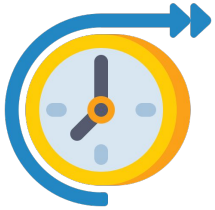


- **Speculative Execution memory corruption attacks**

```
if (p) {
    y = access(...)   speculative accessor
    transmit(y)       speculative transmitter
}
```



Future work



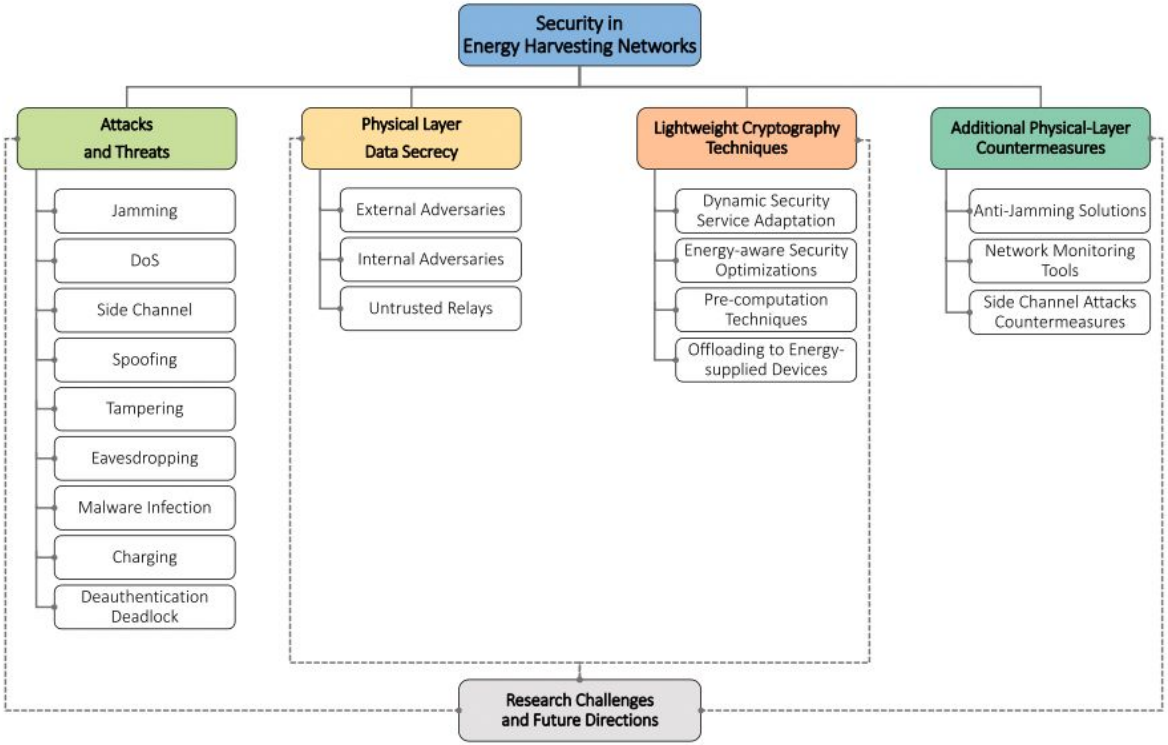
Pointer Authentication based Spectre defense (Ongoing discussion):

- Combine Pointer Authentication with lfences. lfences are instructions that prevent speculative execution.
- Instrumenting all pointers with pointer authentication instructions, similar to RSTI.
- However, the load instruction to the sensitive data would be isolated via programmer annotation and given its own unique modifier and an lfence instruction added to it.
- Another avenue of that is being investigated is possible defenses against speculative execution attacks in Rust.

Future work

Security and Memory safety in Energy Harvesting:

- Energy harvesting processes
- Energy harvesting capabilities are appealing for a variety of applications, such as Wireless Body Area Networks (WBANs) and IoT.



Outline

- Motivation
- Background
- Present Contributions
- Future Work
- **Summary**

Summary of Research

- Design **practical** and **efficient** defense mechanisms against **control-flow hijacking**, **data-oriented** and **speculative execution** attacks.
- Utilization of **hardware security features**.

Past work: PACTight

- Three **pointer integrity** properties
- Completely prevent **control-flow hijacking** attacks
- Low runtime overhead with high security guarantees

RSTI Key Contributions

- Bring back the programmer's intent to the runtime.
- Protect all pointers in a program.
- Low runtime overhead with high security guarantees

Summary of Research

Publications:

- **Tightly Seal Your Sensitive Pointers with PACTight**
  Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, Changwoo Min
  In Proceedings of the 31st USENIX Security Symposium (USENIX Security 2022)

- **Enforcing C/C++ Type and Scope at Runtime for Control-Flow and Data-Flow Integrity**
  Mohannad Ismail, Christopher Jelesnianski, Yeongjin Jang, Changwoo Min, Wenjie Xiong
  Under submission at the 29th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)

Summary of Research

Other Publications:

- **Protect the System Call, Protect (Most of) the World with BASTION**
  Christopher Jelesnianski, **Mohannad Ismail**, Yeongjin Jang, Dan Williams and Changwoo Min
  In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023)
- **VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks**
  Mohannad Ismail+, Jinwoo Yom+, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min
  In Proceedings of Conference on Computer and Communications Security (CCS 2021)
- **Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores**
  Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, **Mohannad Ismail**, Sunny Wadkar, Dongyoon Lee, and Changwoo Min
  In Proceedings of ACM Symposium on Operating Systems Principles (SOSP 2021)
- **POSEIDON: Safe, Fast and Scalable Persistent Memory Allocator**
  Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, **Mohannad Ismail**, and Changwoo Min
  In Proceedings of the 12th Annual Non-Volatile Memories Workshop (NVMW 2021)
- **POSEIDON: Safe, Fast and Scalable Persistent Memory Allocator**
  Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, **Mohannad Ismail**, and Changwoo Min
  In Proceedings of the 21st ACM/IFIP International Middleware Conference (Middleware 2020)

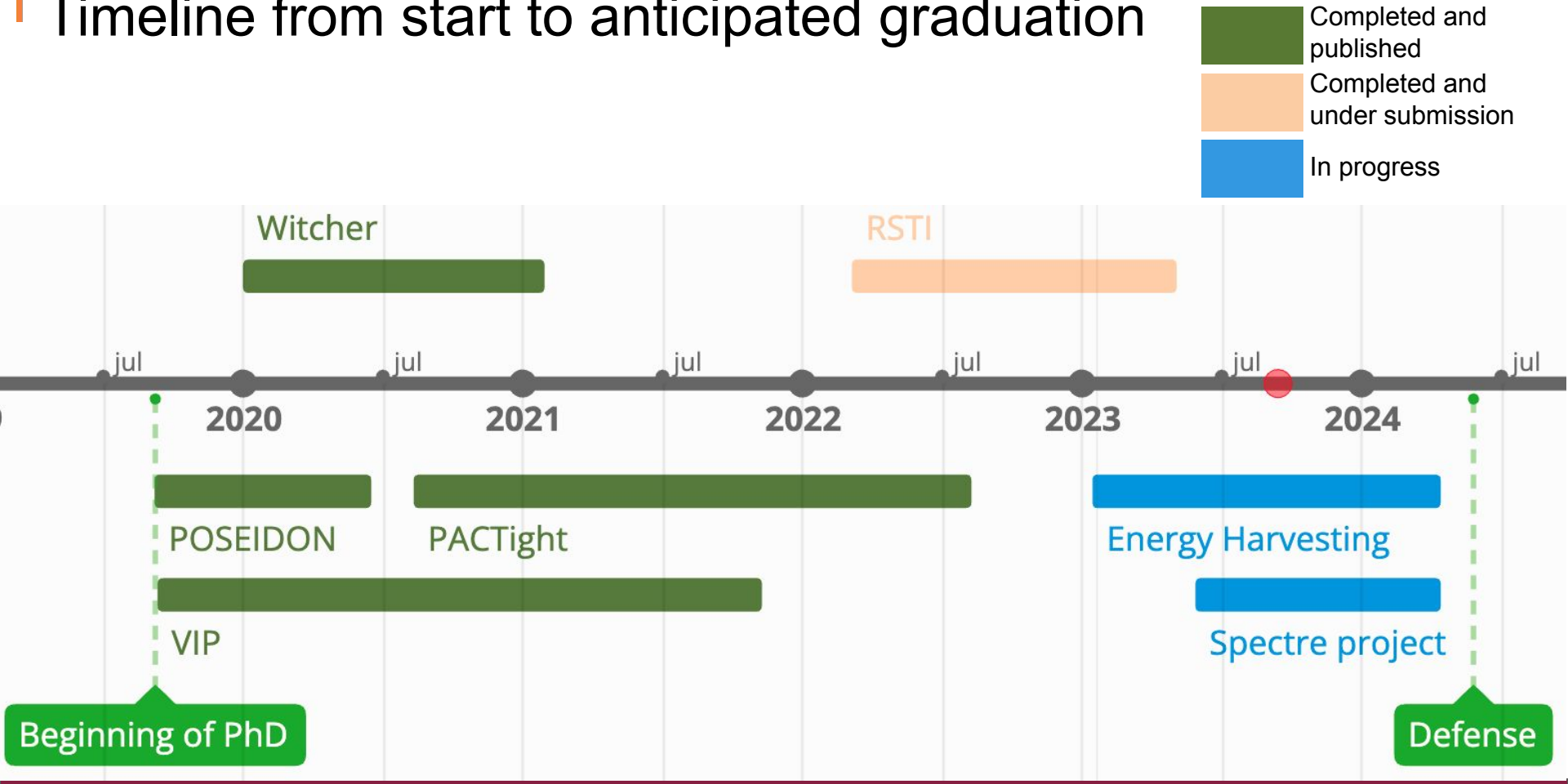
Summer internship experience

- **Intern, Member Of Technical Staff**
  VMware
  Member of the ESXi-ARM team
  Dates Employed: May 2021 – Aug 2021

- **Intern, Member Of Technical Staff**
  VMware
  Return offer, Member of the ESXi-ARM team
  Dates Employed: May 2022 – Aug 2022

- **Intern, Software Engineer**
  Apple
  Member of the security team at the Vision Products Group (VPG)
  Dates Employed: May 2023 – Aug 2023

Timeline from start to anticipated graduation



Timeline for the Spectre project

- Idea and design
  - June 2023 - October 2023
- Implementation and early evaluation
  - Late October 2023 - Late January 2024
- Evaluation and Bug fixing
  - February 2024 - March 2024
- Writing and defense
  - Early March 2024 - Late April 2024

Expected dissertation outline

1. Introduction
   a. Motivation
   b. Contributions

2. Background
   a. Memory corruption attacks
   b. ARM Pointer authentication
   c. Current defenses
   d. Past work: PACTight

3. Runtime Scope Type Integrity

4. Spectre defense mechanism

5. Security in Energy-Harvesting Systems

6. Conclusions

Effective and Practical Defenses Against Memory Corruption and Transient Execution Attacks with Hardware Security Features

Mohannad Ismail

Thank you!

Questions?

Backup Slides

But why not use memory-safe languages?

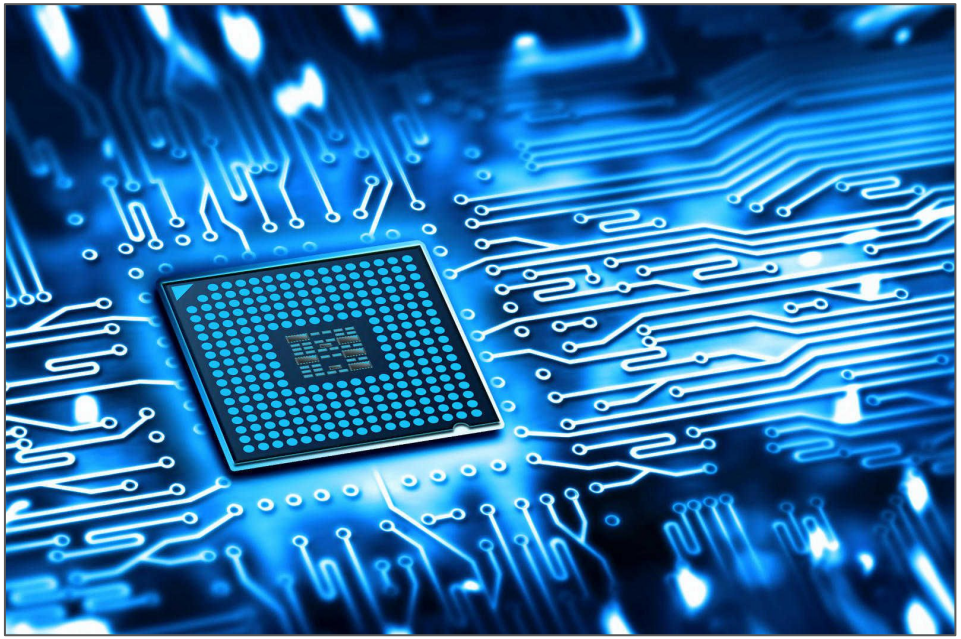
- C and C++ are memory unsafe languages, and their semantics are the root of the memory safety issues
- There's a debate that has been going for some time on whether or not to keep using them.
- Many are proposing to switch to memory safe languages such as Rust.
- But it isn't that simple.

C/C++ is here to stay.

- Performance

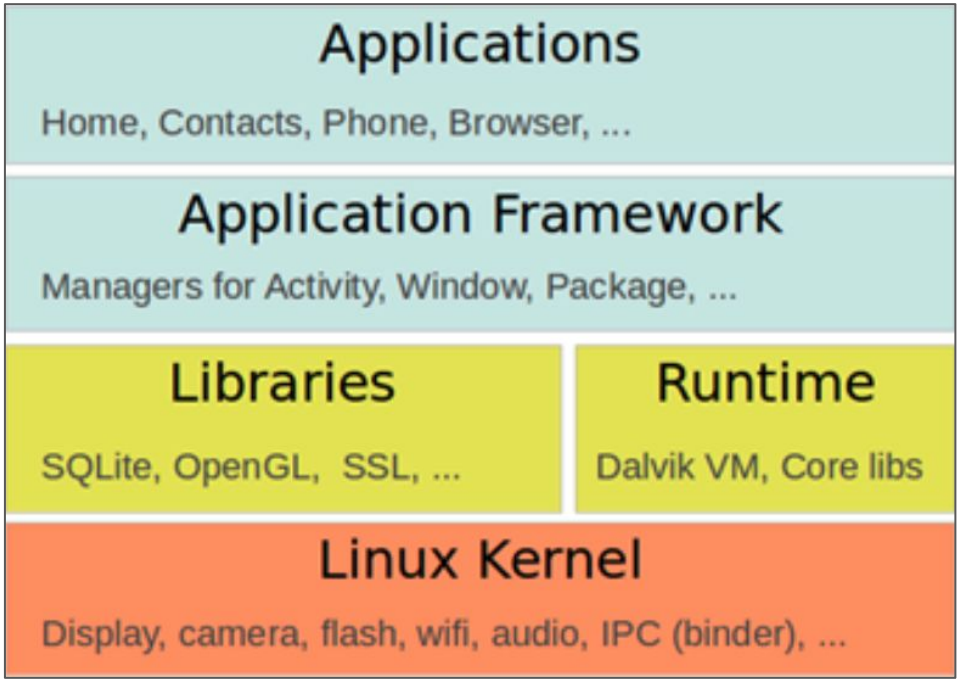
C/C++ is here to stay.

- Performance
- Communication.



C/C++ is here to stay.

- Performance
- Communication.
- Completeness.



C/C++ is here to stay.

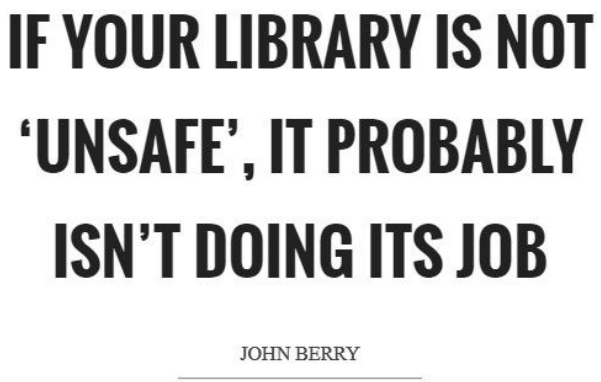
- Performance
- Communication.
- Completeness.
- Maturity

C/C++ is here to stay.

- Performance
- Communication.
- Completeness.
- Maturity
- Legacy code

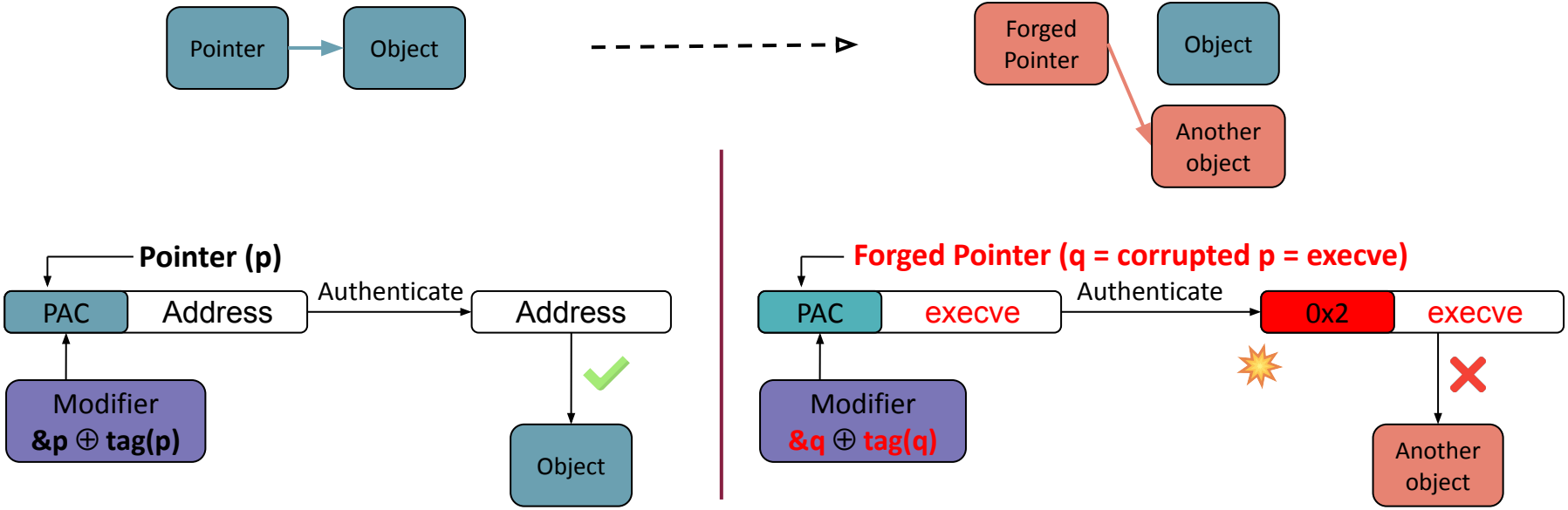
C/C++ is here to stay.

- Performance
- Communication.
- Completeness.
- Maturity
- Legacy code
- Type safe languages still rely on unsafe libraries



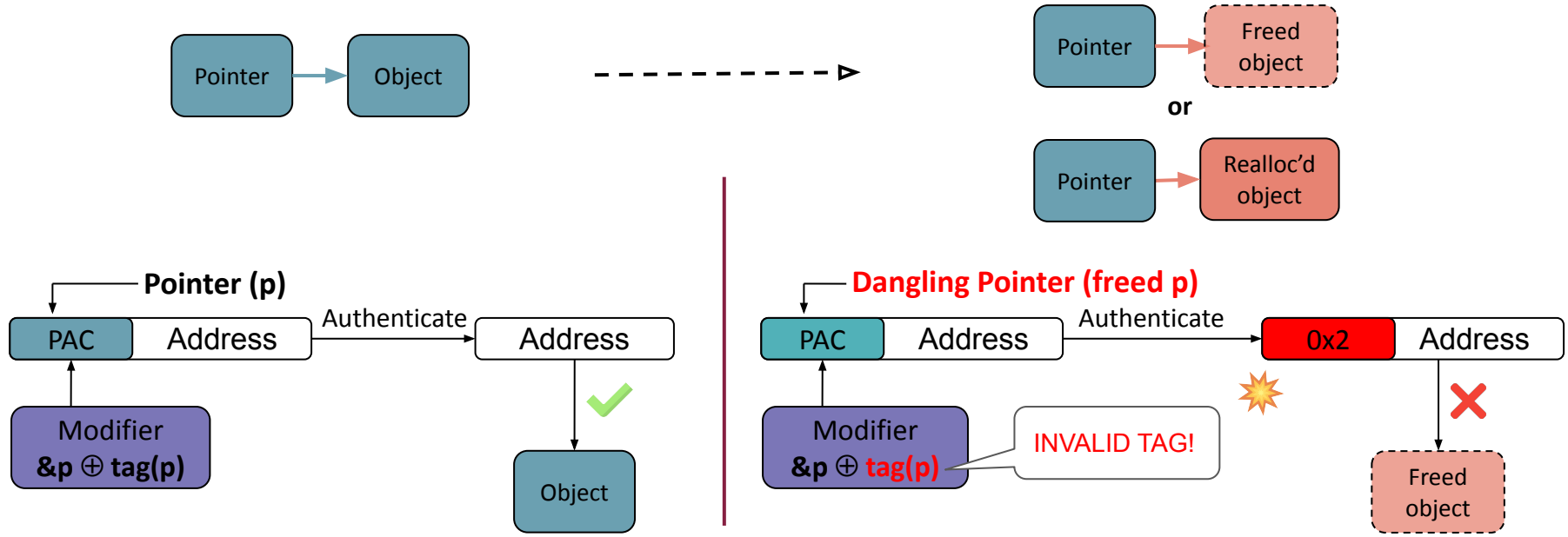
PACTight Design: Enforcing the properties

Unforgeability: 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.

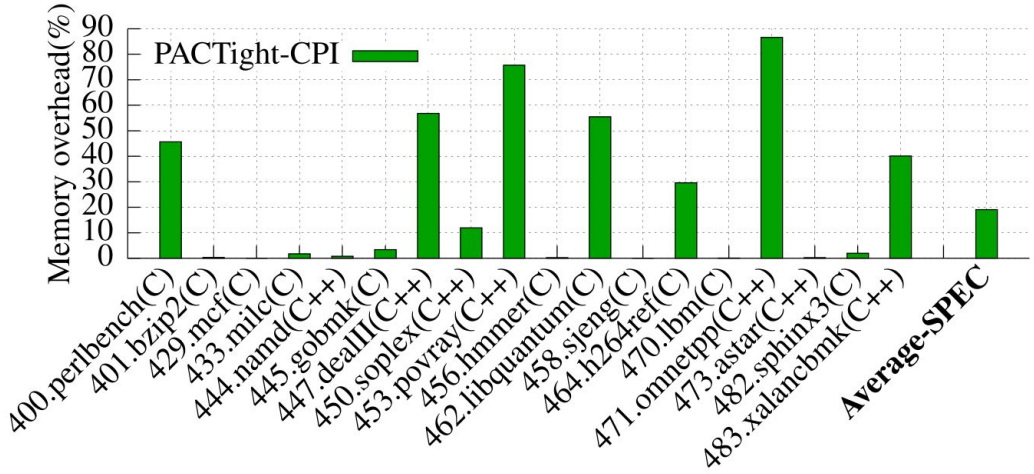


PACTight Design: Enforcing the properties

Non-dangling: 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.



Evaluation: Memory overhead

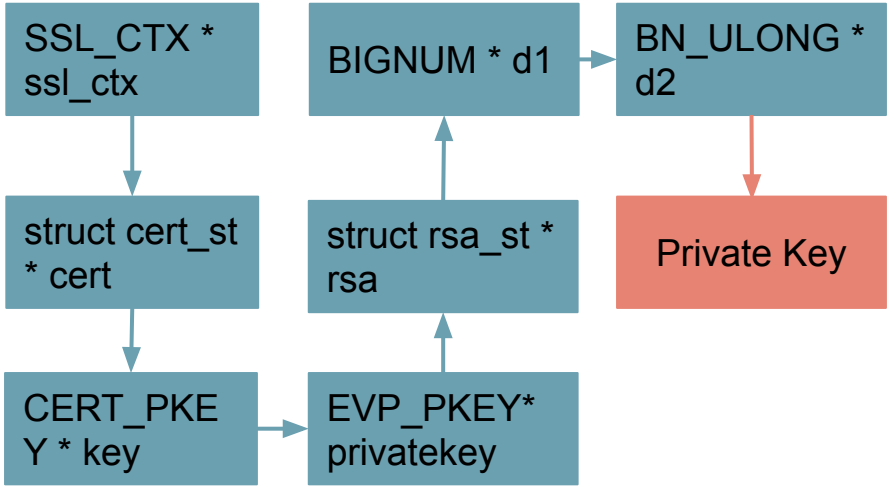


We ran the SPEC benchmarks with the PACTight-CPI protection:

- 19% memory overhead on average.

Data oriented attacks are becoming more sophisticated!

Data-oriented attacks are dangerous memory corruption attacks



Data-oriented Programming (DOP) Attack

RSTI: Design philosophy

- The goal of the defense mechanisms is ensuring **correct** and **proper** execution of a program.
- Meanwhile, the developer of the program has put much information whilst they were writing the program to make sure that it executes in a certain way.
  - All this information is, unfortunately, **lost** at runtime.
- So why not use this **information** as the **security context** that we need, and propagate it to the **runtime** in some way?
- If we can ensure that the program executes in the way that the **developer intended** even when an attacker is present, then the program would not be compromised, since an attacker relies on the anomalous execution of a program by exploiting vulnerabilities.

RSTI: Extracting scope, type and permission

```
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

Type

RSTI: Extracting scope, type and permission

```
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

Scope

RSTI: Extracting scope, type and permission

```
// C code
int main(void){
        const void *cp = malloc (sizeof(void));
}
```

```
//LLVM IR
%2 = alloca i32*, align 8
call void @llvm.dbg.declare(
                metadata i32** %2, // Type
                metadata !13, !dbg !16
%3 = call i8* @malloc(i64 1) #3, !dbg !17
%4 = bitcast i8* %3 to i32*, !dbg !17
store i32* %4, i32** %2, align 8, !dbg !16
```

```
// LLVM IR debug info
!9 = distinct !DISubprogram(name: "main", scope: !6,
        file: !6, line: 14, type: !10, scopeLine: 15
!13 = !DILocalVariable(name: "cp", scope: !9,
        file: !6, line: 16, type: !14) // Scope
!14 = !DIDerivedType(tag: DW_TAG_pointer_type,
        baseType: !15, size: 64)
!15 = !DIDerivedType(tag: DW_TAG_const_type,
        baseType: !12) // Permission
!16 = !DILocation(line: 16, column: 13, scope: !9)
```

Permission

RSTI defense mechanisms

- We propose three RSTI defense mechanisms, each with its own distinct view of the RSTI-type

1. **RSTI-STWC (Scope-Type Without Combining)** authenticates and re-signs pointers when casts happen.

2. **RSTI-STC (Scope-Type with Combining)** combines compatible types together so that it wouldn't need to re-sign pointers when pointer casts happen.

3. **RSTI-STL (Scope-Type with Location)** combines the location, i.e., address, of the pointer (&p) with the scope-type information.

RSTI defense mechanisms

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

(a) RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M2);
19
20
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*, void*	main,foo2	R/W	M1
void*	main	R	M2

(b) RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const = malloc(sizeof(
17                                  void))
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const, 2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   foo2((void*) c);
22   ...}
```

Type	Scope	Permission	RSTI Type
ctx*	main	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3
ctx*	foo	R/W	M4
ctx*	bar	R/W	M5

(c) RSTI-STL

RSTI Enforcement details: Pointer-to-pointer handling

- Not all pointer-to-pointer types are covered. Only ones that are lost when the pointer type goes out of scope, (e.g., being casted and passed as a function argument) and thus cannot be statically detected.
- The IR provides sufficient information to handle pointer-to-pointer cases that do not fall into this category. Thus, we do not need to instrument every pointer-to-pointer with this mechanism.
- RSTI uses a runtime library to handle pointer-to-pointer. Its functions are:
  - **pp_add:** This is called when a casted double pointer function argument is detected. It adds the FE to the metadata along with its RSTI-type.
  - **pp_sign:** This is called before a store of a casted double pointer function argument. It signs the pointer with a PAC based on the RSTI-type and metadata
  - **pp_auth:** This is called before a load of a signed casted double pointer. It authenticates that pointer with the RSTI-type and the original type obtained from the metadata
  - **pp_add_tbi:** This is called following pp_sign. It adds the CE to the top bits of the pointer so that they can be used to obtain the original type

Analysis on RSTI instrumentation: Equivalence Class

- **Equivalence Class of Type (EC_T)**: We refer to the number of basic types in each RSTI-type as Equivalence Class of Type (EC_T).

- **Equivalence Class of Variable (EC_V)**: We refer to the number of variables in each RSTI-type as Equivalence Class of Variable (EC_V).

Analysis on RSTI instrumentation: Equivalence Class

- Now we want to showcase the usefulness of the RSTI-type information. We define some terminology first:
  - **Number of types in program (NT)**: This is the number of basic types a program has, such as int*, void*, etc.
  - **Number of RSTI-types (RT)**: This refers to the actual types that are enforced by the specific RSTI mechanism.
  - **Equivalence Class of Type (EC_T)**: We refer to the number of basic types in each RSTI-type as Equivalence Class of Type (EC_T).
  - **Equivalence Class of Variable (EC_V)**: We refer to the number of variables in each RSTI-type as Equivalence Class of Variable (EC_V).

Analysis on RSTI instrumentation: Equivalence Class

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

(a) RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M2);
19
20
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*, void*	main,foo2	R/W	M1
void*	main	R	M2

(b) RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const = malloc(sizeof(
17                                  void))
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const, 2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   foo2((void*) c);
22   ...}
```

Type	Scope	Permission	RSTI Type
ctx*	main	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3
ctx*	foo	R/W	M4
ctx*	bar	R/W	M5

(c) RSTI-STL

Analysis on RSTI instrumentation: Equivalence Class

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6    // llvm.ptrauth.auth(v_ctx, 2, M2);
7    // llvm.ptrauth.sign(v_ctx, 2, M1);
8    foo((ctx*) v_ctx);
9    // llvm.ptrauth.auth(v_ctx, 2, M2);
10   // llvm.ptrauth.sign(v_ctx, 2, M1);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M3);
19   // llvm.ptrauth.auth(c, 2, M1);
20   // llvm.ptrauth.sign(c, 2, M2);
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*	main,foo, bar,foo2	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3

(a) RSTI-STWC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5
6
7
8    foo((ctx*) v_ctx);
9
10
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // llvm.ptrauth.sign(c, 2, M1);
15
16   const void* v_const = malloc(sizeof(
17                                  void));
18   // llvm.ptrauth.sign(v_const, 2, M2);
19
20
21   foo2((void*) c);
22   ... }
```

Type	Scope	Permission	RSTI Type
ctx*, void*	main,foo2	R/W	M1
void*	main	R	M2

(b) RSTI-STC

```
1  // key = 2 (for pacda/autda)
2  typedef struct{ void (*send_file)(
3                          int x); } ctx;
4  void foo2(void* v_ctx){
5    // M2 = M2 ^ &v_ctx
6    // llvm.ptrauth.sign(v_ctx, 2, M2);
7    // llvm.ptrauth.auth(v_ctx, 2, M2);
8    foo((ctx*) v_ctx);
9
10   // llvm.ptrauth.auth(v_ctx, 2, M2);
11   bar((ctx*) v_ctx); }
12 int main(){
13   ctx* c = malloc(sizeof(*c));
14   // M1 = M1 ^ &c
15   // llvm.ptrauth.sign(c, 2, M1);
16   const void* v_const = malloc(sizeof(
17                                  void))
18   // M3 = M3 ^ &v_const
19   // llvm.ptrauth.sign(v_const, 2, M3);
20   // llvm.ptrauth.auth(c, 2, M1);
21   foo2((void*) c);
22   ...}
```

Type	Scope	Permission	RSTI Type
ctx*	main	R/W	M1
void*	foo2	R/W	M2
void*	main	R	M3
ctx*	foo	R/W	M4
ctx*	bar	R/W	M5

(c) RSTI-STL

Analysis on RSTI instrumentation: Equivalence Class

BM	NT	RT		NV	Largest EC_V		Largest EC_T	
		STC	STWC		STC	STWC	STC	STWC
perlbench	155	318	722	2939	198	82	33	1
bzip2	25	31	55	122	32	13	7	1
mcf	12	35	40	95	9	8	2	1
milc	55	154	195	440	54	18	18	1
namd	30	73	100	230	23	23	10	1
gobmk	120	216	417	1057	111	46	25	1
dealII	2546	4528	8878	21018	676	44	192	1
soplex	129	970	1690	3399	137	27	66	1
povray	282	620	1446	3791	229	25	76	1
hmmer	90	198	405	973	56	24	16	1
libquantum	13	33	44	58	9	4	5	1
sjeng	29	47	73	130	19	9	7	1
h264ref	116	252	354	727	48	23	15	1
lbm	14	14	20	33	12	7	4	1
omnetpp	255	558	1241	2458	94	26	31	1
astar	36	59	98	156	18	11	12	1
sphinx3	88	188	321	686	36	20	12	1
xalancbmk	2558	7503	14073	32097	603	122	206	1