PACTight: Tightly Seal Sensitive Pointers with Pointer Authentication

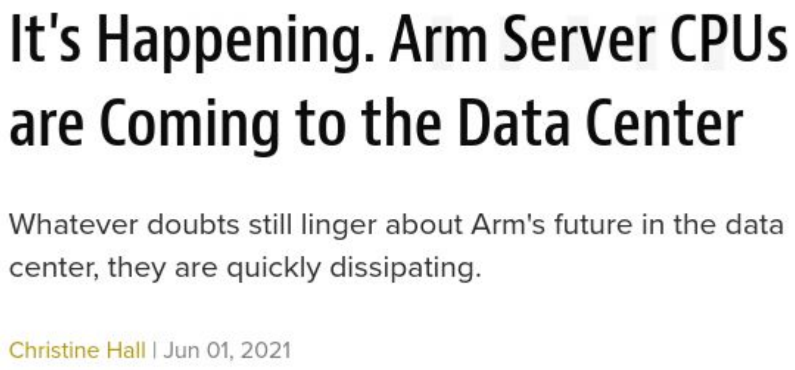
Mohannad Ismail

Master's Defense

VIRGINIA TECH™

ARM is becoming popular

- In recent years, the ARM processor architecture started penetrating into the data center market and mainstream desktop markets (Apple M1) beyond the mobile/embedded segments.
- This opens a new realm in terms of security attacks against ARM, increasing the importance of having an effective and efficient defense mechanism for ARM.



It's Happening. Arm Server CPUs are Coming to the Data Center

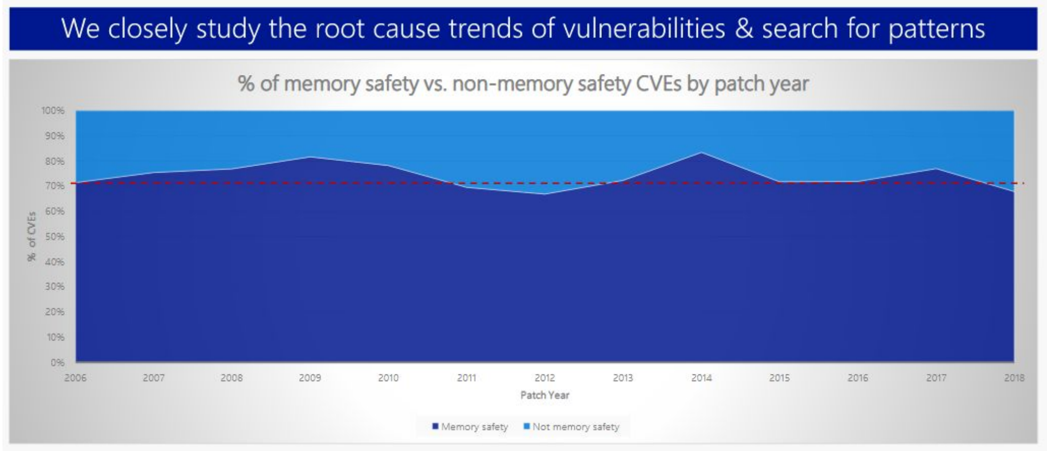
Whatever doubts still linger about Arm's future in the data center, they are quickly dissipating.

Christine Hall | Jun 01, 2021

Memory corruption vulnerability is the root of all EVIL

- Microsoft reported that **70%** of all security bugs are due to various **memory safety** issues.

- Control-flow hijacking attacks and use-after-free attacks are some of the most serious types of memory corruption attacks.



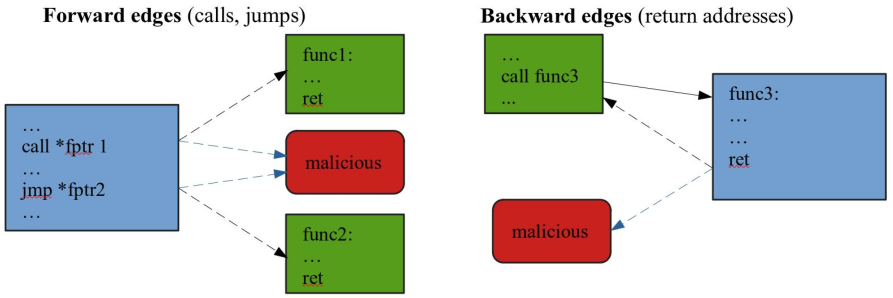
How will we overcome this problem?

- Problem:
  - Memory corruption vulnerabilities are still a very prevalent problem, particularly control-flow hijacking and use-after-free memory attacks, with most defenses focused on x86.
  - However, ARM has made efforts from its side and introduced Pointer Authentication (PA), a hardware security feature to protect pointers with cryptographic primitives.
  - Current defenses that utilize PA either have limited coverage or high performance overhead.
- Goal:
  - Our goal is to develop an efficient defense mechanism for ARM, utilizing PA, to protect pointers against memory corruption attacks.
- Approach:
  - The main strength in PA is the use of a salt or modifier. We aim to provide a unique modifier that strengthens the pointer protection.
- Outcome:
  - We developed PACTight, a PA based defense with four defense mechanisms to tightly seal sensitive pointers with high coverage and security guarantees, as well as low overhead.

Outline

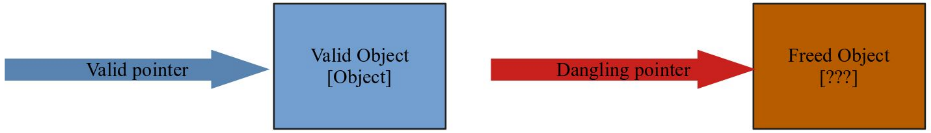
Control-flow hijacking and use-after-free attacks are critical

- Control-flow hijacking attacks aim to subvert the control-flow of a program by carefully corrupting code pointers, such as return addresses and function pointers.

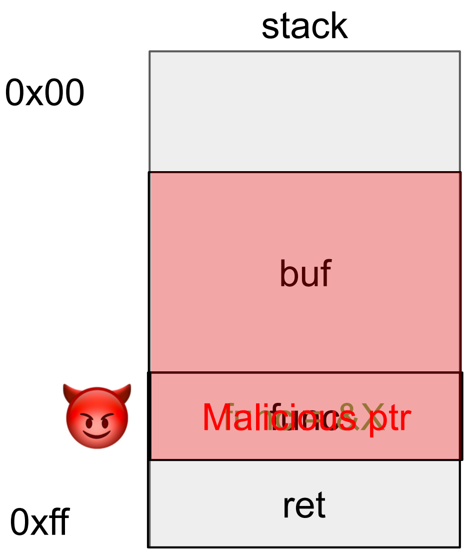


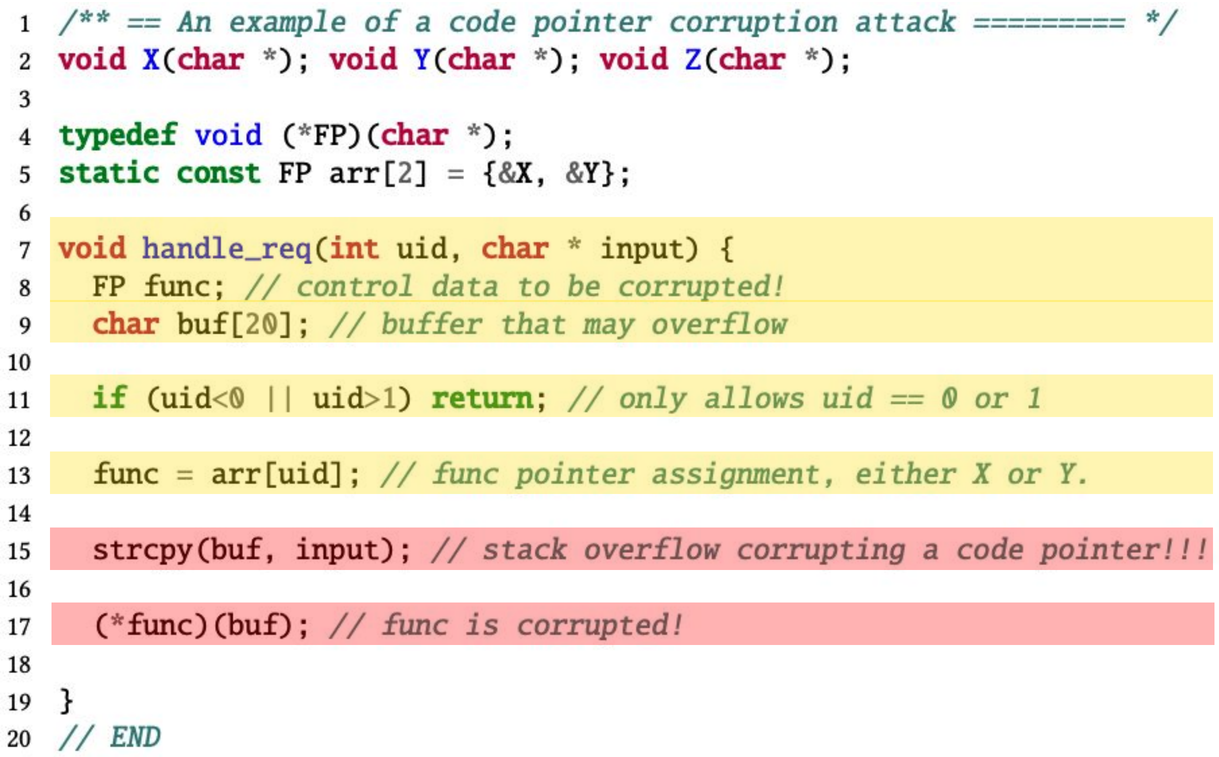
- Use-After-Free (UAF) is a vulnerability related to incorrect use of dynamic memory during program operation.

- If after freeing a memory location, a program does not clear the pointer to that memory, an attacker can use the error to hack the program.



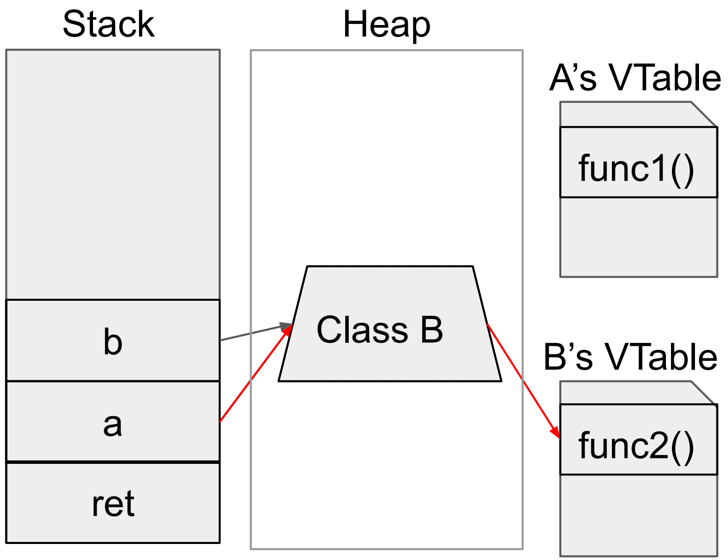
Vulnerable Control Data Example





```
/** == An example of a code pointer corruption attack ========= */
void X(char *); void Y(char *); void Z(char *);

typedef void (*FP)(char *);
static const FP arr[2] = {&X, &Y};

void handle_req(int uid, char * input) {
  FP func; // control data to be corrupted!
  char buf[20]; // buffer that may overflow

  if (uid<0 || uid>1) return; // only allows uid == 0 or 1

  func = arr[uid]; // func pointer assignment, either X or Y.

  strcpy(buf, input); // stack overflow corrupting a code pointer!!!

  (*func)(buf); // func is corrupted!

}
// END
```

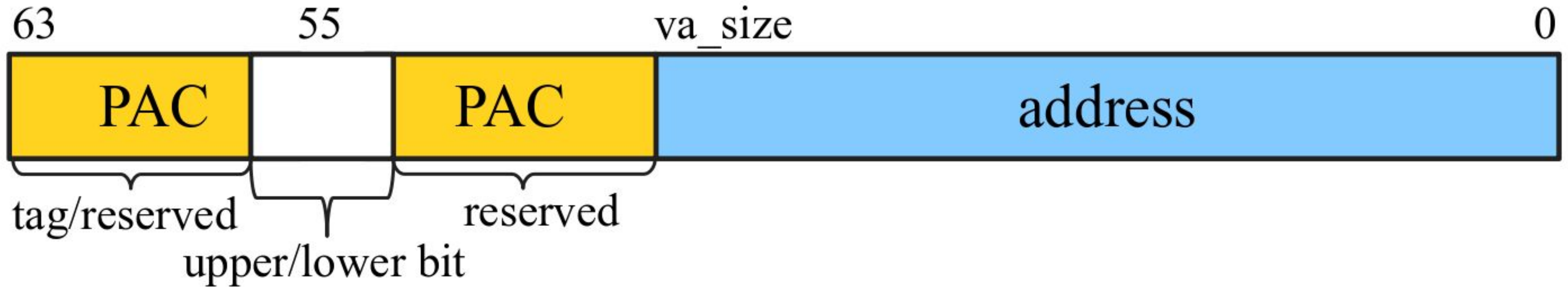
C++ VTable pointer Use-after-free example



```
class A {
public:
    virtual void func1() {};
};
class B {
public:
    virtual void func2() {};
};

int main() {
    A *a = new A();

    delete a;

    B *b = new B();

    a->func1();
}
```

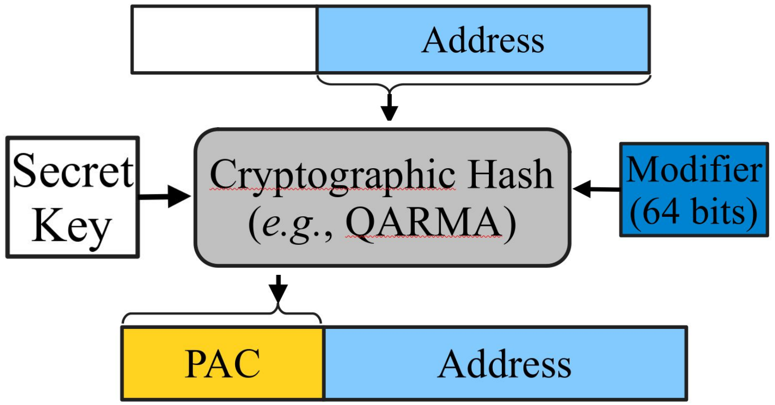
ARM’s Pointer Authentication

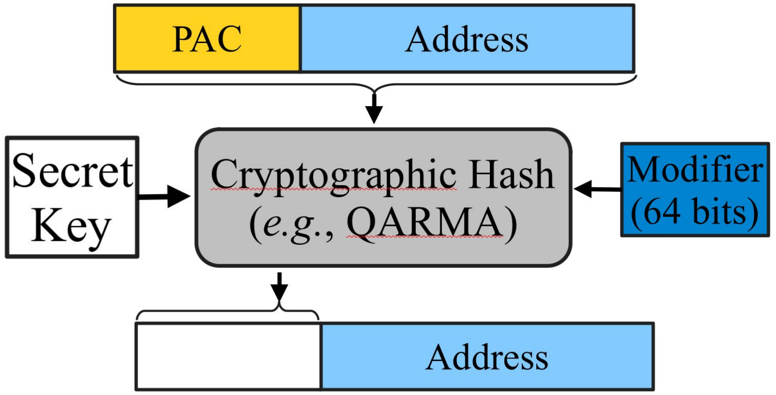
- ARMv8.3-A introduced a new hardware security feature called Pointer Authentication (PA).
- A pointer authentication code (PAC) is generated by a cryptographic hash function, as a message authentication code (MAC) to put cryptographic integrity protection on the pointers.
- The PAC is then placed on the top unused bits of a 64-bit pointer.



ARM’s Pointer Authentication

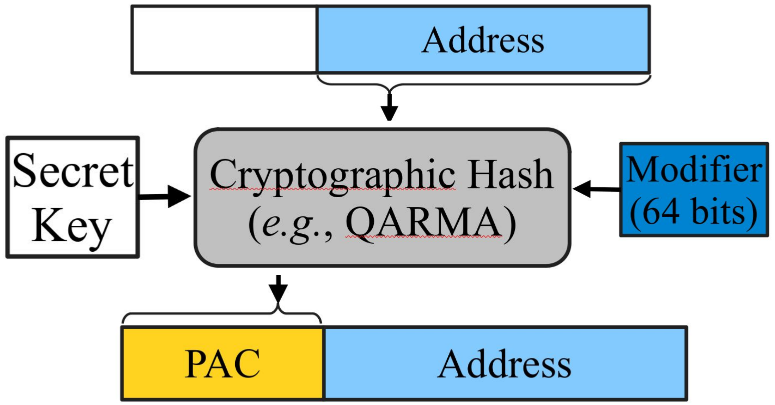
- **PAC signing**
  - PAC utilizes a cryptographic hash algorithm, namely QARMA. The algorithm takes two 64-bit values (the pointer and modifier), as well as a 128-bit key, and generates a 64-bit PAC.
- **PAC authentication**
  - Authentication of the pointer is straightforward. The cryptographic algorithm takes the pointer with the PAC and the modifier. The PAC is then regenerated and compared with the one on the passed pointer.

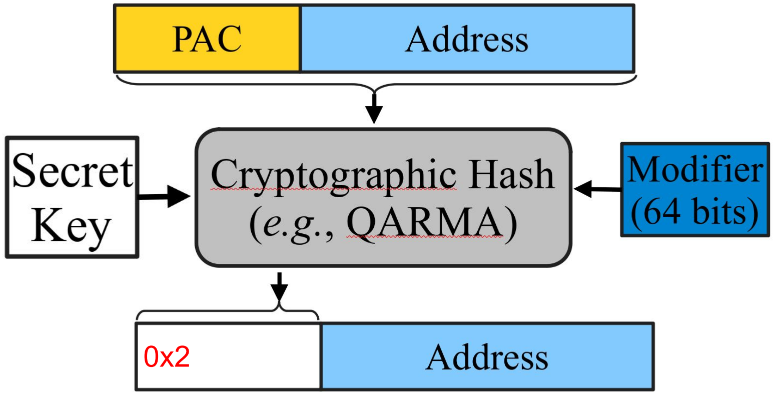




ARM’s Pointer Authentication

- **PAC signing**
  - PAC utilizes a cryptographic hash algorithm, namely QARMA. The algorithm takes two 64-bit values (the pointer and modifier), as well as a 128-bit key, and generates a 64-bit PAC.
- **PAC authentication**
  - Authentication of the pointer is straightforward. The cryptographic 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

Defense	Protection scope	Attacker abilities	PAC modifier
PARTS-CFI [1]	Return addresses and indirect code pointers	Arbitrary read-write	SP (Stack Pointer) for return addresses and type-id for indirect code pointers.
PACStack [2]	Return addresses	Arbitrary read-write	Previous return address on the stack
PTAuth [3]	Heap allocated objects	Arbitrary write	A generated object-id

[1] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In Proceedings of the 28th USENIX Security Symposium (Security)
[2] Hans Liljestrand, Lachlan J. Gunn, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack. In Proceedings of the 30th USENIX Security Symposium (Security)
[3] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In Proceedings of the 30th USENIX Security Symposium (Security)

Limitations of state-of-the-art PAC techniques

- PARTS-CFI
  - Key approach: relies on a static modifier based on the LLVM ElementType.
  - Limitation: repeated LLVM ElementType, thus attackers can reuse the PAC generated for one in the context of using the other.
- PACStack
  - Key approach: protecting return addresses recursively.
  - Limitation: relies on the presence of a forward edge CFI technique.
- PTAuth
  - Key approach: protection against temporal attacks.
  - Limitations:
    - Defends only against attackers with arbitrary write
    - Vulnerable to an intra-object violation.
    - Pointers within the same object are copyable
    - Only protects the heap and doesn’t handle stack protection, even from temporal attacks

Outline

Introducing PACTight

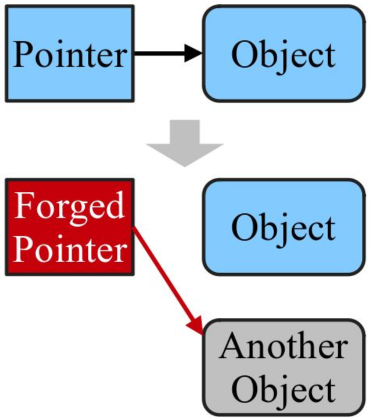
- Goal:
  - We propose PACTight, which is a PA based defense against control-flow hijacking and use-after-free attacks.

- Approach and idea:
  - Based on the limitations of prior PA approaches, we define three security properties of a pointer such that, if achieved, prevent pointers from being tampered with. They are:
    - **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.

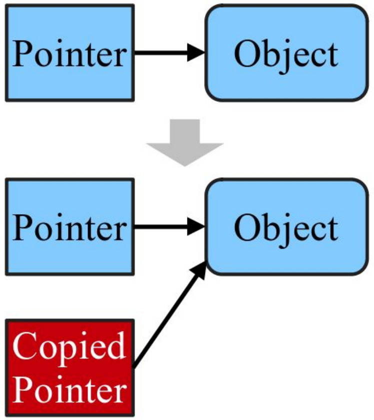
- Threat model and coverage:
  - PACTight assumes a stronger threat model such that the attacker has both arbitrary read and write capabilities
  - PACTight provides better coverage by protecting a variety of security-sensitive pointers.

Introducing PACTight

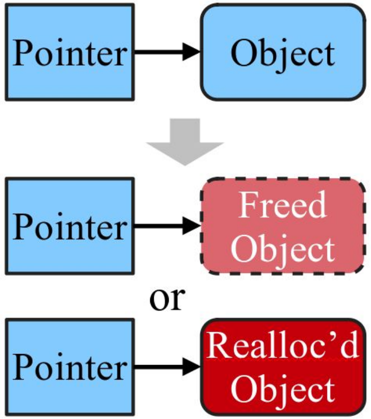
- The three properties:



Forgeability



Copyability



Dangling

PACTight overview

- PACTight enforces the three properties in order to prevent the pointers from being abused.

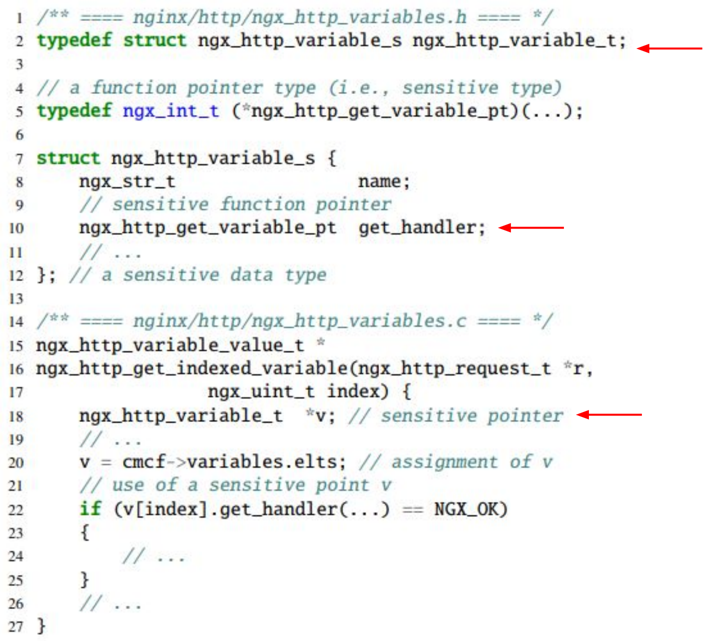
- PACTight tightly seals pointers and guarantees that a sealed pointer cannot be forged, copied, and is not dangling.

- We implement PACTight to achieve pointer integrity by protecting all sensitive pointers and provide precise, deterministic memory safety for these sensitive pointers.

- Deterministic memory safety means that we provide spatial and temporal safety for a subset of the pointers in a program, i.e. sensitive pointer.

PACTight: Sensitive pointer definition

- Sensitive pointers here refers to all code pointers and data pointers that can reach to a code pointer.
- Protecting these sensitive pointers achieves the balance between full memory safety and protecting only function pointers.
- Example of a sensitive data pointer in (simplified) NGINX source code. ngx_http_variable_t is a sensitive data type since it contains a function pointer get_handler (Line 10).



```
1  /** ==== nginx/http/ngx_http_variables.h ==== */
2  typedef struct ngx_http_variable_s ngx_http_variable_t;
3
4  // a function pointer type (i.e., sensitive type)
5  typedef ngx_int_t (*ngx_http_get_variable_pt)(...);
6
7  struct ngx_http_variable_s {
8      ngx_str_t                  name;
9      // sensitive function pointer
10     ngx_http_get_variable_pt   get_handler;
11     // ...
12 }; // a sensitive data type
13
14 /** ==== nginx/http/ngx_http_variables.c ==== */
15 ngx_http_variable_value_t *
16 ngx_http_get_indexed_variable(ngx_http_request_t *r,
17                     ngx_uint_t index) {
18     ngx_http_variable_t  *v; // sensitive pointer
19     // ...
20     v = cmcf->variables.elts; // assignment of v
21     // use of a sensitive point v
22     if (v[index].get_handler(...) == NGX_OK)
23     {
24         // ...
25     }
26     // ...
27 }
```

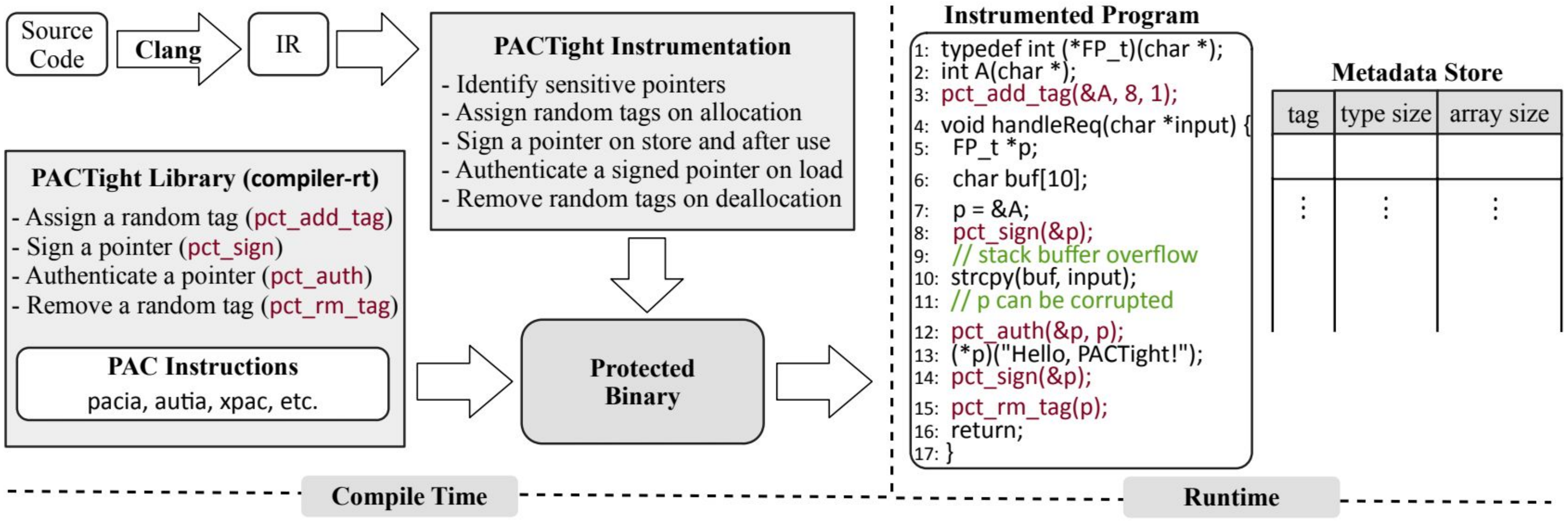
Outline

Threat model and assumptions

- Our threat model assumes a powerful adversary with read and write capabilities by exploiting input-controlled memory corruption errors in the program.
- The attacker cannot inject or modify code due to Data Execution Prevention (DEP), which is by default enabled in most modern operating systems
- We assume that the hardware and kernel are trusted, specifically that the PA secret keys are generated, managed and stored securely.
- Attacks targeting the kernel and hardware attacks, such as Spectre, are out of scope.

Outline

PACTight overall design



PACTight Instrumentation

- Identify sensitive pointers
- Assign random tags on allocation
- Sign a pointer on store and after use
- Authenticate a signed pointer on load
- Remove random tags on deallocation

PACTight Library (compiler-rt)

- Assign a random tag (pct_add_tag)
- Sign a pointer (pct_sign)
- Authenticate a pointer (pct_auth)
- Remove a random tag (pct_rm_tag)

PAC Instructions

pacia, autia, xpac, etc.

Instrumented Program

```
typedef int (*FP_t)(char *);
int A(char *);
pct_add_tag(&A, 8, 1);
void handleReq(char *input) {
  FP_t *p;
  char buf[10];
  p = &A;
  pct_sign(&p);
  // stack buffer overflow
 strcpy(buf, input);
 // p can be corrupted
 pct_auth(&p, p);
 (*p)("Hello, PACTight!");
 pct_sign(&p);
 pct_rm_tag(p);
 return;
}
```

Metadata Store

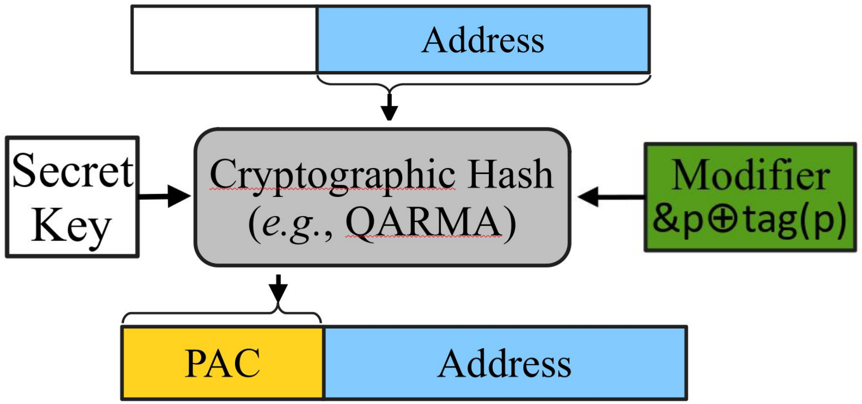
tag	type size	array size
⋮	⋮	⋮

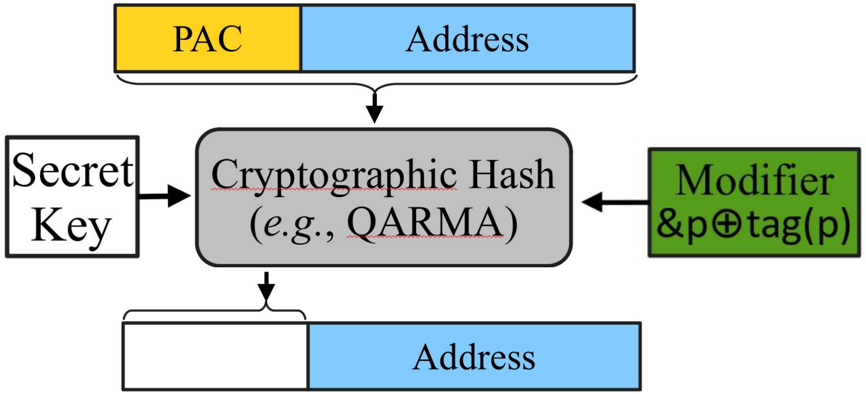
PACTight design and runtime: Enforcing the properties

- In order to enforce the three properties, PACTight relies on the PAC modifier.
- The modifier is a user-defined salt that is incorporated by the cryptographic hash into the PAC in addition to the address.
- Any changes in either the modifier or the address result in a different PAC, detecting the violation.

PACTight design and runtime: Enforcing the properties

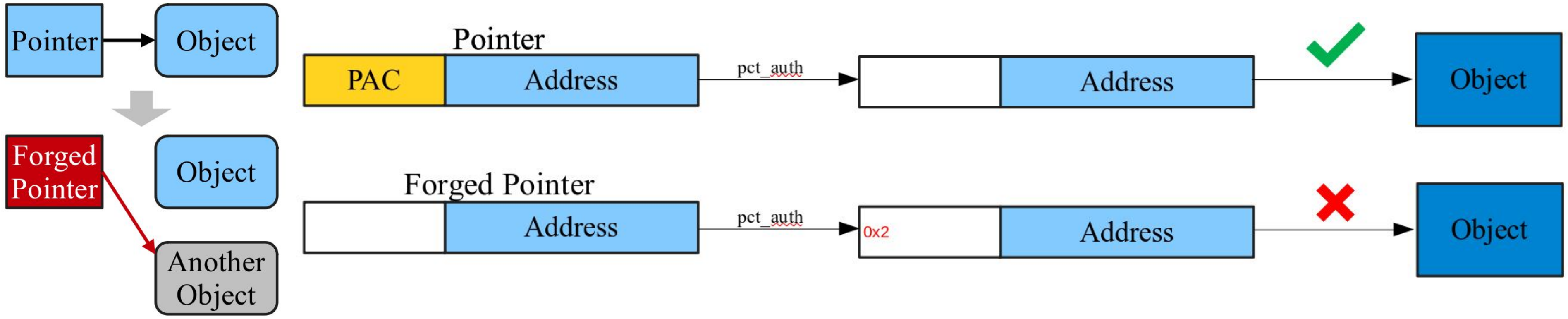
- We propose to blend the **address of a pointer (&p)** and a **random tag (tag(p))** associated with a memory object to efficiently enforce the PACTight pointer integrity property





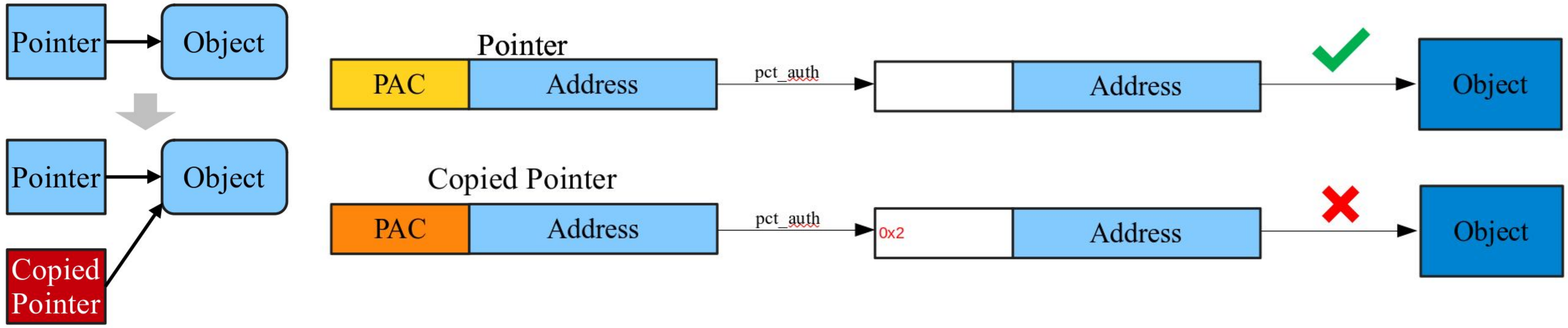
PACTight design and runtime: Enforcing the properties

- **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.



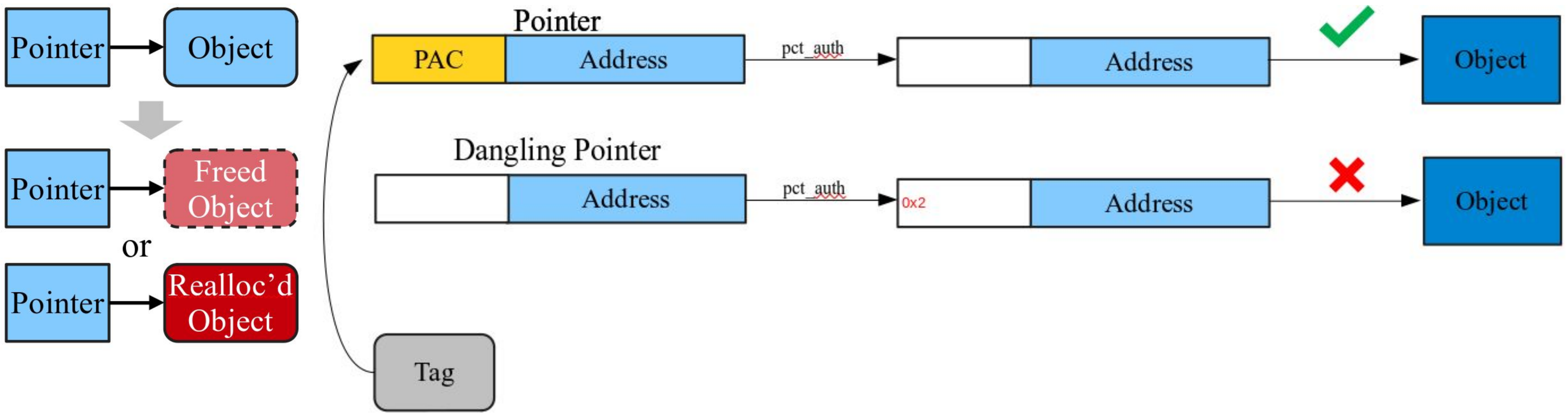
PACTight design and runtime: Enforcing the properties

- **Non-copyability:** PACTight adds the location of the pointer (&p) as a part of the modifier. This guarantees that the pointer can only be used at that specific location. Any change in the location by copying the pointer changes the modifier and thus triggers an authentication fault.



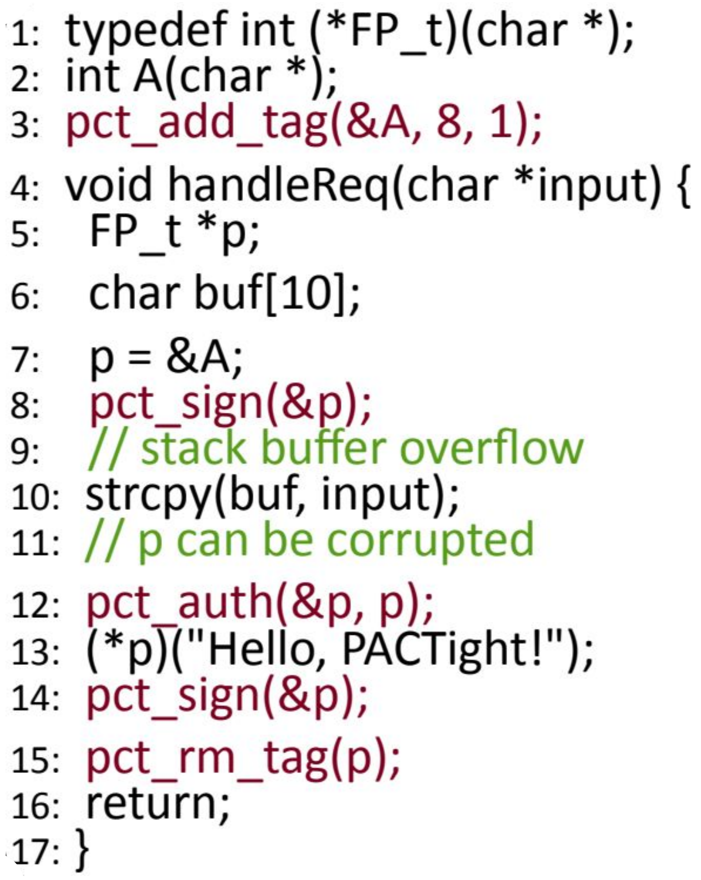
PACTight design and runtime: Enforcing the properties

- **Non-dangling:** PACTight uses a random tag ID to track the life cycle of a memory object.The life cycle of a PACTight-sealed pointer is bonded to that of a memory object.



PACTight Runtime: Runtime library

- The PACTight runtime library provides four APIs to enforce pointer integrity.
  - **pct_add_tag(p,tsz,asz)** sets the metadata for a newly allocated memory region.
  - **pct_sign(&p)** signs a pointer with the associated random tag that was generated by pct_add_tag.
  - **pct_auth(&p, p+N)** authenticates a pointer with the associated metadata.
  - **pct_rm_tag(p)** removes the metadata associated to a pointer from the metadata store.



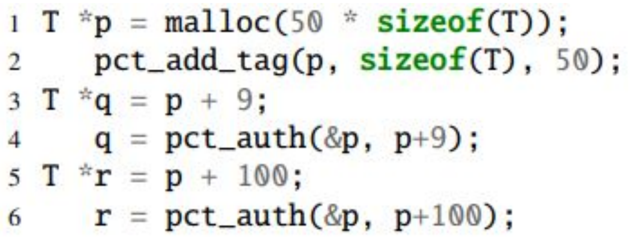
```
typedef int (*FP_t)(char *);
int A(char *);
pct_add_tag(&A, 8, 1);
void handleReq(char *input) {
  FP_t *p;
  char buf[10];
  p = &A;
  pct_sign(&p);
  // stack buffer overflow
  strcpy(buf, input);
  // p can be corrupted
  pct_auth(&p, p);
  (*p)("Hello, PACTight!");
  pct_sign(&p);
  pct_rm_tag(p);
  return;
}
```

PACTight Runtime: Pointer operations

- Since a PACTight-signed pointer has a PAC in its upper bits, care must be taken not to break the semantics of existing C/C++ pointer semantics. In particular, we take care of the following cases:
  - **PACTight-signed pointer comparison:**
    - Even if two pointers refer to the same memory address, their PACs are different since the locations of the two pointers are different (i.e., &p!=&q). Hence, PACTIGHT strips off the PAC from the PACTIGHT-signed pointer before comparison by looking for the icmp instruction.
  - **PACTight-signed pointer assignment:**
    - When assigning one signed pointer (source) to another signed pointer (target), the target pointer should be signed again with its location
  - **PACTight-signed pointer argument:**
    - There are functions that directly manipulate a pointer. For those functions, PACTIGHT strips off the PAC before passing the signed pointer as an argument.

PACTight Runtime: Pointer operations

- Since a PACTight-signed pointer has a PAC in its upper bits, care must be taken not to break the semantics of existing C/C++ pointer semantics. In particular, we take care of the following cases:
  - **PACTight-signed pointer arithmetic:** PACTight supports pointer arithmetic on arrays. In the case of arrays, all elements in the array get the same unique random tag, with the rest of the metadata keeping track of the size of an element and the number of elements in the array. This is done to efficiently enforce spatial safety.



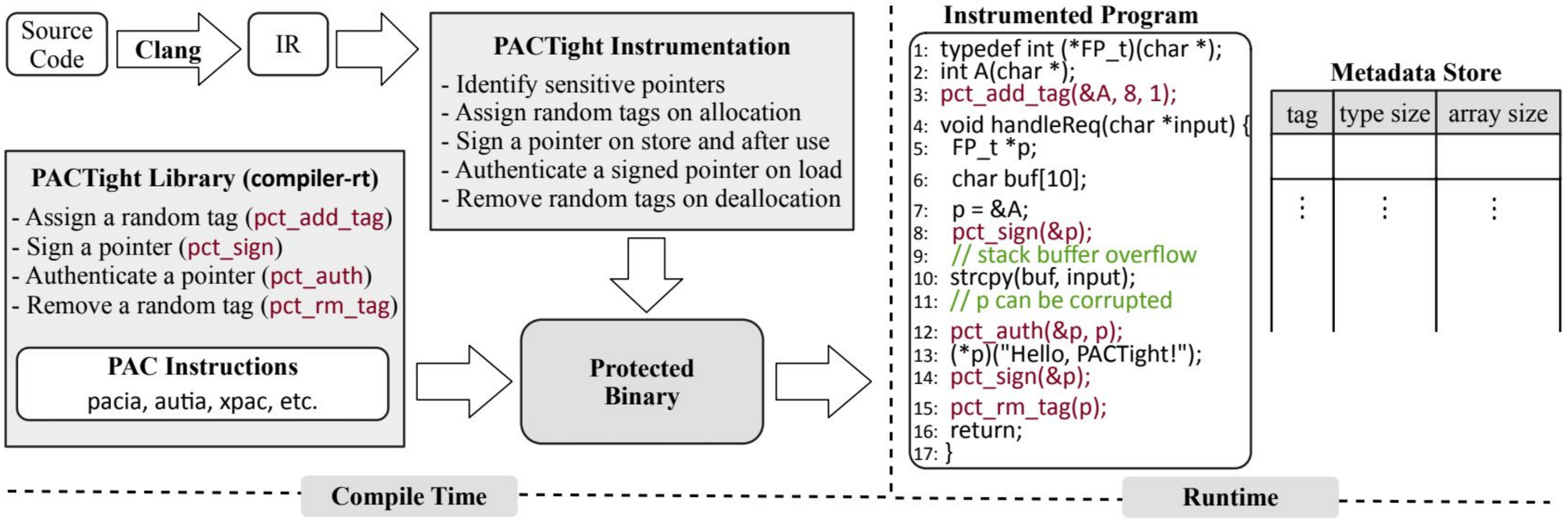
```
T *p = malloc(50 * sizeof(T));
    pct_add_tag(p, sizeof(T), 50);
T *q = p + 9;
    q = pct_auth(&p, p+9);
T *r = p + 100;
    r = pct_auth(&p, p+100);
```

  - The mechanism works the same for temporal memory safety; a freed object will not have a tag, and newly allocated object in the same location will have a different random tag. Thereby, PACTight can effectively reject spatial memory violation and temporal memory violation.

PACTight Runtime: Metadata Store

- PACTight maintains a metadata store for allocated memory objects. For each allocated memory object, the metadata store maintains a 64-bit random tag ID, the size of each individual element (or type size), and the number of elements in an array (or array size).
- Non-array objects will be treated as an array having a single element.
- We implemented the metadata store as a hash table using the address (i.e., p) as a key.
- The metadata store is allocated when the program starts and is maintained by PACTight's runtime library.

PACTight overall design


Source Code
Clang
IR
PACTight Instrumentation
- Identify sensitive pointers
- Assign random tags on allocation
- Sign a pointer on store and after use
- Authenticate a signed pointer on load
- Remove random tags on deallocation
PACTight Library (compiler-rt)
- Assign a random tag (pct_add_tag)
- Sign a pointer (pct_sign)
- Authenticate a pointer (pct_auth)
- Remove a random tag (pct_rm_tag)
PAC Instructions
pacia, autia, xpac, etc.
Protected Binary
Instrumented Program
Metadata Store
tag
type size
array size
Compile Time
Runtime

```
typedef int (*FP_t)(char *);
int A(char *);
pct_add_tag(&A, 8, 1);
void handleReq(char *input) {
  FP_t *p;
  char buf[10];
  p = &A;
  pct_sign(&p);
  // stack buffer overflow
 strcpy(buf, input);
 // p can be corrupted
 pct_auth(&p, p);
 (*p)("Hello, PACTight!");
 pct_sign(&p);
 pct_rm_tag(p);
 return;
}
```

Outline

PACTight Defense Mechanisms

- Now we present the PACTight defense mechanisms built on top of the PACTight design and runtime.

- The PACTight compiler passes automatically instrument 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)
  - Return address protection (backward edge protection).

PACTight Defense Mechanisms: PACTight-CFI

- PACTight-CFI guarantees forward-edge control-flow integrity by ensuring the PACTight pointer integrity properties for all code pointers.
- It authenticates the PAC on a function pointer at legitimate function call sites. At all other sites, the code pointer is sealed with the PACTight signing mechanism so it cannot be abused.
- PACTight-CFI identifies all code pointers using LLVM type information. PACTight-CFI also recursively looks through all elements inside a composite type.
- PACTight-CFI enforces the PACTight pointer integrity properties on code pointers and this implies that the equivalence class (EC) size (i.e., the number of allowed legitimate targets at one call site) is always one

PACTight Defense Mechanisms: PACTight-VTable

- C++ relies on virtual functions to achieve dynamic polymorphism. At every virtual function call, a proper function is used in accordance with the object type.

- The mapping of an object type to a virtual function is done by the use of a virtual function table (VTable) pointer, which is a pointer to an array of virtual function pointers per object type.

- PACTight-VTable identifies a VTable pointer in a C++ object by analyzing types in LLVM. It investigates all composite types and checks if it is a class type having one or more virtual functions.

PACTight Defense Mechanisms: PACTight-CPI

- PACTight-CPI increases the coverage of PACTight-CFI to guarantee integrity of all sensitive pointers.

- Sensitive pointers are all code pointers (i.e., PACTight-CFI coverage) and all data pointers that point to code pointers.

- PACTight-CPI expands the type analysis of PACTight-CFI to include sensitive data pointers as well as code pointers, in addition to return addresses and C++ VTable pointers.

PACTight Defense Mechanisms: PACTight-RET

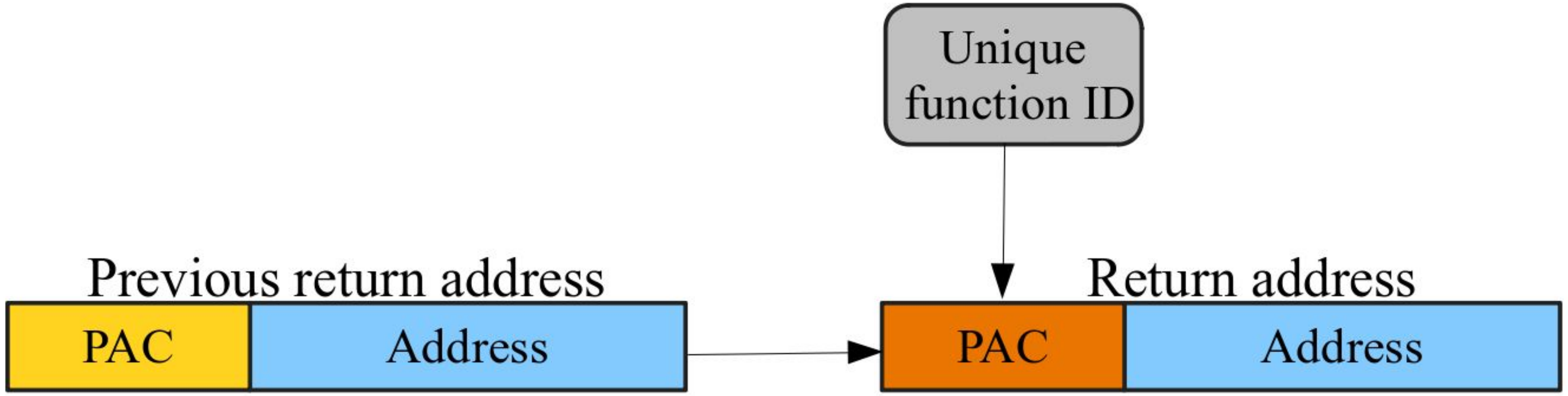
- Protecting return addresses is critical because they are, after all, the root of ROP attacks. At the same time, the return address protection scheme should impose minimal performance overhead.

- One interesting fact is that a return address can not be a dangling pointer. Hence the non-dangling property is not necessary to be enforced so the random tag is also not necessary.

- Not using the random tag has huge performance benefits because the metadata store lookup to get the random tag can be removed.

PACTight Defense Mechanisms: PACTight-RET

- Instead of blending the location of a return address in a stack to provide the non-copyability property, we use the signed return address of a previous stack frame.
- We blend a caller's unique function ID and the signed return address from the previous stack frame to generate the modifier.



Outline

PACTight Implementation

LLVM Compiler pass

Custom compiler pass for static analysis and instrumentation (3237 LoC)

AArch64 Backend

In order to efficiently enforce PACTight-RET (121 LoC)

PACTight API library

Library for PACTight primitives (656 LoC)

+Additional support

Link Time Optimization (LTO)

Outline

Evaluation: Evaluation methodology

- We evaluate PACTight by answering the following 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 environment:
  - We ran all evaluations on Apple’s M1 processor, which is the only commercially available processor supporting ARMv8.4 architecture with ARM PA instructions.
  - Specifically, we used an Apple Mac Mini M1 equipped with 8GB DRAM, 4 big cores, and 4 small cores.
- We ported our PACTight prototype to Apple’s LLVM 10 fork. For all applications, we enabled O2 and LTO optimizations for fair comparison.

Evaluation: Security Evaluation

- We evaluated PACTIGHT with three real-world exploits to test its effectiveness against real vulnerabilities
- **CVE 2015-8668:** This is a heap-based buffer overflow in the image file format library libtiff. PACTight-CFI/CPI successfully detects this and stops it from completing by enforcing pct_auth on the corrupted function pointer.
- **CVE-2019-7317:** This is a use-after-free exploit in libpng library, which implements an interface for reading and writing PNG format files. Since PACTight-CPI does recursive identification, image is considered sensitive by PACTight-CPI.
- **CVE-2014-1912:** This is a buffer overflow vulnerability in python2.7 that happens due to a missing buffer size check. PACTight-CFI/CPI detects this by detecting the corrupted function pointer with pct_auth.

Evaluation: Security Evaluation

- We evaluated PACTight with five synthesized attacks for C++ to demonstrate how PACTight-VTable can defend against virtual function pointer hijacking attacks, COOP attacks - a Turing complete attack that crafts fake C++ objects.

- We used CFIXX [1] C++ test suite by Burow et al. It contains four virtual function pointer hijacking exploits (FakeVT-sig, VTxchg-hier, FakeVT, VTxchg) and one COOP exploit.

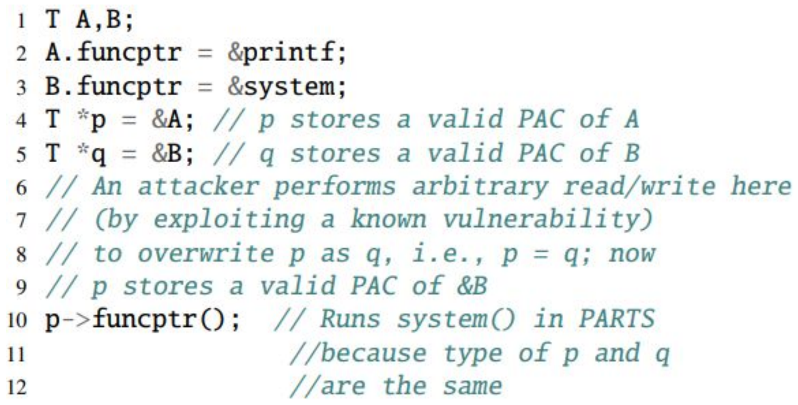
- PACTight-VTable detects all the exploits by enforcing pct_auth on the virtual function pointer before the virtual function call to detect if it has been corrupted.

[1] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2018

Evaluation: Security Evaluation

- We describe here a synthesized exploit that bypasses PARTS and PTAuth, relying on the security guarantees provided by the non-copyability property.

- PARTS [1] is vulnerable to this attack while PACTight is not. This is possible in PARTS since both pointers have the same LLVM ElementType. Our incorporation of a pointer location (&p) into the modifier with the non-copyability property blocks this attack.



```
T A,B;
A.funcptr = &printf;
B.funcptr = &system;
T *p = &A; // p stores a valid PAC of A
T *q = &B; // q stores a valid PAC of B
// An attacker performs arbitrary read/write here
// (by exploiting a known vulnerability)
// to overwrite p as q, i.e., p = q; now
// p stores a valid PAC of &B
p->funcptr();   // Runs system() in PARTS
                //because type of p and q
                //are the same
```

[1] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In Proceedings of the 28th USENIX Security Symposium (Security)

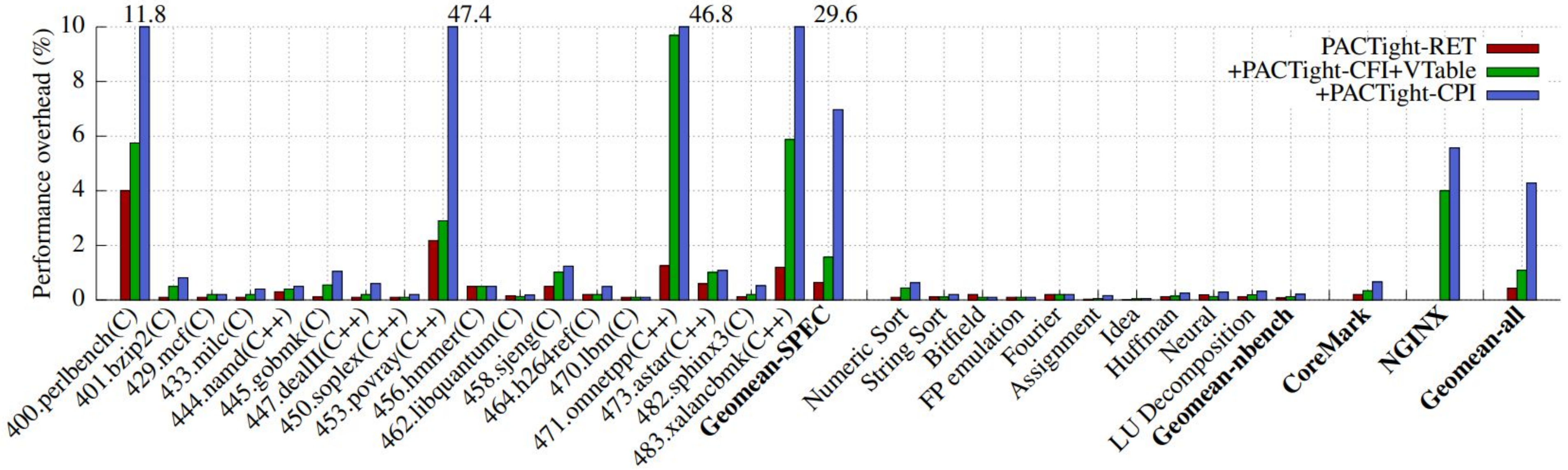
Evaluation: Performance Evaluation

- For performance evaluation, we use three benchmarks – namely SPEC CPU2006, nbench, and CoreMark – which have been used in prior works, and one real-world application, NGINX web server.

- In order to run SPEC CPU2006, we ported the SPEC benchmark to the Apple M1 and built it from scratch.

- We were not able to run one benchmark, 403.gcc, in the SPEC benchmark on the Apple M1 even with Apple's vanilla Clang/LLVM compiler so we omitted the results of 403.gcc. We suspect that there is a bug in the MacOS/M1 toolchain.

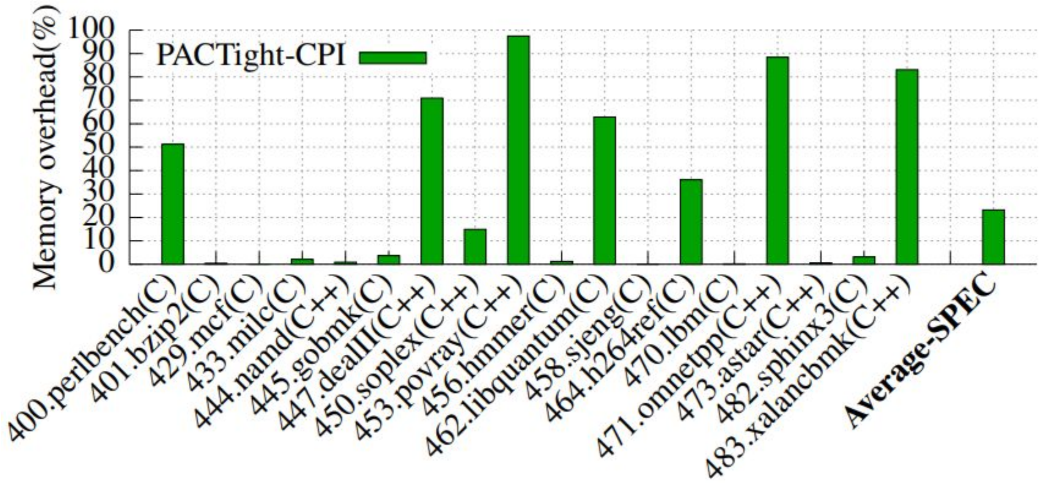
Evaluation: Performance Evaluation



- The SPEC benchmarks have a geometric mean of 0.64%, 1.57%, and 6.97% for PACTIGHT-RET, PACTIGHTCFI+VTable+RET, and PACTIGHT-CPI, respectively. The geometric means of all benchmark applications are 0.43%, 1.09%, and 4.28% for PACTIGHT-RET, PACTIGHT-CFI+VTable+RET, and PACTIGHT-CPI, respectively.
- As can be seen, PACTIGHT has very low overhead on almost all benchmarks and across all the protection mechanisms.

Evaluation: Performance Evaluation

- In order to see how much additional memory is used for PACTIGHT metadata store, we measured the maximum resident set size (RSS) during the execution of the SPEC CPU2006 benchmarks.
- We ran the SPEC benchmarks with the PACTIGHT-CPI protection because it is the highest level of protection in PACTIGHT so it requires the largest number of entries in the metadata store.
- In spite of measuring the highest security mechanism with the most instrumentations, PACTIGHT imposes an overhead of 23% on average.



Outline

Discussion and limitations

- Information leakage attack on the metadata store:
  - In our threat model, assuming a powerful attacker with arbitrary read and write capabilities, an attacker is able to access the PACTight metadata store while it is probabilistically hidden using address space layout randomization (ASLR).
  - However, even if the PACTight metadata is leaked, an attacker is not able to exploit the leaked information.
  - In order for an attacker to take advantage of the leakage, she has to launch an attack from a different location and this is already protected by the non-copyability property.

Discussion and limitations

- PACTight-CPI over-approximation
  - We over-approximate when detecting security-sensitive pointers in our static analysis. That is, we regard a pointer as a security-sensitive pointer if we cannot determine a pointer as non-security-sensitive at static time.
  - This conservative approach may induce false positives, however, such false positives will not compromise PACTight's security guarantees
- Alternative Return Address Protection.
  - We believe that extending PACTight's return address mechanism to use ARM Memory Tagging Extention (MTE) or an approach similar to Intel CET – hardware-based secure shadow stack – would result in a more secure mechanism than the current one.

Outline

Conclusion

- PACTight is an efficient and robust mechanism to guarantee pointer integrity utilizing ARM's Pointer Authentication mechanism.
- We identified 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.
- We demonstrated the security of PACTight against real-world and synthesized attacks and showcased its low performance and memory overhead, 4.28% and 23.2% respectively, against a variety of benchmarks, including SPEC 2006, nbench and CoreMark with real PAC instructions.

Publications so far

- **Tightly Seal Your Sensitive Pointers with PACTight**
  Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, Changwoo Min
  Under submission at the 31st USENIX Security Symposium [Major Revision]

- **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
  - Implemented and added support for ARM’s hardware security feature Pointer Authentication in ESXi-ARM. Support includes detecting hardware support, initializing the keys with a pseudorandom generator, and configuring the registers. Support was added in the vmkernel.
  - Contributed a fix for ESXi-ARM on QEMU to QEMU open source.
  - Participated in various tasks with the team and contributed to the ESXi source code.

  Dates Employed: May 2021 – Aug 2021

- I will be rejoining VMWare again (return offer) this next summer (2022), where I will be working with the same team again in Palo Alto, CA.

Thank You !
Questions?

Extra slides

PACTight Design: Achieving our goal

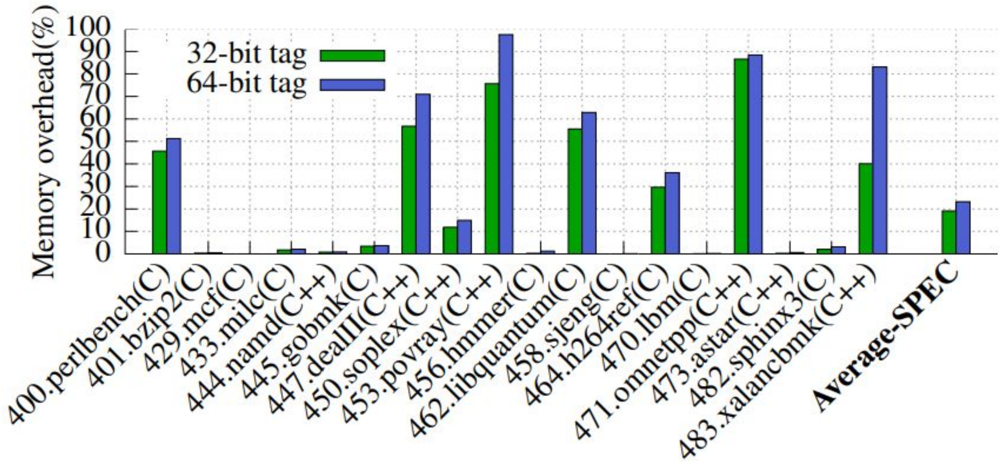
- Goal:
  - The overarching goal of PACTight is completely preventing control-flow hijacking and use-after-free attacks in a program with low performance overhead and be compatible with legacy (C/C++) programs.
- Properties:
  - In order to achieve this goal, we decided to enforce protection of the three properties on sensitive pointers
  - Sensitive pointers are all code pointers and all data pointers that are reachable to any code pointer
  - Guaranteeing the integrity of all sensitive pointers is sufficient to make control-flow hijacking impossible.
- Approach:
  - Our approach relies on leveraging Pointer Authentication to enforce the three properties.
  - PACTight relies on a unique modifier to enforce all three properties.
  - Instrumentation is done by static analysis of the LLVM IR.
  - Pointers are signed and authenticated with the PACTight runtime library

Evaluation: Performance Evaluation - 32-bit tag

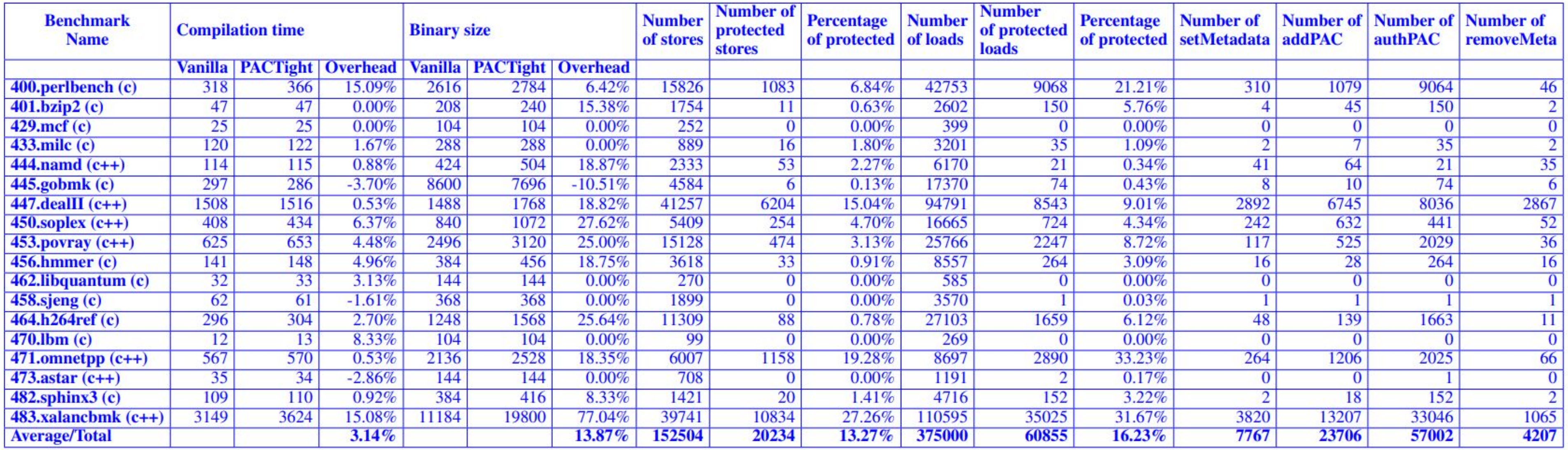
- PACTIGHT imposes an overhead of 23% on average for 64-bit tags and 19% on average for 32-bit tags.

- This makes sense due to the metadata being 16 bytes in the case of a 64-bit tag, and 12 bytes in the case of a 32-bit tag.

- The difference in memory overhead between 32-bits and 64-bit tags is around 4%, but use of 64-bit tags provides a much higher level of entropy.



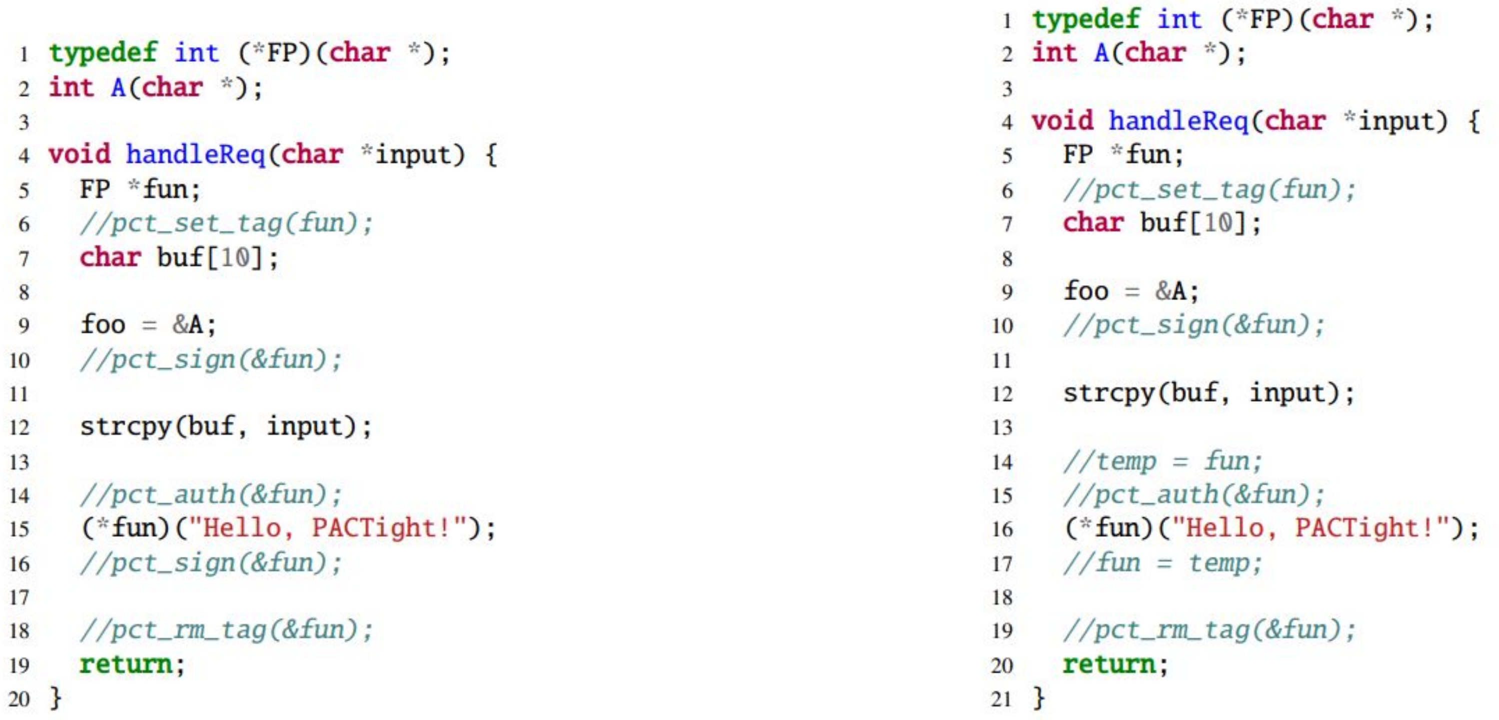
Instrumentation statistics



Benchmark Name	Compilation time			Binary size			Number of stores	Number of protected stores	Percentage of protected	Number of loads	Number of protected loads	Percentage of protected	Number of setMetadata	Number of addPAC	Number of authPAC	Number of removeMeta
	Vanilla	PACTight	Overhead	Vanilla	PACTight	Overhead										
400.perlbench (c)	318	366	15.09%	2616	2784	6.42%	15826	1083	6.84%	42753	9068	21.21%	310	1079	9064	46
401.bzip2 (c)	47	47	0.00%	208	240	15.38%	1754	11	0.63%	2602	150	5.76%	4	45	150	2
429.mcf (c)	25	25	0.00%	104	104	0.00%	252	0	0.00%	399	0	0.00%	0	0	0	0
433.milc (c)	120	122	1.67%	288	288	0.00%	889	16	1.80%	3201	35	1.09%	2	7	35	2
444.namd (c++)	114	115	0.88%	424	504	18.87%	2333	53	2.27%	6170	21	0.34%	41	64	21	35
445.gobmk (c)	297	286	-3.70%	8600	7696	-10.51%	4584	6	0.13%	17370	74	0.43%	8	10	74	6
447.dealII (c++)	1508	1516	0.53%	1488	1768	18.82%	41257	6204	15.04%	94791	8543	9.01%	2892	6745	8036	2867
450.soplex (c++)	408	434	6.37%	840	1072	27.62%	5409	254	4.70%	16665	724	4.34%	242	632	441	52
453.povray (c++)	625	653	4.48%	2496	3120	25.00%	15128	474	3.13%	25766	2247	8.72%	117	525	2029	36
456.hmmer (c)	141	148	4.96%	384	456	18.75%	3618	33	0.91%	8557	264	3.09%	16	28	264	16
462.libquantum (c)	32	33	3.13%	144	144	0.00%	270	0	0.00%	585	0	0.00%	0	0	0	0
458.sjeng (c)	62	61	-1.61%	368	368	0.00%	1899	0	0.00%	3570	1	0.03%	1	1	1	1
464.h264ref (c)	296	304	2.70%	1248	1568	25.64%	11309	88	0.78%	27103	1659	6.12%	48	139	1663	11
470.lbm (c)	12	13	8.33%	104	104	0.00%	99	0	0.00%	269	0	0.00%	0	0	0	0
471.omnetpp (c++)	567	570	0.53%	2136	2528	18.35%	6007	1158	19.28%	8697	2890	33.23%	264	1206	2025	66
473.astar (c++)	35	34	-2.86%	144	144	0.00%	708	0	0.00%	1191	2	0.17%	0	0	1	0
482.sphinx3 (c)	109	110	0.92%	384	416	8.33%	1421	20	1.41%	4716	152	3.22%	2	18	152	2
483.xalancbmk (c++)	3149	3624	15.08%	11184	19800	77.04%	39741	10834	27.26%	110595	35025	31.67%	3820	13207	33046	1065
Average/Total			**3.14%**			**13.87%**	**152504**	**20234**	**13.27%**	**375000**	**60855**	**16.23%**	**7767**	**23706**	**57002**	**4207**

Table 3: Instrumentation statistics for PACTIGHT-CPI in SPECCPU2006

PACTight optimization



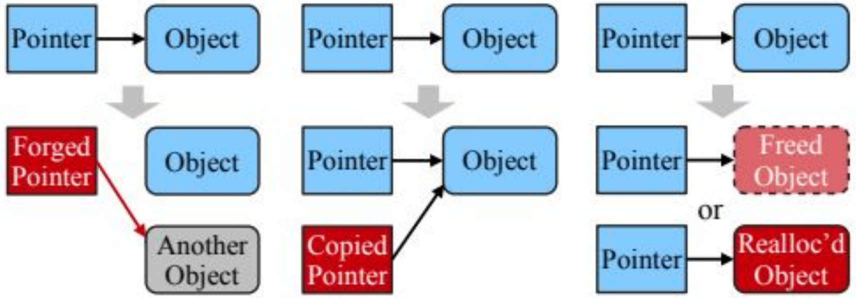
```c
typedef int (*FP)(char *);
int A(char *);

void handleReq(char *input) {
  FP *fun;
  //pct_set_tag(fun);
  char buf[10];

  foo = &A;
  //pct_sign(&fun);

  strcpy(buf, input);

  //pct_auth(&fun);
  (*fun)("Hello, PACTight!");
  //pct_sign(&fun);

  //pct_rm_tag(&fun);
  return;
}
```

```c
typedef int (*FP)(char *);
int A(char *);

void handleReq(char *input) {
  FP *fun;
  //pct_set_tag(fun);
  char buf[10];

  foo = &A;
  //pct_sign(&fun);

  strcpy(buf, input);

  //temp = fun;
  //pct_auth(&fun);
  (*fun)("Hello, PACTight!");
  //fun = temp;

  //pct_rm_tag(&fun);
  return;
}
```

Figure 7: Two examples showcasing the instrumentation effect of the optimization The example on the left shows the instrumentation without the optimization, and the example on the right shows the instrumentation with the optimization. As can be seen, the optimization removes the `pct_sign` after the dereference, thus reducing the number of PAC instructions executed.

PACTight Design: Pointer integrity

- Based on the limitations of prior PAC approaches and our observation on how a pointer can be compromised, we define three security properties of pointer integrity.
  - **Unforgeability:** A pointer can be forged (i.e., corrupted) to point to an unintended memory object.
  - **Non-copyability:** A pointer can be copied and re-used maliciously. If non-copyability is guaranteed, the security impact is non-replayability, and thus pointer attacks that replay PAC-ed pointers for malicious use would be prevented.
  - **Non-dangling:** A pointer can refer an unintended memory object if its pointee object is freed or the freed memory is reallocated



(a) Forgeability (b) Copiability (c) Dangling pointer