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

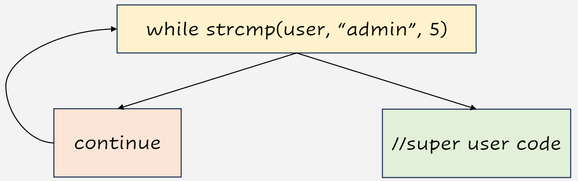
Sharjeel Khan, Bodhisatwa Chatterjee, Santosh Pande
School of Computer Science, Georgia Institute of Technology

Non-Control Data Attacks

- Applications written in memory unsafe languages such as C++ suffer from various memory vulnerabilities
- These vulnerabilities make the application susceptible to data-attacks, which leads to following scenarios:
  - Privilege Escalation
  - Information Leakage
  - Control-Flow Manipulation
  - Unwanted Code Imports
- **Non-Control Data Attacks** occur when an adversary corrupts program data that does not directly manipulate program control such as *function calls* and *return addresses*

Control-Flow Bending

- **Control-Flow Bending** is a popular instance of Non-Control Data attacks, where an attacker can change the control-flow of program in such a way that it follows a valid path in the program control-flow graph
- Such attacks typically involve flipping the branch outcomes to divert the branch target to a code region of interest



- Such attacks are harder to defend against, since traditional techniques are unable to distinguish the "correct" program execution path
- Frequent *program branches* and presence of *program pointers* complicate the problem in terms of overhead and ability to analyze and defend

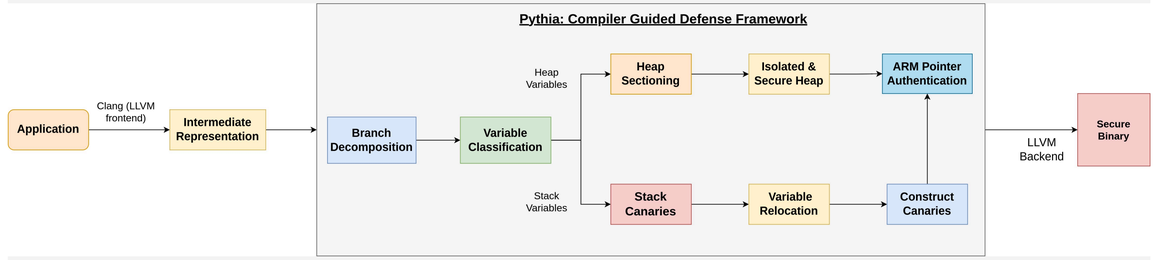
Exploiting Pointer Misdirection & Dualism

- *The first contribution of this work is to present a new class of non-control data attacks that are based on data pointer manipulation and exploiting pointer arithmetic*
- Programmers write optimized code that exploits the dualism between program pointers and array pointers

- These attacks arise from two possible program vulnerabilities:
  - *Input channel* variables gaining access to program pointer
  - Variables participating in branch predicates can be tainted to flip the branch outcomes

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

Pythia: Compiler Guided Defense Mechanism



Pythia: Compiler-Guided Defense Mechanism Against Control Flow Bending Attacks

Branch Decomposition & Input Channel Construction

- The set of branch sub-variables of a branch predicate statement represents every possible program variable that can affect the outcome of the given branch
- For the computation of the branch sub-variable set, we leverage the *backward program slices* of branch predicate variables
- The set of variables that involve input channel, can be computed using their respective program forward-slices

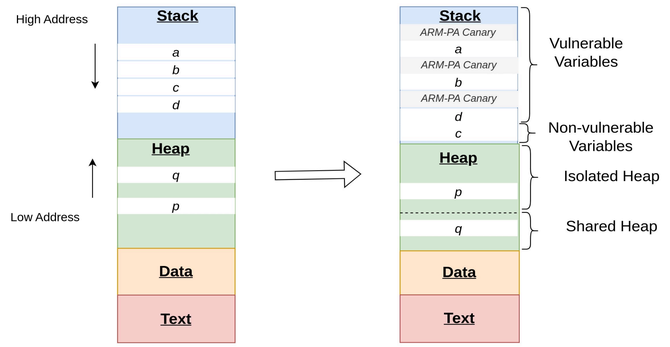
Stack Canaries & Heap Sectioning

- Pythia refines the set of vulnerable variables by first segregating *statically and dynamically allocated program variables*

Source Program

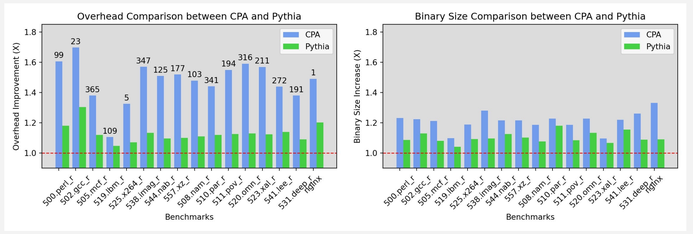
```
void foo() {
 int a, b, c, d, *p, *q;
 a = initVar(10);
 p = new int(10);
 scanf("%d", &b);
 q = new int(20);
 d = a * b;
 if (p < d)
    p = &a;
 else
    q = &a;
}
```

Memory Space Layout Transformation

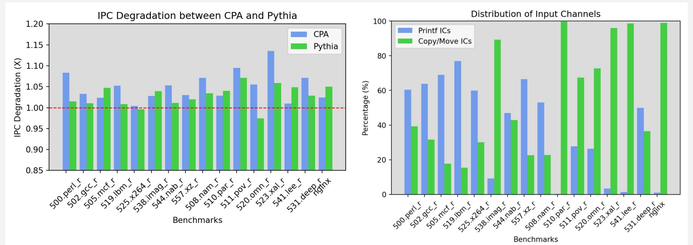


- Pythia re-arranges the stack memory layout to allocate the vulnerable variable to the stack bottom (higher address).
- In the event of any overflow triggered by an adversary, the stack memory space of non-vulnerable variables will not be affected.
- Pythia splits the program heap into an *isolated section* and *shared section*. The vulnerable program variables are allocated to the secure portion of the heap, while others are on the shared portion.
- Pythia accomplishes heap sectioning by creating two variations of the memory allocation algorithm: one for isolated allocation and shared allocation.

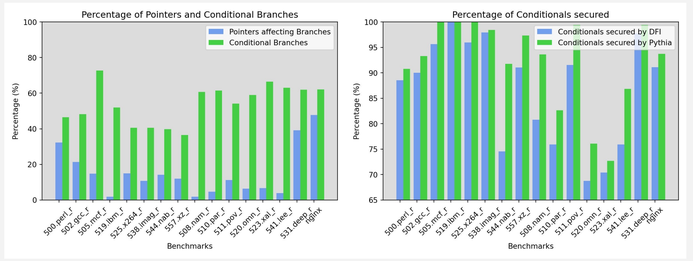
Results



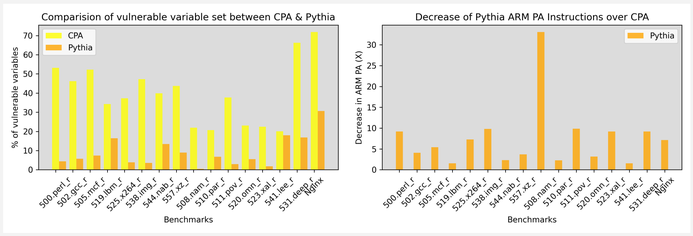
Runtime overhead and Binary Size Comparison



IPC Degradation and Input Channel Distribution



Distribution of Data Pointers and Additional Conditional Branches Secured



Vulnerable Variables and ARM-PA instructions comparison

Conclusion

- *Pythia* is a compiler-guided defense framework that combines traditional compiler analysis with pointer authentication.
- *Pythia*'s performance-aware approach has an average overhead of **13.07%** without compromising security guarantees.
- In addition, *Pythia* can secure **5.6%** branches more than DFI and fully secure 3 applications.

Georgia Tech College of Computing
School of Computer Science
ASPLOS