Pythia: Compiler-Guided Defense Against Non-Control Data Attacks

Sharjeel Khan Bodhi Chatterjee Santosh Pande

ASPLOS

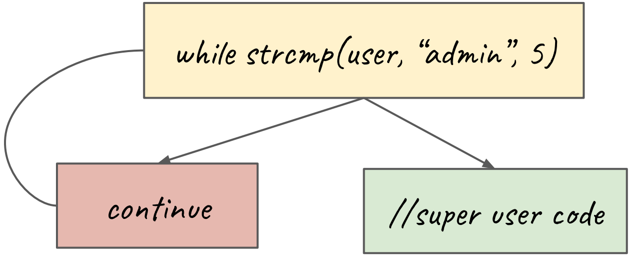
Georgia Tech

Software Memory Vulnerabilities = Non-Control Data Attacks

Software Memory Vulnerabilities

Privilege Escalation
Information Leakage
Control-Flow Manipulation
Unwanted Code Imports
Arbitrary Code Execution

Implications of Data Attacks

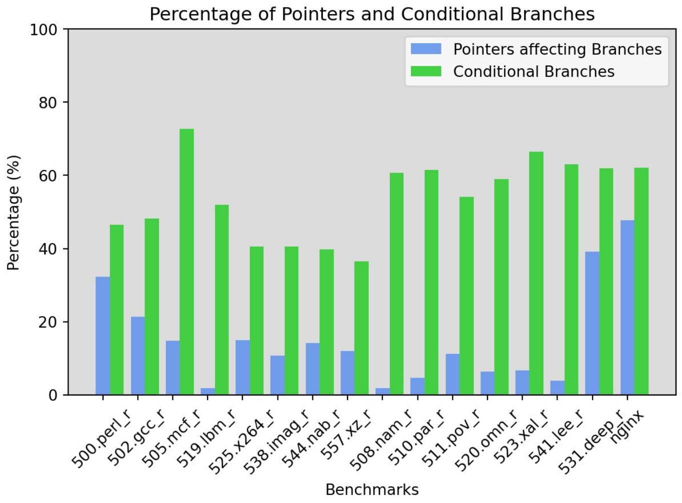


Flipping Branch Predicate by String Overflow

- ⏩ Applications written in *memory unsafe languages* (C/C++) are vulnerable to data-attacks
- ⏩ **Control-flow Bending** involves diverting a program branch's target into alternative paths
- ⏩ Challenging to defend against because of *prevalence of program branches* & *program pointers*

Conditional Branches & Program Pointers = Scalability & Analyzability

- Prevalence of conditional program branches impacts the *scalability* of this problem
  - Over **55%** of terminator IR instructions in SPEC 2017 are conditionals (~700,000)
  - Over **56%** of Nginx are conditionals (~16,000)



- Presence of program pointers affecting branches leads to *un-analyzability*
  - Out of 900,000 pointer instructions in SPEC 2017, more than 200,000 affect conditionals branches

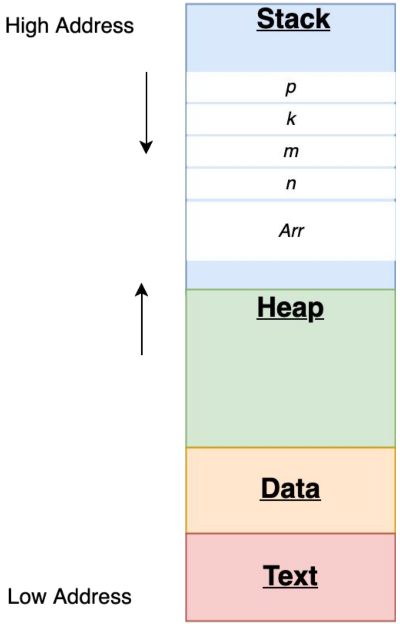
Georgia Tech

Pointer Positioning Attack: Data Pointer Corruption

```c
int *p ;
int k, m, n ;
int Arr [100] ;
p = Arr ;
scanf ( " % d " , & k ) ;
...
m = n - 1;
*p = n + 1;
if ( m > n ) {
   // privileged execution
}
```

p stores the base address of *Arr*

p points to *m* by overwriting pointer value and this alias sets *m = n +1*



Pointer **p** can be corrupted by **k** due to the overflow (memory vulnerability)

Real-World Control-Flow Bending: Privilege Escalation

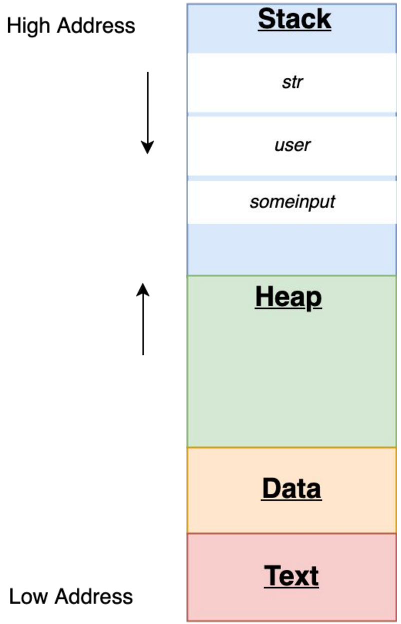
```
void Access ( char pwd [20]) {
   char str [ SIZE ] , user [ SIZE ];
   char someinput  [ DIFF_SIZE ];
   ...
   verify_user ( user , pwd ) ;
   if ( strncmp ( user , " admin " ,5) ) {
      // super user code
      ...
   } else {
      // normal user code
      ...
   }

   strcpy ( str , someinput ) ;

   if ( strncmp ( user , " admin " ,5) ) {
      // super user code
      ...
   } else {
      // normal user code
      ...
   }
}
```

Input string declaration

First string comparison after verifying username with password

Copying str into *someinput* could lead to an overflow

Second string comparison could be manipulated to be True, via buffer-overflow causing super user privilege



Existing State-of-the-Arts: High Overheads, Generalizability & Compatibility Issues

Defenses Against Non-Control Data Attacks

Data-Flow Integrity (DFI)

✅ Handles Control-Flow Bending by checking reachability constraints

❌ High Overheads (40%-100%) resulting from excessive runtime checking

❌ Incompatibility with C++ codebases

❌ Inability to handle program pointer positioning attacks

Address Randomization

✅ Performs address space re-layout to prevent control-flow bending

❌ High Runtime Overheads (20%-55%) arising from performing randomization at runtime

❌ Specific to program stack/heap only

Memory Safety

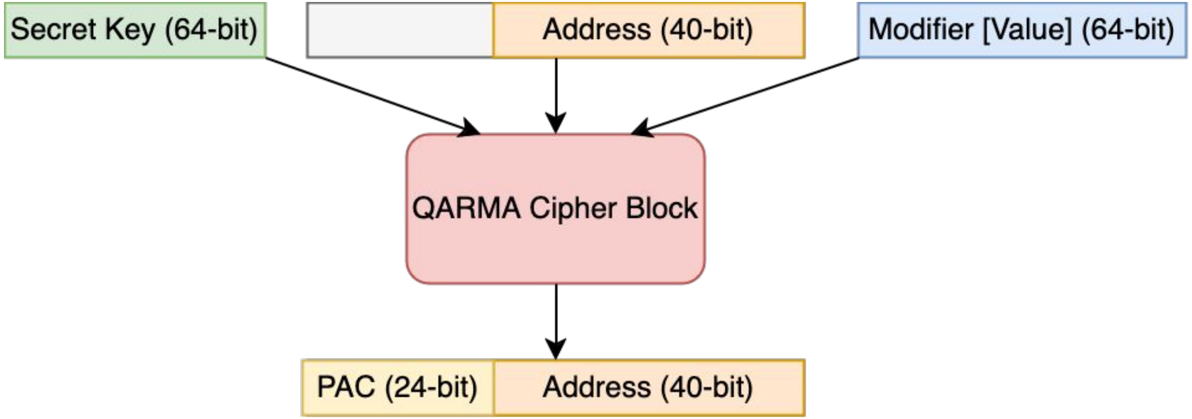
✅ Introduces *non-trivial language extensions* to prevent spatial & temporal memory violations

❌ Significant runtime overheads due to runtime pointer checks

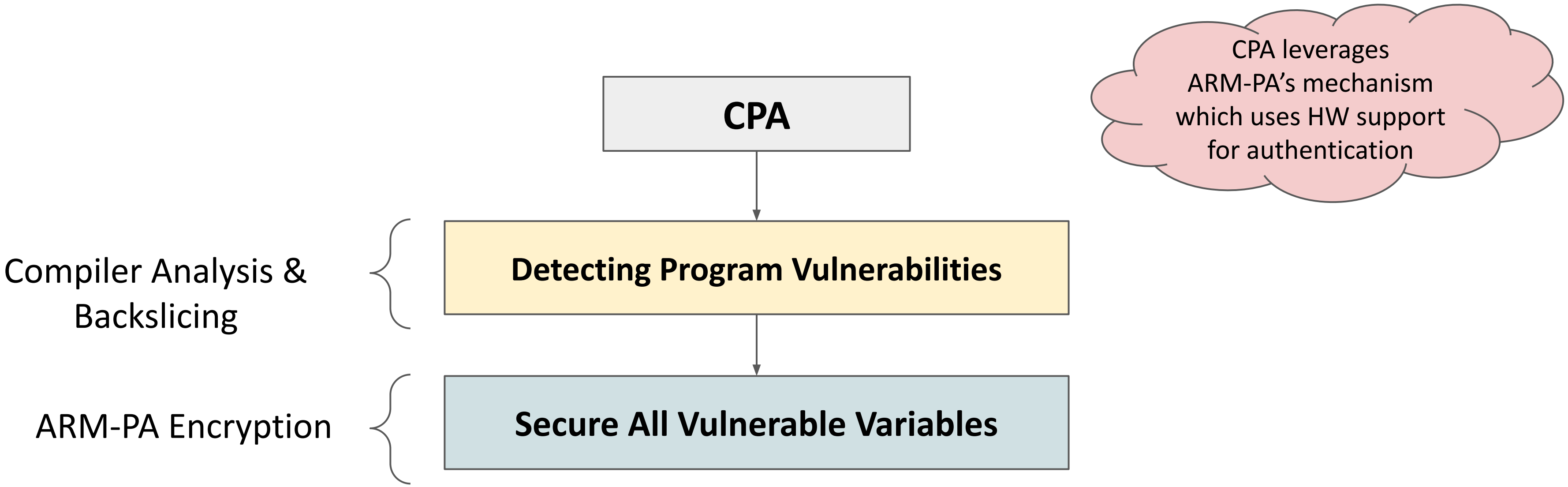
❌ Issues with C++ codebases

ARM Pointer Authentication (PA)

- *Specialized hardware mechanism* that ensures the integrity of data and code pointers associated with the program
- Not all bits are required to define the address space - top bits are used as **Pointer Authentication Code** (PAC)
- In 64-bit architectures, the pointer address space is less than 40-bits so it leaves PAC with 24 bits (1 in 16 million chance)
- Each encryption (PACD*) and authentication (AUTD*) instruction takes around ~2ns



Complete Pointer Authentication (CPA) for Branches

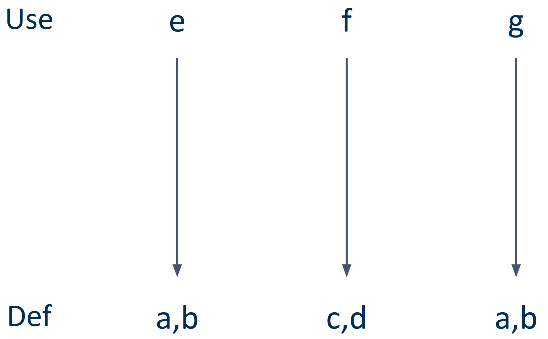


This is a *baseline* scheme where all possible program vulnerabilities are secured with ARM-PA to prevent overflows

Detecting Program Vulnerabilities: Variable "Backslicing"

- For a given branch, traverse the variables' *Use-Def* chains to detect all variables influencing that branch
- Program pointers and their possible aliases are also secured
- To secure variables:
  - **Authenticate** (decrypt) pointers variables before **loads**
  - **Sign** (encrypt) pointers in the set before **stores**

```
int a = initVar(10),
    b = initVar(20);
    c = initVar(30);
    d = initVar(40);
do  {
    /*loop body*/
    e += a * b;
    f += c / d;
    if (e == f)
        break;
    g = a + b;
} while (g);
```



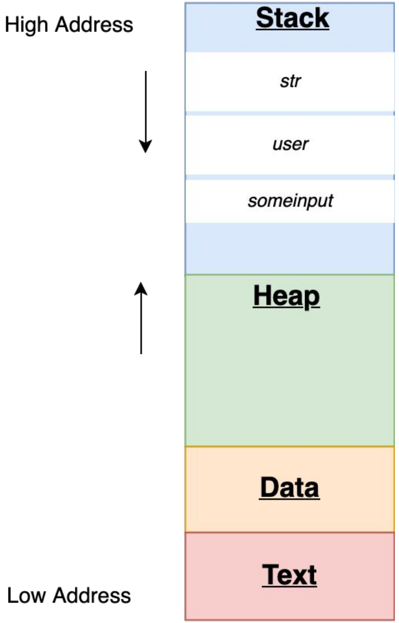
Example: Real-World Control-Flow Bending with CPA

```
void Access ( char pwd [20]) {
   char str [ SIZE ] , user [ SIZE ];
   char someinput  [ DIFF_SIZE ];
   llvm.pythia.sign(user);

   verify_user ( user , pwd ) ;
   llvm.pythia.auth(user);
   if ( strncmp ( user , " admin " ,5) ) {
      // super user code
      ...
   } else {
      // normal user code
      ...
   }

   strcpy ( str , someinput ) ;
   llvm.pythia.auth(user);
   if ( strncmp ( user , " admin " ,5) ) {
      // super user code
      ...
   } else {
      // normal user code
      ...
   }
}
```

Encrypt pointer to user

Authenticate pointer to user
Branch Backslice Variables: user

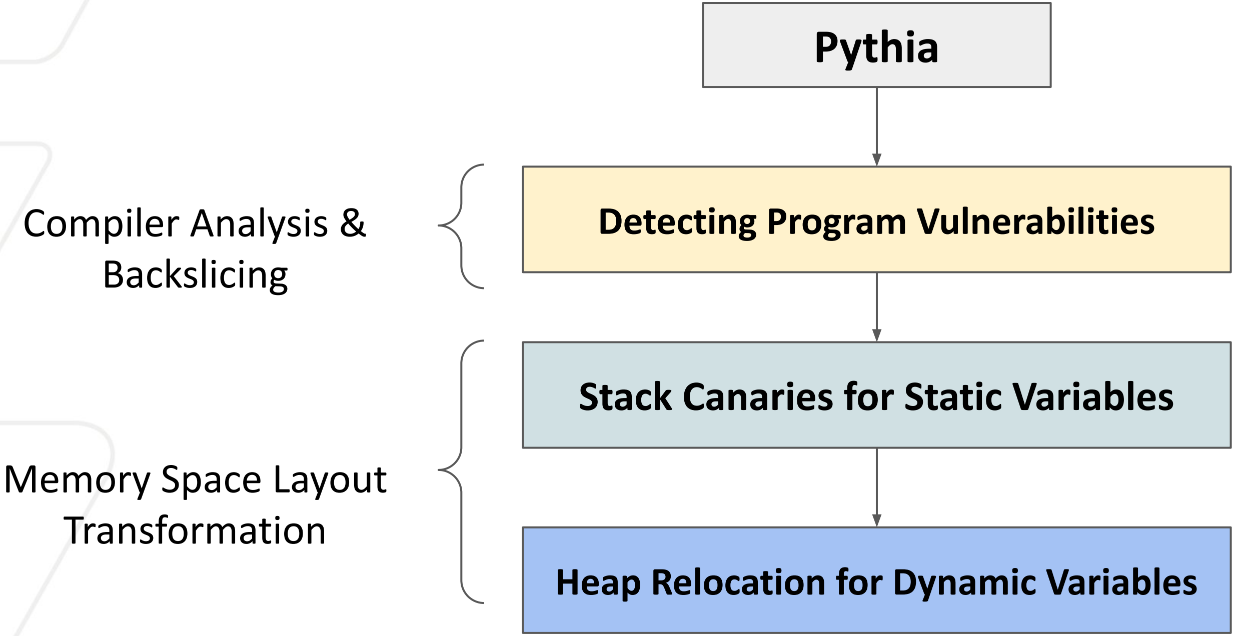
Authenticate pointer to user
Branch Backslice Variables: user



*Our experiments have shown that following this naive scheme incurs upto **48%** runtime overheads, and bloats the binary size by **22%***

Georgia Tech

Pythia: Performance-Aware Defense Approach for Control-Flow Bending

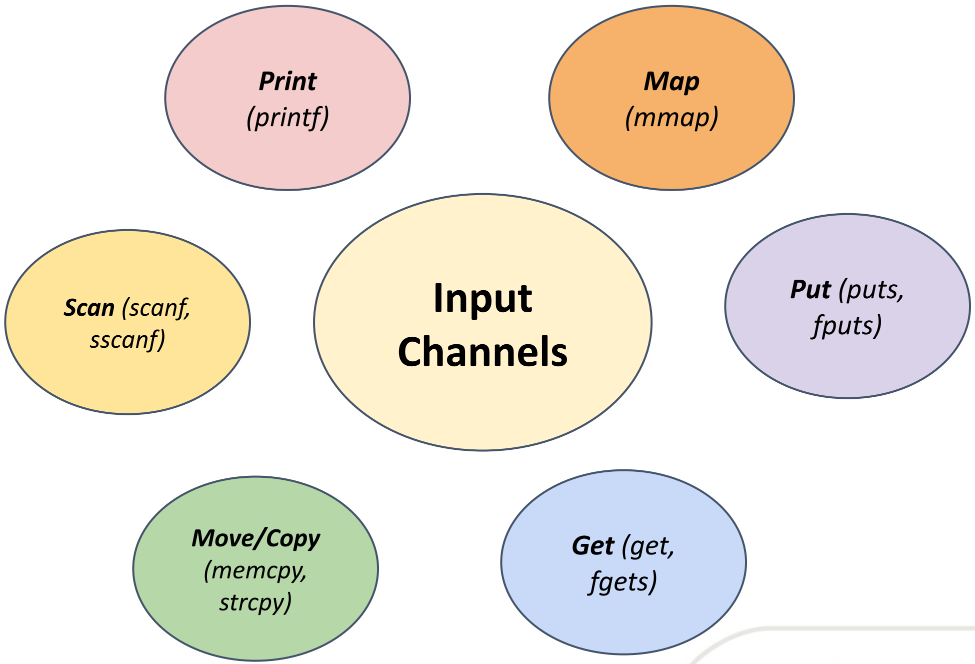


Input Channels: Source for Attackers to Trigger Overflows

- Most overflow-based attacks are triggered via program's **input-channels**

- Attackers manipulate the arguments of input-channel functions to trigger buffer overflows

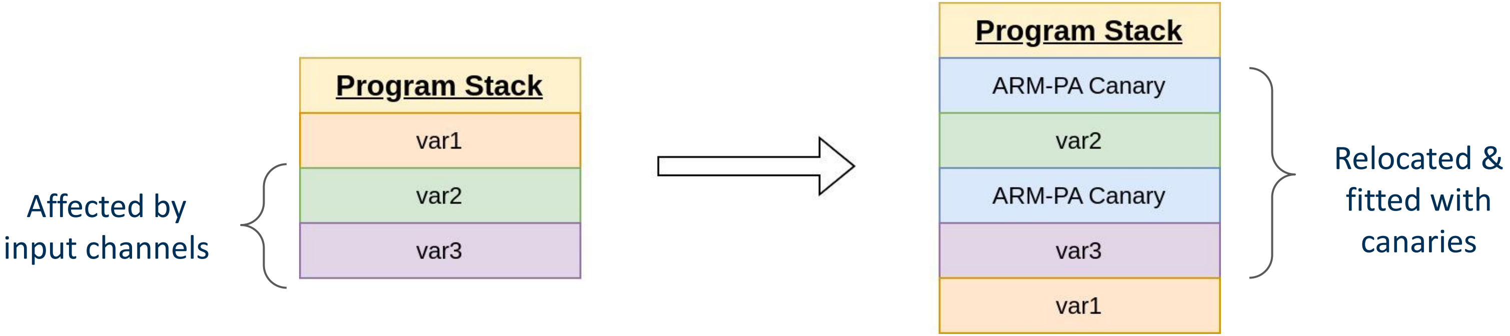
- Backslicing input-channel function variables (rather than branch variables), refines the set of vulnerable variables



Georgia Tech

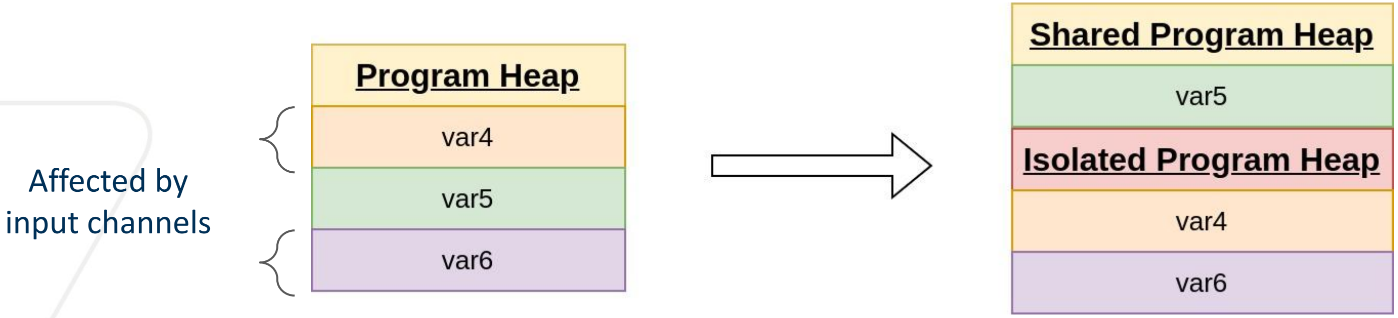
Memory Layout Transformation: Stack Canaries

- ❑ Stack variables used in ICs are moved to the top of function frame
- ❑ Canaries are added in between each vulnerable stack variables
- ❑ Before IC for a stack variable, the canary gets re-encrypted in case of a long call-chains
- ❑ After IC for stack variable, the canary gets authenticated to check for overflows



Memory Layout Transformation: Heap Relocation

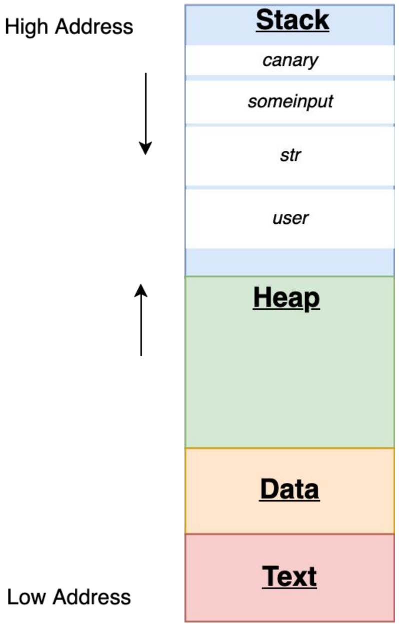
- Vulnerable heap variable are relocated to an *isolated part of the heap* memory
- This is achieved via Pythia's custom memory allocation (e.g. malloc -> smalloc)
- Pythia's custom memory allocation reserves a big chunk of heap in the beginning and allocates based on heap allocation function calls
- These vulnerable heap variables and its uses are encrypted/authenticated with ARM PA on every store/load respectively



Real-World Example: Control-Flow Bending with Pythia

```
void Access ( char pwd [20]) {
    int canary = lib_random();
    char someinput  [ DIFF_SIZE ];
    char str [ SIZE ] , user [ SIZE ];
    llvm.pythia.sign(canary);

    verify_user ( user , pwd ) ;
    if ( strncmp ( user , " admin " ,5) ) {
        // super user code
        ...
    } else {
        // normal user code
        ...
    }
    llvm.pythia.auth(canary);
    canary = lib_random();
    llvm.pythia.sign(canary);
    strcpy ( str , someinput ) ;
    llvm.pythia.auth(canary);
    ...
}
```

- Canary got created
- someinput got moved to top
- Encrypt canary value
- Authenticate canary then re-randomize it
- Encrypt canary value
- Input Channel with someinput as target
- Authenticate canary so there was no overflow



Georgia Tech

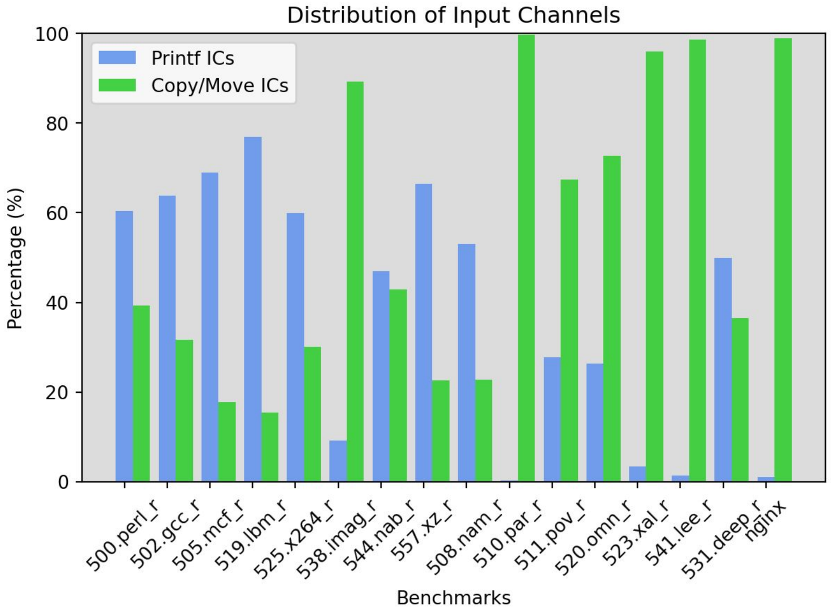
Evaluation

- How effective is the conservative scheme in defending against non-control data attacks, and what are its runtime overheads?
- How secure is Pythia’s performance-aware approach involving stack canaries and heap sectioning approach against non-control data attacks?
- Does it manage to reduce the runtime overheads and ARM-PA instructions compared to the conservative scheme?
- How does Pythia compare to DFI in terms of securing vulnerable branches in applications that can be manipulated through the input channels?

Summary of Results

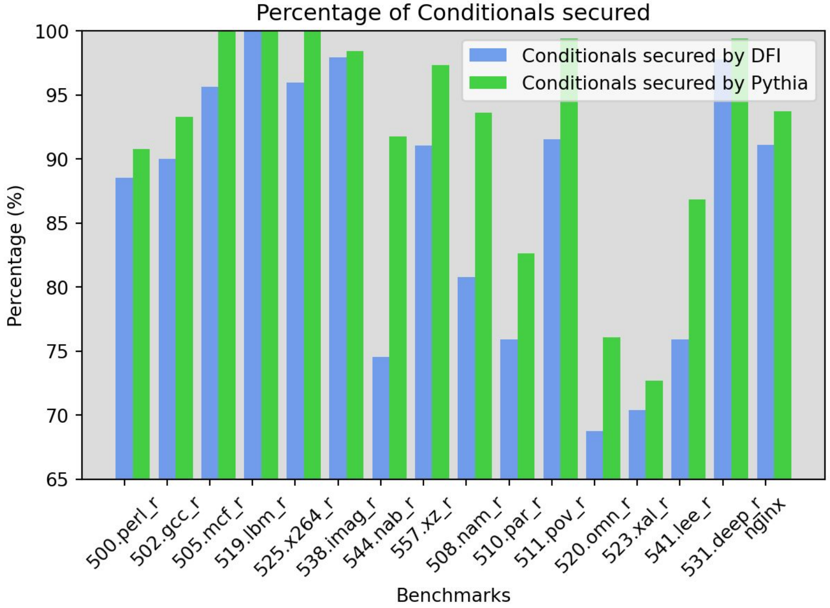
- Performance Results:
  - Average Overhead: **47.88%** (CPA) vs **13.07%** (Pythia)
  - Average Binary Increase: **21.56%** (CPA) vs **10.37%** (Pythia)
- Security Results:
  - 25326 ICs with 31.5% prints and 65.9% move/copys
  - Pythia decreased PAs by 4.25x
  - Branch protection: 92.2% (Pythia) vs 86.6% (DFI)
  - 100% Branch Protection: 3 (Pythia) vs 1 (DFI)
  - Works on well-known real-life attacks such as ProFTPd

Security Results (Input Channels)



~25000 ICs with 31.5% prints and 65.9% move/copys

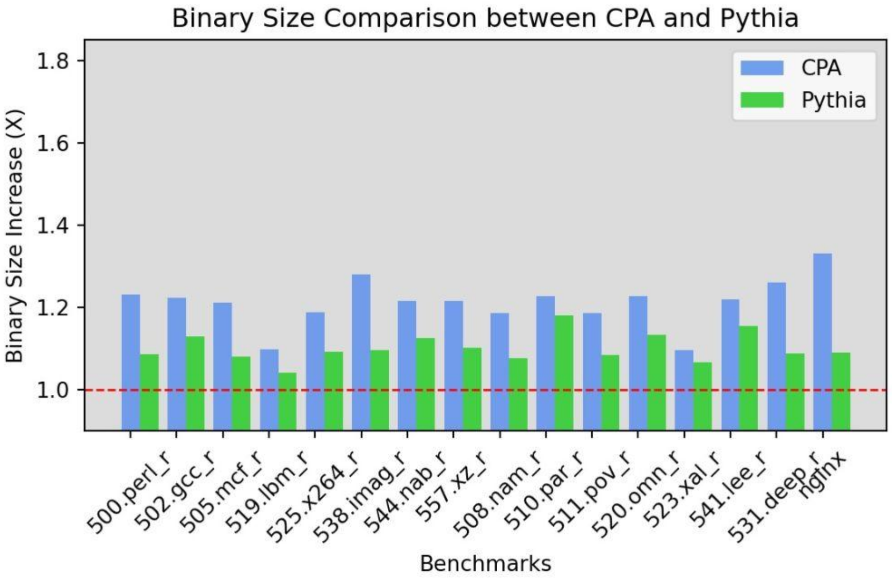
Security Results (Branch Protection)



Branch protection: 92.2% (Pythia) vs 86.6% (DFI)

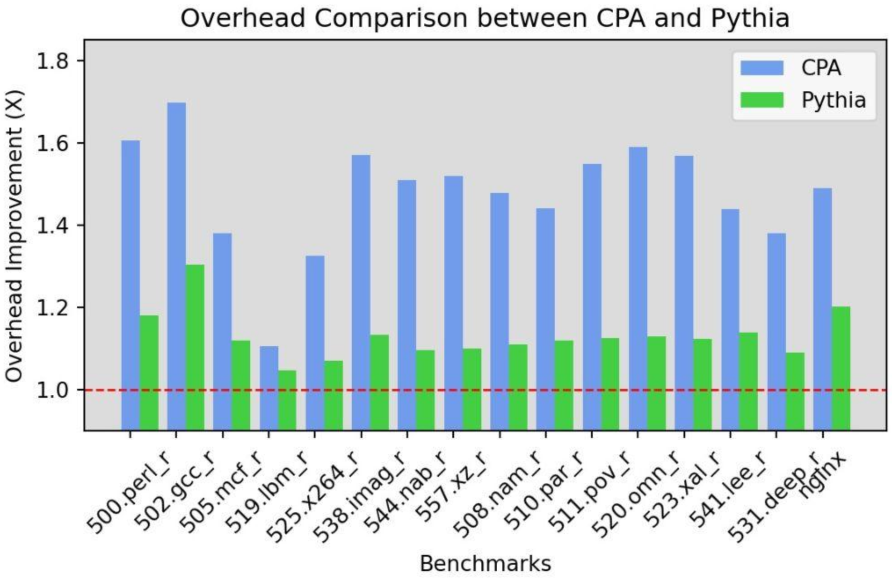
100% Branch Protection: 3 (Pythia) vs 1 (DFI)

Performance Results (Binary Size)



Average Binary Increase: **21.56%** (CPA) vs **10.37%** (Pythia)

Performance Results (Overheads)



Average Overhead: **47.88%** (CPA) vs **13.07%** (Pythia)
Pythia decreased PAs by 4.25x

Georgia Tech

Conclusion

- Pythia:
  - Compiler-guided defense framework with pointer authentication to tackle Non-Control Data Attacks
  - Utilizes hardware components already available in ARM chips
  - Effective on *pointer-intensive applications* and *C++ codes*
  - Works on well-known real-life attacks such as **ProFTPd**
- Pythia’s average overhead is **13.07%** compared to Complete Pointer Authentication’s average overhead of **47.88%**
- Pythia can secure **5.6% (~33000)** more branches than DFI in SPEC 2017 and Nginx and fully secure 3 applications

Georgia Tech

Thank You

- Questions?