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

Sharjeel Khan
smkhan@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Bodhisatwa Chatterjee
bodhi@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Santosh Pande
santosh.pande@cc.gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

Abstract

Modern C/C++ applications are susceptible to *Non-Control Data Attacks*, where an adversary attempts to exploit memory corruption vulnerabilities for security breaches such as privilege escalation, control-flow manipulation, etc. One such popular class of non-control data attacks is **Control-flow Bending**, where the attacker manipulates the *program* data to flip branch outcomes, and divert the program control flow into alternative paths to gain privileges. Unfortunately, despite tremendous advancements in software security, state-of-art defense mechanisms such as *Control-flow Integrity* (CFI), are ineffective against control-flow bending attacks especially those involving flipping of branch predicates.

In this work, we propose a *performance-aware defensive scheme*, **Pythia**, which utilizes ARM's pointer authentication (PA) hardware support for *dynamic integrity checking of forward slices of vulnerable program variables* that can be affected by input channels, and *backward slices of branch variables* (including pointers) that may be misused by the attacker. We first show that a *naive scheme of protecting all vulnerabilities can suffer from an average runtime overhead of* **47.88**%. We then demonstrate how overheads can be minimized by *selectively adding ARM PA-based canaries* for statically-allocated program variables and creating *secure sections* for dynamically-allocated variables to avoid overflows by input channels. Our experiments show that employing this hybrid approach results in an average overhead to **13.07**%. We empirically show that Pythia offers better security than state-of-the-art data flow integrity (DFI) technique, especially in pointer-intensive code and C++ applications with 92% branches secured on an average and 100 % secured in case of 3 applications.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Software and application security**.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651343

Keywords: Control-Flow Bending, ARM-PA, Program Slices, Stack Canaries

ACM Reference Format:
Sharjeel Khan, Bodhisatwa Chatterjee, and Santosh Pande. 2024. Pythia: Compiler-Guided Defense Against Non-Control Data Attacks. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620666.3651343

1 Introduction

Real-world applications are vulnerable to various *data attacks*, where an adversary with a malicious intent attempts to exploit software memory corruption vulnerabilities. This includes targeting instances of stack/string buffer overflow [19, 41, 47, 78], integer overflow [9, 75, 76], heap corruption [29, 59, 64] , use-after-free [45, 81], etc with the objective of either gaining privileged access (*privilege escalation*), executing unwanted code segments (*control flow manipulation*), read/write values from memory (*information leakage*) or other ways to hinder intended process execution. These vulnerabilities are predominantly featured in the *Common Weakness Enumeration (CVE)* 2023 [1] list.

The attacks resulting from memory vulnerabilities can be broadly classified into two groups - **Control-Data Attacks** and **Non-Control Data Attacks**. Instances of control-data attacks typically involve an adversary corrupting a *program code-pointer* (function pointers, return statements, etc), to 'hijack' the control-flow of a program and divert it to a designated target. To defend against such control-flow hijacking attacks, a common methodology is to monitor the targets of indirect control-transfer instructions and restrict them to a set of feasible targets. This forms the basis of *Control-Flow Integrity* (CFI) mechanism and over the past two decades, CFI and its variants have evolved significantly over time and have been a focus of extensive research in security literature [2, 11, 12, 20–23, 25, 33, 35, 37–39, 52, 58, 61, 73, 77, 80, 82, 84]. However, recent works have highlighted that the current CFI mechanisms are often inadequate and can even create additional security vulnerabilities [13, 18, 49].

On the other hand, **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. A popular instance of non-control data attacks is *Control-flow Bending* [13], 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 (CFG). Such attacks typically involve flipping the branch outcomes to divert the branch target to a code region of interest (such as privileged or sensitive code). Such attacks are credible threats [13, 16, 32, 34], and are harder to defend against, since CFI techniques are unable to distinguish the "correct" program execution path if there are multiple feasible execution paths in the CFG. In other words, analyzing static artifacts such as return addresses of functions or branch targets are not adequate to detect such attacks, which calls for dynamic monitoring of the program data-flow. Such monitoring incurs significant runtime overheads especially since program branches are very frequent (typically every 10^{th} instruction is a branch). Prior works have proposed defense mechanisms for non-control data attacks either by isolating sensitive parts of the program [27, 43, 67, 70, 85], introducing language extensions to enforce constraints [36, 54–56, 66], or by monitoring 'critical' program variables [83].

Tainting branch variables through memory corruptions and overflow leading to privilege escalation are the traditional ways of control-flow bending attacks. In this work, we show that the problem and the range of attacks in this category could be broader.

Pointer Misdirection & Exploiting Pointer Dualism (§3): *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.* We show that an adversary can successfully carry out *control-flow bending* attacks by either tainting variable(s) that contribute to the computation of conditional branch predicates, or by manipulating data pointer to point towards branch predicate variables. Such attacks cannot be satisfactorily defended by employing static program analysis techniques such as *Data-flow Integrity* (DFI) [14] because of the inability to deal with pointer arithmetic and lack of field sensitive analysis.

To establish an effective defensive mechanism against data attacks stemming from input channels, entire *program execution path* from input channel to the branch must be analyzed and protected. This also allows early detection of these attacks and paves the way for adopting necessary mitigating mechanisms. However, detecting the dynamic execution path of an application is an extremely challenging problem, because of frequent branch instructions and the presence of pointers in the program. This also makes the program path input-dependent, and it can change dynamically across various execution instances. Performing pointer-based path analysis at individual branch-level granularity can be quite challenging limiting the extent of protection offered by the technique. In this work, we show that relying on the protection of individual variables encountered along these paths leveraging crypto-based integrity checks in hardware is a more viable solution than developing data-flow integrity checking mechanisms in software solving both problems of precision and overheads.

Over the last few years, CPU vendors have started to add new hardware extensions to enable cryptographic defense mechanisms for pointers. *ARM pointer authentication* (PA) [5] is one such mechanism, that allows verification of a pointer's integrity based on its address bits. One naive solution to utilize the ARM-PA mechanism for preventing control-flow bending attacks, would be to simply convert all variables associated with conditional branches into pointers, and then sign and authenticate them. However, our experiments show that this leads to substantial overheads (47.88%), as every variable requires to be encrypted when stored from memory, and then authenticated before subsequent loads.

On the other hand, another diametrically opposite way to tackle to most data attacks is simply to prevent an attacker from mitigating any program variables that can lead to any subversion of control-flow. This means that we need to eliminate the notion of *dispatcher functions or gadgets* [13], i.e functions that can overwrite their return address in the presence of attacker-supplied arguments. Usually, such functions involve program *input channels*, through which attackers can manipulate program variables. The presence of dispatcher gadgets enables an attacker to overwrite memory locations that comprise the branch predicate variables, thereby flipping the branch outcome. Thus, to establish a secure defense mechanism against control flow bending attacks with low overheads, we propose *Pythia*, a *compiler-guided defense mechanism* that is based on a hybrid model of ARM-PA and the elimination of dispatcher functions.

Complete Defense against Non-Control Data Attacks: *The second contribution of this paper is a conservative scheme that protects against all known control-flow bending attacks.* This is achieved by first determining two categories of "vulnerable" program variables: **a)** *forward-slices* of input channels variables (input channel construction), that can be leveraged by an attacker to cause buffer overflows into branch variables, and **b)** *backward-slices* of branch predicate variables (branch decomposition) that can be tainted with the malicious intent to cause control-flow bending. Program variables from both these "vulnerable" categories are encrypted with ARM-PA across the board to ensure complete defense against control-flow bending attacks, the new class of pointer misdirection, and pointer exploitation attacks. Additionally, this scheme also performs alias analysis to handle pointer variables in both categories.

Performance-Aware Complete Defensive against Non-Control Data Attacks: *The third contribution of this paper is an end-to-end compiler framework for defending against non-control data attacks with low overheads. Pythia* improves upon the conservative defense mechanism by selectively using ARM-PA to minimize the runtime overheads. To achieve

this, *Pythia* further categories the vulnerable program variables into two sub-categories: **a)** *Statically-Allocated Variables* that reside in the program's stack memory space, and **b)** *Dynamically-Allocated Variables* that originate in the program's heap memory space. *Pythia* tackles the former by performing a re-orientation of stack variables, and then adding stack canaries with ARM-PA to detect any overflows, while the latter is handled with heap sectioning, that transfers dynamically-allocated vulnerable program variables into a "*secure*" section of the heap. This reduces the number of ARM-PA instructions by 4.25x, and brings down the runtime overhead significantly.

Pythia has been tested on C/C++ benchmarks from SPEC 2017 Benchmark Suite [10], popular Real-world examples [15], and in additional benchmarks such as Nginx [57]. We show that Pythia can detect these possible control-flow bending attacks in these workloads and achieved an average runtime overhead of only 13.07%, compared to 47.88% from CPA. Compared to the conservative scheme, we also show that *Pythia* reduces the number of ARM-PA instructions by a factor of total number of branches present in the program.

2 Background and Motivation

2.1 Control-Flow Bending

Control-Flow Bending [13] is a generalization of non-control data attack, where the attacker manipulates the program data (non-code pointer) which results in diverting the control flow into alternative paths in the program CFG. CFI is completely ineffective against this as the "bended" target is always in the set of feasible targets in the program CFG. Although these attacks have been known to exist for a while, this problem has remained untackled mostly and state-of-the-art approaches such as (DFI) [14] leaves important practical cases of pointer intensive code and C++ application bases untackled prompting this work.

2.2 Motivating Examples: Non-Control Data Attacks

String-Buffer Overflow leading to Privilege Escalation: A simple example for a non-control data attack [86] leading to privilege escalation, in shown in Listing 1. In this example, the user is verified by an input password, and the result is assigned to the variable "*user*". The *user*-variable is frequently checked to provide access to certain operations. However, in between the checks, the function interacts with the user to get some other inputs. Input pointer "*someinput*" can be manipulated by the user, causing a buffer overflow vulnerability in line 13, and leading to superuser access. This attack is not handled by any CFI mechanism, because technically both the targets in lines 15 and 18 are feasible. This constitutes a classic instance of control-flow bending where CFI is unable to distinguish the 'possible' static target from the 'actual' dynamic target, where line 15 should only be a target when the user has privileged access.

```
1  void Access(char pwd[20]) {
2   char str[SIZE], user[SIZE];
3   char *someinput;
4
5   verify_user(user, pwd);
6   if(strncmp(user,"admin",5)){
7     // super user code
8     ...
9     }else{
10     // normal user code
11     ...
12    }
13  strcpy(str,someinput);
14  if(strncmp(user,"admin",5)){
15     // super user code
16     ...
17  }else{
18   // normal user code
19   ...
20   }
21 }
```

Listing 1. Simple example [86] of a Non-Control Data Attack exploiting string buffer-overflow leading to Privilege Escalation

ProFTPd Attack leading to Information Leakage: The ProFTPd attack [34] is a Data-Oriented Programming (DOP) based attack to eventually leak the private key by breaking ASLR through the *sreplace* function in ProFTPd shown in Code 2. The vulnerability in the function is because of a faulty check at Line 24. The attacker first triggers this overflow check by constructing inputs through CWD (change directory) semantics. When '*cp*' points to the last character of the buffer *buf*, that is $(cp - buf + 1)$ equals *blen*, the check returns $false$ and Line 27 overwrites string terminator inside the buffer. During subsequent iterations of the while loop, at Line 14, $strlen(pbuf) > blen$ and invoking *sstrncpy* overflows the buffer into the stack overwriting local variables such as '*rarr*' and '*cp*'. Since both the source and destination in the string copy function *sstrncpy* at Line 14 are corrupted, the attacker controls the number of bytes copied in the successive iterations of the while loop.

It can be seen that the root cause of the above attacks is the ability to flow (taint) values into branch predicates or position the respective pointers involved by manipulating them. Data flow integrity techniques such as DFI fail in the presence of pointers when it comes to field insensitive analysis and do not deal with pointer arithmetic. This leads us to propose defensive mechanisms that first identify these vulnerabilities and defend against these attacks by selectively leveraging ARM-PA.

```
1  char* sreplace(char*s, ...) {
2    ...
3    char *m, *r, *src = s, *cp;
4    char **mptr, **rptr;
5    char *marr[33], *rarr[33];
```

```
6   char buf[BUF_MAX] = {'\0'}, *pbuf = NULL;
7   size_t mlen=0, rlen=0, blen; cp=buf;
8   ...
9   while(*src){
10  for(mptr=marr,rptr=rarr;*mptr; mptr++,rptr++) {
11      mlen = strlen(*mptr);
12      rlen = strlen(*rptr);
13      if(strncmp(src,*mptr,mlen) == 0){
14      sstrncpy(cp,*rptr,blen-strlen(pbuf));
15      if(((cp + rlen) - pbuf + 1) > blen){
16        cp = pbuf + blen - 1;
17        } /*Overflow Check*/
18        ...
19        src += mlen;
20        break;
21         }
22    }
23    if(!*mptr) {
24      if((cp - pbuf + 1) > blen){ // off-by-one
      error
25          cp = pbuf + blen - 1;
26      } /*Overflow Check*/
27      *cp++ =*src++;
28     }
29   }
30 }
```

Listing 2. Example of ProFTPd Vulnerability leading to Information Leakage

2.3 Background: ARM Pointer Authentication

The ARM Pointer Authentication (ARM-PA) [5] is a specialized hardware mechanism that ensures the integrity of data and code pointers associated with the program. It was first introduced in the ARMv8-A architecture. The intuition here is that modern architectures allocate a larger number of bits for defining the address space of pointers. However, not all bits are required to define the address space. For instance, in 64-bit architectures, the address space of pointers does not require more than 40-bits. Thus, these unused bits are utilized to assign a *Pointer Authentication Code* (PAC). Based on the PAC bits, it's possible to determine if a pointer's integrity has been compromised or not. Recently, this mechanism has gained popularity as memory safety mechanisms for both control and non-control data attacks [31, 48, 51, 65].

2.4 Overview of Pythia

We now describe *Pythia* (Fig. 1) a compiler-guided defensive framework that utilizes a performance-aware approach to combat the problem of non-control data attacks, namely control-flow bending. *Pythia* improves on a conservative defensive mechanism that first analyzes all conditional branch statements, and input channels present in the application. The conditional branch variables are decomposed, and input channel variables are mapped, into their constituent variables by taking their program *back-slices* and *forward-slices* respectively. This allows us to capture the entire set of "vulnerable" program variables that can be tainted and result in control-flow bending. For pointer-type variables, an alias analysis is performed to determine all possible variables that can be pointed-to by a specific pointer that has a potential alias with these variables. The conservative scheme simply encrypts all such variables with ARM-PA instruction, resulting in complete defense against control-flow bending attacks, but with substantial runtime overheads.

On the other hand, *Pythia* follows a performance-aware approach to reduce the runtime overheads incurred by the conservative scheme. After the branch and input channel decomposition, *Pythia* then further classifies the vulnerable variables into *statically-allocated* (stack) variables and *dynamically-allocated* (heap) variables. The rationale behind such an approach is that statically allocated variables in stack memory have a fixed address associated with them, and their integrity can be checked by adding a canary to them, which acts as an indicator for potential overflows. However, this approach does not work satisfactorily for dynamically-allocated program variables that reside in the heap memory, because of limited control over the memory allocation at the user level. To tackle such variables, *Pythia* sections the heap memory into an isolated section where the vulnerable program variables reside, and into a shared section where other variables are allocated. To achieve this, *Pythia* uses a custom implementation of *malloc* that is combined with ARM-PA checks. This allows *Pythia* to selectively use ARM-PA, instead of applying it across the board and allows it to minimize the runtime overheads significantly.

2.5 Threat Model & Attacker Goals

In this paper, we assume that the attacker can corrupt any program variable, at any point in time, with unlimited attempts. This essentially means that a control-flow bending attack can occur at any point during program execution. In our threat model, we assume that either an attacker can directly corrupt the value that participates in the branch predicate or can corrupt a value that comprises a backward slice of a branch ie, a value that participates in the computation of branch predicate value through an input channel. A third mechanism is also available to the attacker to position a pointer to point to the branch variable so that the attacker can load a malicious value into the branch variable using the alias of the pointer (ie, by leveraging the l-value of pointer dereference encountered before the branch). Thus, the objective of Pythia is to detect such attacks as early as possible, since corrupt program variables can lead to erroneous program states as a cascading effect.

The goal of the attacker is to divert the application's control-flow into alternate execution paths, either to obtain privileged access, leak information or hinder program execution in any manner. For the purposes of this paper,

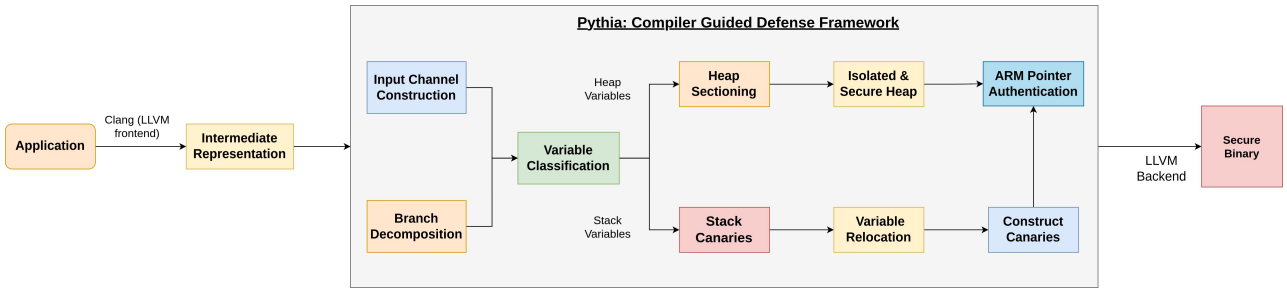

Figure 1. Pythia: Compiler-Guided Defense Framework against Control-Flow Bending Attacks. The vulnerable program variables are segregated based on whether they are statically allocated (stack) or dynamically allocated (heap) in the program memory layout, and Pythia leverages ARM PA for constructing canaries for stack-allocated variables, and for isolating the heap.

we consider any diversion in control-flow as a 'successful attack'.

2.6 Basic Terminologies

Definition 2.1 (Input Channel (IC))**.** The input channel is any function that is vulnerable to memory corruption. Attackers manipulate these functions to modify the variables of the program's memory.

In this paper, we consider six different categories of *input channel* functions: **print**, **scan** (Reads strings specifically in a format - *scanf*), **move/copy**, **get** (e.g. *fgets*), **put** (e.g. *strcpy, memmove, memcpy*) and **map** (maps files or devices to the virtual address space - *mmap*)

Definition 2.2 (Def-Use (DU) Chains)**.** *Use-Def Chains is a widely used data-flow graph that links a variable definition with its corresponding definition.*

Definition 2.3 (Upwards-Exposed Use)**.** *A program variable v has an upwards-exposed use at a program point p, if there exists only one path from its definition to p.*

To quantify the early detection capabilities of a security scheme, we define **attack distance**:

Definition 2.4 (Attack Distance)**.** *Attack distance represents the number of static program instructions between the beginning of a backward slice where the protection starts and the branch predicate.*

The attack distance shows how high the protection must start to secure the input channel in terms of the number of instructions. Intuitively, if a technique's attack distance is not greater than or equal to the attacker's attack distance (which is the input channel), it will not be able to protect the branch. In such cases, an attacker can taint the branch predicate's backward slice through the unprotected values in the program without being detected by the protection scheme, leading to successful attacks. Due to these reasons, an input channel based attack can be detected only if the defensive mechanism has a large enough attack distance that is higher than the input channel.

3 Pointer-Based Control-Flow Bending Attacks

In this section, we describe a class of attack that involve manipulation of program pointers and use of pointer arithmetic to achieve control flow bending.

3.1 Exploiting Pointers and Array Dualism

Often, programmers write optimized code that exploits the dualism between program pointers and array pointers. Consider the code snipped presented in Listing 3. In this case, an input channel variable k (line 3) is used to increment the base address of array *Arr* (line 4) through l. In the normal, non-malicious execution of the code (when p is not aliased to m), the privileged code is bypassed. In this code snippet, however, an attacker can input a malicious value of k overflowing into l, which can set the pointer p point to m. In such an event, the attacker can make a pointer p point to m, setting a new value of m to $n + 1$, bending the predicate $m > n$, and gaining privileged access.

In summary, these attacks arise from two possible vulnerabilities: **1)** Input channel variables gaining access to program pointers, that can potentially point across the entire range of variables, **2)** Variables participating in branch predicates can be tainted to flip the branch outcomes.

```
1  int *p, Arr[100], l, k;
2  int m, n;
3  p = Arr; //p stores the base address of Arr
4  scanf("%d", &k);
5  p = p + l; // l represents the stride for an
       element
6  ...
7  m = n-1;
8  *p = n+1; // p aliased to m by setting right value
       of k and this alias sets  m = n+1
9  if (m > n) {
10  //privilaged execution
11 }
```

Listing 3. Simple example of an adversary exploiting the dualism between array pointers and program pointers

4 Pythia: Design and Implementation

This section describes the *Pythia Compiler Framework*, which incorporates lightweight defense mechanism against control-flow bending attacks. We first describe a conservative defensive mechanism that can thwart all known instances of control-flow bending attacks (§4.1 - 4.2), which serves as a baseline. We then illustrate the performance-aware defensive approach taken by *Pythia* (§4.3), which minimizes ARM-PA checks over the conservative baseline. An upper-bound probability estimate for brute force attacks targeting stack canaries, and a qualitative analysis of their security strengths is discussed in §4.4.

The core insight that is leveraged by Pythia is that control-flow bending attacks can be triggered by only a finite subset of program variables. These variables are either *branch predicate variables*, *variables with input channels*, or *program pointers*. In this paper, we call the collective set of these variables as **vulnerable variables**. The goal here is to isolate such variables so that their integrity can be authenticated with ARM-PA.

4.1 Branch Decomposition & Input Channel Construction

The program path taken by an application during its execution can vary in the presence of branch statements. Specifically, the dynamic program control flow is contingent upon the individual program variables that constitute the branch predicate. However, the program control flow can also be subverted by manipulating other program variables that entail a direct/in-direct definition of branch variables. Therefore, we need to consider all such possible program variables that can be exploited to flip the outcome of a conditional branch and subvert the program's control-flow.

A naive approach to solve this problem would be to simply authenticate and secure all possible program variables. However, adopting such an approach will entail frequent authentication across every variable and their uses, and will incur significant overheads. Thus, to minimize runtime overheads, we first need to determine the set of *vulnerable* program variables, which can act as a source of control-flow bending attacks. We start by defining the notion of *branch sub-variables*:

Definition 4.1 (Branch Sub-Variable). *A program variable v, is a branch sub-variable for a conditional branch b, if it's either a branch predicate variable of b, or if it contains an upwards-exposed use of at least one of branch predicate variables of b.*

In a nutshell, *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 backward program slice of a variable is obtained by traversing its *Use-Def (UD) Chain* (2.2) against the direction of control flow. The intuition here is that each branch variable essentially is an upwards-exposed use of other program variable(s) above the branch predicate in the program [53]. This backwards traversal is performed transitively from the branch prediction to the start of its function. This process is illustrated with a simple example in Fig 2.

Algorithm 1: Branch Decomposition Algorithm

```
Input: Conditional Branch Instruction BrInst
Result: Branch sub-variable set B_sub(BrInst)
worklist, B_sub ← ϕ
parent_funct ← getParentFunction(BrInst)
Bvars ← BrInst.getVars()
for each variable bvar ∈ Bvars do
    def_set ← getallDefinitions(bvar)
    for each definition def ∈ def_set do
        worklist.push_back(def)
    end
end
while worklist is not empty do
    d ← remove a definition from the worklist
    for each operand op ∈ d do
        if op ∈ Bt then
            B_sub.push_back(op)
        else
            def_set ← getAllDefinitions(op)
            for each definition def ∈ def_set do
                worklist.push_back(def)
            end
        end
    end
end
```

The process of branch sub-variable computation has been summarized as *branch decomposition algorithm* (Algorithm 1). It follows a worklist-based approach that iteratively captures sub-variables emanating from branch variables by traversing their def-use chain in a backwards (against the control-flow) direction. The worklist keeps track of all definitions remaining to be decomposed at a particular time-step. The branch decomposition algorithm deals with pointer variables by loading the value stored at the address pointed, after performing null pointer checks. The alias of all the pointers present in the function and their backward slices are also analyzed by this algorithm.

On the other hand, in order to refine the set of vulnerable program variables that can lead to control-flow bending attacks, we can analyze program variables that are a part of user input channels. Similar to branch decomposition, the set of variables that involve input channel, can be computed using their forward-slices. A *forward program slice* is obtained by traversing the use-def chains of variables along the direction of dataflow. This is the exact reverse of the branch decomposition algorithm, as here we find the sub-variables by walking the use-def chain in a forward manner, analyzing any definition that uses the input channel variables. This process is illustrated in Fig 2. The *input channel*

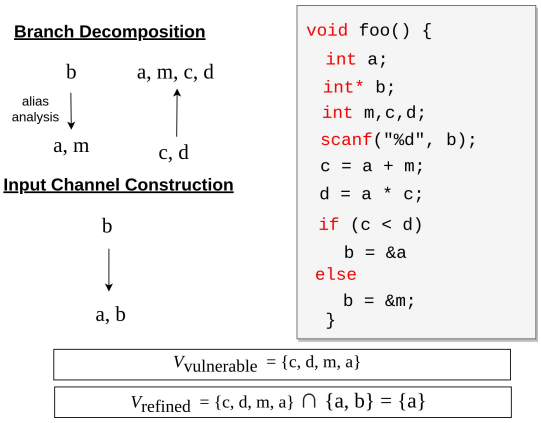

Figure 2. Simple illustration of computing vulnerable program variables by branch decomposition & input channel construction.

construction algorithm also follows the same structure as the branch decomposition worklist algorithm. *The intersection of the sub-variables sets obtained from branch decomposition and input-channel construction, constitutes the refined set of vulnerable program variables.*

Once the set of vulnerable program variables is computed and refined, the next step is to secure them to prevent control-flow bending attacks. We will now discuss various defensive mechanisms that can be used to secure these variables.

4.2 Conservative Approach: Complete Pointer Authentication using ARM-PA

In this defensive approach, all vulnerable program variables are simply encrypted using ARM-PA mechanism (without any refinement), and then decrypted at every subsequent load, to authenticate their integrity. ARM-PA leverages the unused address bits in program pointers to maintain variable integrity. Therefore, in order to successfully apply this scheme to vulnerable variables, data pointers are created for each non-pointer vulnerable variable. Each created data pointer is encrypted at its definition, and when it's stored to the memory, its integrity is checked before every use. In case an attacker has attempted to taint any variable, it will be detected before it is loaded from the memory. This scheme of encrypting every vulnerable program variable results in the inclusion of at least two ARM-PA instructions. In addition, this scheme also finds out the may-aliases of pointers and ensures that they adhere to the ARM-PA encryption and decryption scheme for accessing.

For a single vulnerable variable i with u_i number of uses, this conservative scheme would introduce *one additional instruction for encrypting* during store (each variable can be defined only once in SSA IR form) plus the u_i *number of additional decrypting instructions* for each use. This leads to $1+u_i$ number of additional program instructions, for vulnerable program variable i. Therefore, in a program with B conditional branches and v vulnerable variables, each with u uses

Algorithm 2: Complete Pointer Authentication (CPA) Algorithm

```
1  Input: Set of all conditional branches Branches present in
     program
   Result: Encrypt and Authenticate the set of vulnerable variables
           associated with Conditional Branches using ARM-PA
2  for each branch instruction br ∈ Branches do
3      vulnerable_vars ← branchDecomposition(br)
4      for each definition def ∈ vulnerable_vars do
5          ARM_PA.encrypt(def)
6          for each load L_v ∈ def do
7              ARM_PA.authenticate(L_v)
8          end
9          for each store S_v ∈ def do
10             ARM_PA.encrypt(S_v)
11         end
12     end
13 end
```

(on average), the maximum number of extra instructions that are instrumented in the program is given by:

$$\text{Avg ARM-PA instructions} = B\ v\ (2\ u + 1) \tag{1}$$

An algorithm depicting the complete pointer authentication scheme is presented in Algo. 2. The scheme takes in all vulnerable variables associated with conditional branches and input-channels present in the program. It first computes the set of vulnerable program variables by branch decomposition (line 3). It then creates a pointer reference for each one of them. In addition, it also performs alias analysis for these pointers. It then adds ARM-PA encryption on the *store* instruction, and then ARM-PA authentication (decryption) for each subsequent use.

4.3 Performance-Aware Approach: Stack Canaries & Heap Sectioning

Pythia further refines the set of vulnerable variables by first segregating statically and dynamically allocated program variables. For statically-allocated program variables that reside in the program's stack memory, *Pythia* relocates them on the bottom of the stack memory to isolate and capture the effect of buffer overflows effectively. For dynamically allocated variables, *Pythia* divides the program's heap memory into two sections (*isolated* and *shared*), and vulnerable variables are relocated to the isolated section to prevent buffer overflow attacks.

⋆ **Securing Statically Allocated Variables**: *Pythia* first detects all the branch sub-variables that are allocated in the program's stack memory, within a function. It then performs input channel construction on such variables to determine the interaction between the variable uses and input channels. A stack-allocated variable is marked as vulnerable if any of its (direct/indirect) uses is passed on as arguments to input channels. *Pythia* re-arranges the stack memory layout to allocate the vulnerable variable to the stack bottom

(lower address). In the event of any overflow triggered by an adversary, the stack memory space of non-vulnerable variables will not be affected since the stack memory usually grows only in one direction, i.e towards the higher address. Any possible try of writing at the beginning of a stack array would cause a bus error so it can only write towards the end of the array (the higher address).

Algorithm 3: Stack Re-layout & Canary Algorithm

```
Input: Set of all conditional branches Branches present in program
Result: Perform stack re-layout & encrypt vulnerable variables with canaries
vulnerable_vars_s ← ∅
for each branch instruction br ∈ Branches do
    backsliced_vars_s ← branchDecomposition(br)
    for each branch variable b_v ∈ backsliced_vars do
        if b_v ← isStaticMemoryAllocation() then
            vulnerable_vars_s ← InputChannelConstruction(b_v)
        end
    end
end
for each vulnerable stack variable s_v ∈ vulnerable_vars_s do
    stackMemoryLayout(s_v)
    can_sv ← addCanary(s_v)
    stackMemoryLayout(ca)
    ARM_PA.encrypt(ca)
    relevant_uses ← getInputChannelUses(s_v)
    for each dispatcher use du ∈ relevant_uses do
        ARM_PA.decrypt(ca)
        de_ref ← *ca
        ARM_PA.encrypt(ca)
    end
end
```

However, despite the stack layout re-orientation, overflows from vulnerable stack variables can still spill into one another and lead to control-flow bending. To solve this problem, *Pythia* adds canaries with random values between each vulnerable stack variable. *Pythia* inserts integrity checks in the stack canaries, which serve as an indicator of buffer overflow in a vulnerable stack variable. The initialization of the stack canary value is chosen at random, to prevent an adversary from reverse-engineering the mechanism by analyzing the program binary. To minimize runtime overheads, *Pythia* uses hardware ARM-PA since it directly utilizes the memory location's address bits for encryption & decryption. In case of a memory violation, the ARM-PA decryption mechanism triggers a program crash.

An algorithm depicting the stack re-layout and the canary encryption is presented in Algorithm 3. This scheme ensures that no vulnerable program variable can overwrite any other program variable, which prevents control-flow bending attacks resulting from tainting statically-allocated program variables through malicious inputs.

Similar to statically allocated program variables, control-flow bending attacks can also originate from dynamically allocated program variables. Although it's straightforward to detect vulnerable heap-allocated variables that interact with input channels, performing heap re-layout is extremely challenging since it involves performing non-trivial changes in the system's default memory allocation algorithm. The goal here is to develop a simple light-weighted scheme that doesn't involve adding canaries across the entire structure of heap memory, which will cause significant overheads, and defeat the purpose of dynamic memory allocation.

★ **Securing Dynamically Allocated Variables**: In order to protect dynamically allocated program variables, *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, and the other variables are on the shared portion of the heap. *Pythia* accomplishes heap sectioning by creating two variations of the memory allocation algorithm: one for isolated allocation and shared allocation. These algorithms handle memory allocations for specific address ranges. *Pythia* first detects the dynamically allocated vulnerable program variables that interact with input channels. It then replaces their heap memory allocation for them to be mapped in the *isolated* heap. *Pythia*'s custom memory allocation is based on *glibc*'s *malloc* implementation, and both libraries are linked at the compile time.

Algorithm 4: Heap Sectioning

```
Input: Set of all conditional branches Branches present in program
Result: Perform Heap-Sectioning & encrypt dynamic vulnerable variables
vulnerable_vars_h ← ∅
for each branch instruction br ∈ Branches do
    backsliced_vars_h ← branchDecomposition(br)
    for each branch variable b_v ∈ backsliced_vars do
        if b_v ← isDynamicallyMemoryAllocation() then
            vulnerable_vars_h ← InputChannelConstruction(b_v)
        end
    end
end
safeAddr ← performHeapSectioning()
for each vulnerable heap variable h_v ∈ vulnerable_vars_h do
    relocate(h_v, safeAddr)
    relevant_uses ← getInputChannelUses(h_v)
    for each dispatcher use du ∈ relevant_uses do
        ARM_PA.decrypt(du)
        de_ref ← *du
        ...
        *du ← de_ref
        ARM_PA.encrypt(du)
    end
end
```

After sectioning the program heap memory into isolated and shared regions, *Pythia* uses ARM-PA to encrypt the vulnerable variable and its uses. The scheme of securing dynamically allocated variables has been summarized in Algorithm 4. It follows a similar flow as the stack layout algorithm (3)

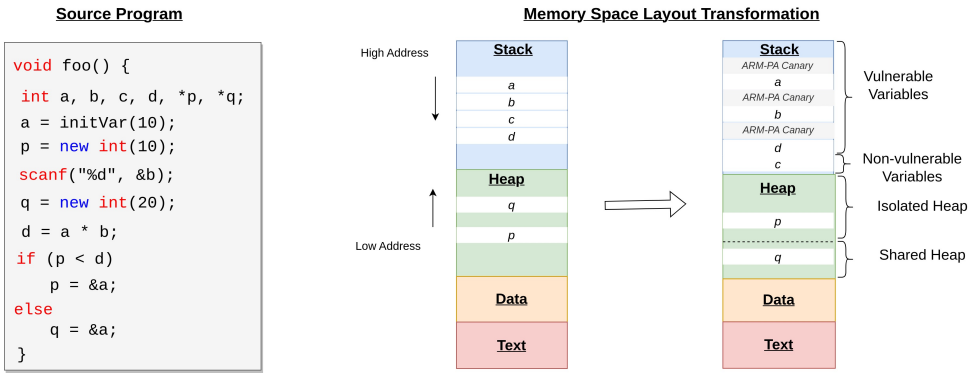

Figure 3. Illustration of *Pythia*'s defensive mechanism on a simple program. The statically vulnerable allocated variables a, b, d are relocated to the bottom of stack memory, and equipped with ARM-PA encrypted canaries. The dynamically allocated vulnerable variable p is moved to the *isolated* section of heap memory, where it is encrypted.

of performing branch decomposition, and then input channel construction, only for dynamically allocated program variables (lines 2-10). The heap sectioning procedure marks specific memory address regions in the heap memory as isolated (line 11), vulnerable variables are allocated in those 'safe' addresses (line 13). The portion of heap memory distributed between isolated and shared regions can be adjusted based on the number of secure heap variables. Finally, ARM-PA is leveraged to encrypt these variables (line 14), and all of their uses (lines 15-18).

4.4 Analyzing Pythia's Security Strengths & Overheads

⋆ **Instruction Overhead**: In contrast to the conservative scheme (§4.2), the performance-aware scheme of *Pythia* reduces the total number of ARM-PA instructions that are instrumented in the program. For both statically allocated program variable sv_i and dynamically allocated program variable dv_i, this scheme adds one encryption and one decryption for every use du_i with input channel. If a program has sv statically allocated variables and hv dynamically allocated variables, with du number of uses (on average), an upper-bound of additional ARM-PA instructions (I) can be obtained as:

$$I \leq B\left[sv\ (1+3\ du) + hv\ (1+2\ du)\right] \tag{2}$$

$$I \leq B\left[sv\ (1+2\ du) + sv\ du + hv\ (1+2\ du)\right] \tag{3}$$

$$\leq B\left[(1+2\ du)\ v' + sv\ du\right] \tag{4}$$

$$\approx B\ (1+2\ du)\ v' << B\ (1+2\ u)\ v \tag{5}$$

In Eq. 5, v' represents the sum of vulnerable statically and allocated variables. Comparing this with the conservative scheme (Eq. 1), we find that the number of refined variables is much lesser than the actual vulnerable variables, i.e $v' << v$. Thus, in practice the upper bound in Eq. 5 leads to a much smaller value than in Eq. 1, despite the extra $sv \times du$ term.

⋆ **Tackling Interprocedural Overflows**: A special case of control-flow bending can occur during function calls. When a function (caller) calls another function (callee) within its body, it might be possible for the callee function to trigger a buffer overflow which might spill into caller's stack canaries. This typically happens when the callee function's arguments are passed by reference (or pointers). For statically allocated variables passed by references (or pointers), *Pythia* performs alias analysis to check if they may point to any of the vulnerable variables, and stores their value in a global pointer canary (authenticated with ARM-PA). For dynamically allocated variables passed by pointers, *Pythia* checks if such variables are aliased with interprocedural heap allocation functions (e.g. *malloc*) and just uses the pointers passed. If we get a case such that the variable passed is a statically allocated variable along one call chain and a dynamically allocated variable, we pass the dynamically allocated variable as the argument and in the global pointer for canary because we would have authenticated it with ARM-PA along the statically allocated variable call chain. Therefore, with the use of global pointers to canaries, *Pythia* can detect buffer overflows that span across different function calls.

⋆ **Handling Brute-Force/Canary Attacks**: Encryption and authentication-based security mechanisms such as ARM-PA can often be susceptible to brute-force attacks where an attacker repeatedly runs the application to guess the canary values (pointer authentication code) correctly. *Pythia* ensures that the canary values are re-randomized on every entry to the function. A wrong guess will crash the program which will force the attacker to guess the canary value across different executions of the application. This makes each program invocation independent of the previous attempt. Therefore, the probability of an adversary guessing the canary value of a Linux system with 24-bit PAC correctly within N repeated attempts for a program with k canary:

$$\mathbb{P}(\text{Brute Forcing}) = k\left(1 - \frac{1}{2^{24}}\right)^{N-1}\frac{1}{2^{24}} \approx \frac{k}{2^{24}} \quad (6)$$

Eq. 6 shows that there is *1 in 16 million chance that a brute-force attack can successfully guess the authentication value for one canary*. In addition, the brute force can be modeled as a geometric random variable. As a result, the expected number of tries ($E[X]$) is $1/p$ where $p = \frac{1}{2^{24}}$ meaning it will take 16 million tries before the attacker can figure out a canary. For k canaries, it just needs to divide these chances over those k canaries.

In addition, we re-randomize whenever the canary's neighbor stack variable will be used by an input channel. As a result, any value extracted through a buffered read would be useless since the canary's value had changed already. One important point to note here is that because of this re-randomization, the "window" in which an attacker can break a canary exists only during the specific function invocation, between the start of the function's execution and the load instruction protected by the canary.

5 Implementation

Pythia was implemented as a unified set of compiler passes in LLVM 14. The *Pythia* code is split between the LLVM module passes, LLVM machine code passes, the random library, and the secure memory allocation library. The codebase is around ~3420 LOCs including some edits to LLVM original files to include our passes and intrinsics. The secure memory allocation is based on *glibc malloc* implementation.

Module Pass: The algorithms presented in the paper are implemented as LLVM Module Pass. LLVM's *mem2reg* pass transforms the program IR by promoting memory references into register references, thereby reducing the loads/stores. We created intrinsic functions for ARM-PA encryption for the remaining *loads*, *stores*, and *alloca* instructions, along with metadata for the backend machine pass.

Alias Analysis: *Pythia* uses LLVM's in-built alias analyses (*basic-aa, globals-aa, aa*, and *tbaa*) for handling pointers in backward and forward program slices.

Machine Pass: To handle register spills at the machine code generation level, we leverage the instrumented metadata and intrinsics to detect additional encryption & authentication points. In addition, we use the same data to add canaries that were missed or need to be moved. The canaries are populated with C++ random number generator with a library call at each invocation of the function, and right before the input channel for stack variables.

6 Evaluation

The evaluation of *Pythia* answers the following set of questions about program security and performance overheads:

- How effective is the conservative scheme in defending against non-control data attacks, and what are its runtime overheads?
- How do ARM-PA instructions and heap sectioning instructions affect the performance of benchmarks?
- 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?
- Can Pythia be effectively used to detect the control-flow bending attacks with low overheads in real-world examples?

Experimental Setup: The experiments were conducted on an Apple MacBook M1 Pro, running Linux Ubuntu 22.04. The system has 10-cores CPU (8 performance cores at 3.2 GHZ and 2 efficiency cores at 2.06 GHZ) with 16-core GPU and 16-core neural engine and 24 MB L3 cache. Our experiments were ran without frequency scaling or any manual core-scheduling.

Benchmark Programs: Our experiments were performed on programs from SPEC 2017 Benchmarks [10] on ref inputs, Nginx [57] and also on representations of real-world examples that have control flow bending vulnerabilities shown in §2.2.

Performance Baselines: We evaluate *Pythia*'s performance comparison with two different baselines:

- **Vanilla Execution**: Application is compiled with O3-flag without adding any new instruction.
- **Complete Pointer Authentication (CPA)**: Conservative defensive scheme described in §4.2, where all the (un-refined) vulnerable variables are simply encrypted with ARM-PA.

6.1 Performance Evaluation

The performance results are normalized against the vanilla execution baseline, where no security mechanism is utilized.

⋆ **Complete Pointer Authentication (CPA)**: CPA scheme encrypts the un-refined set of vulnerable program variables. As illustrated in Fig. 6(a), the vulnerable variables set in CPA consist of about 29% of all program variables on average. These variables are encrypted at least once (initial store), and decrypted at least once (all live program variables have at least one use). Overall, since the set of un-refined variables is constituted by a significant number of program variables, the total number of ARM-PA instructions that are added to the program is significant. Furthermore, any spills due to register allocation will lead to even more ARM-PA instructions instrumented in the program (Fig.6(b)). After the addition of PA instructions, CPA incurs an average overhead of **47.88%**

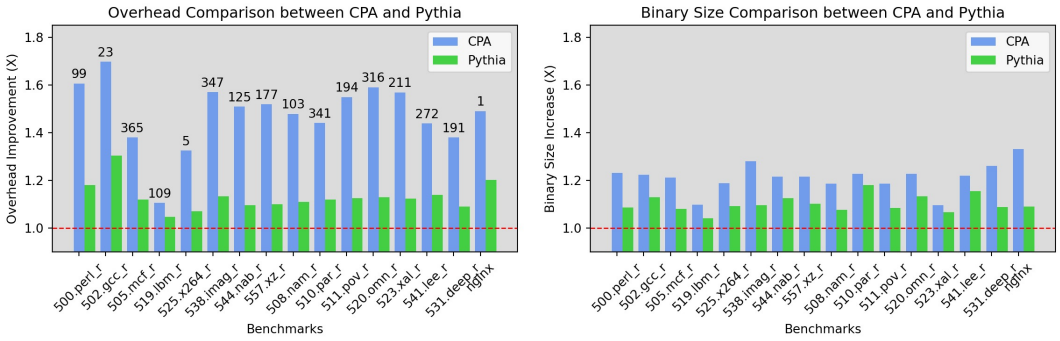

Figure 4. (a) Runtime overhead comparison between CPA and Pythia. (b) Binary size comparison between CPA and Pythia. The baseline absolute numbers for spec in secs and nginx in GB/s are shown on top.

with a worst-case of **69.8%** as seen in Fig. 4(a). The worst case is 502.*gcc_r* which has the most number of vulnerable variables resulting in a maximum number of ARM-PA instructions added.

The extra ARM-PA instructions also adversely affect the IPC count, and application binary sizes (Fig. 4-5). However, the IPC does not suffer radically since ARM-PA directly leverages hardware support in variable authentication. As shown in Fig. 5(a), the average IPC degradation for CPA scheme is around **4.9%**, with the worst IPC degradation of **13%** in 523.*xalancbmk_r*. This is caused by repeated execution of ARM-PA instructions inside loop nests. Similarly, the addition of ARM-PA instructions causes application binaries to bloat. As shown in Fig. 4(b), the average binary size increased by **21.56%**, with the maximum of **33.2%** in *nginx*. Furthermore, increased instructions in the program lead to additional LLC misses.

⋆ **Pythia**: In contrast to CPA, Pythia refines the set of vulnerable variables by performing input channel construction on them. As a result, Pythia reduces the number of vulnerable program variables by about **4.5x** (Fig.6(a)). Another major justification for refining the set of vulnerable program variables is the observation that ~74% of conditional branches in all benchmarks are not affected by input channels at all. Instead, only 1.26% directly affected conditional branches and 25.1% indirectly affected conditional branches can result in control-flow bending. Overall, only *5.1%* of program benchmarks are marked as vulnerable.

Aside from vulnerable variable refinement, the stack relayout and heap sectioning in *Pythia* also decreases the overall amount of ARM PA instructions. However, stack canaries add new stack define instructions (mov), call random number generator library function, and load/store instructions for encryption/decrypted, which adds up to the overhead. The library call for heap sectioning adds an extra overhead of about 23 ns on average.

Thus, these factors combine to minimize the program's runtime overhead when secured with *Pythia*. As shown in Fig. 4, *Pythia*'s performance overhead dropped to an average of 13.07% with the most noticeable change in 500.*perlbench_r* from **60.7%** (CPA) to **18%**. Maximum overhead in the *Pythia* scheme was **25.4%** (502.*gcc_r*). *Pythia*'s program IPC degradation (Fig. 5) decreased by **2.8%** on average - which bears testimony to the fact that adding ARM-PA selectively increases the opportunities of out-of-order processing, where more instructions can be completed in the same cycle. Note that heap sectioning makes the heap memory more fragmented. In case heap variables from isolated and shared sections are accessed consecutively, cache misses might increase due to non-local accesses. This is why certain benchmarks (510.*parest_r*) have slightly more cache misses in *Pythia* over the baseline. Another implication of instrumenting less ARM-PA instructions is that the application binary size decreased to **10.37%**, with 510.*parest_r* having the highest binary bloat with **17.99%**.

6.2 Security Mechanism Evaluation

⋆ **Input Channel (IC)**: Our experiments found 25326 input channels functions across the 16 benchmarks, whose distribution is presented in Fig. 5(b). As seen in the figure, the most common input channel functions across the benchmark are *print* (31.5%) and *move/copy* (65.9%). The rest (*map, scan, get, put*) account for only 2.6% of the input channels. These input channels are either predefined library functions (such as printf), or custom user-implemented versions. Our experiments have revealed that such input channel functions often get translated as intrinsics in the LLVM IR, especially in C++ benchmarks, making detecting their presence easier. Benchmarks (such as 510.*parest_r*, 502.*gcc_r*) contain the maximum input channels.

⋆ **Pointer Authentication** : The CPA baseline instrumented a total of 5×10^5 PA instructions across all the benchmarks. Specifically, 502.*gcc_r* and 510.*parest_r* have the maximum number of PA instructions (1.3×10^5 each). As seen in Fig. 6 (b), Pythia dramatically decreased total PA instructions to 1.1×10^4 with 510.*parest_r* having the most with 59, 680

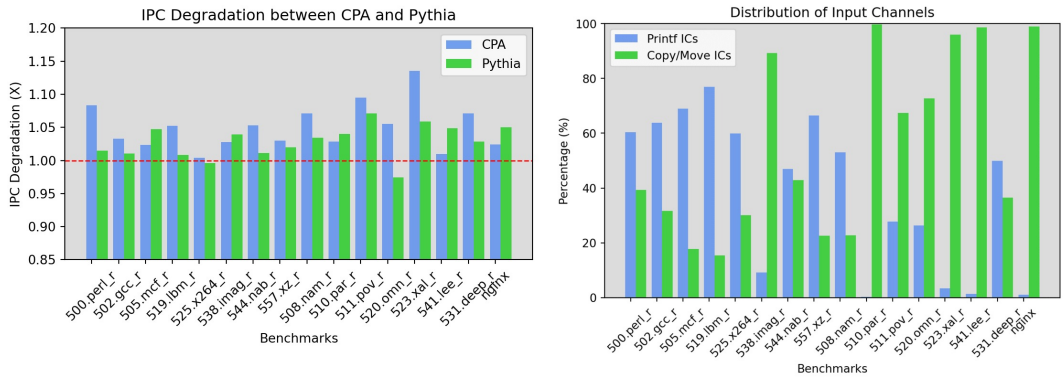

Figure 5. (a) IPC degradation comparison between CPA and Pythia. (b) Distribution of printf ICs and copy/move ICs based on the total ICs in the benchmark.

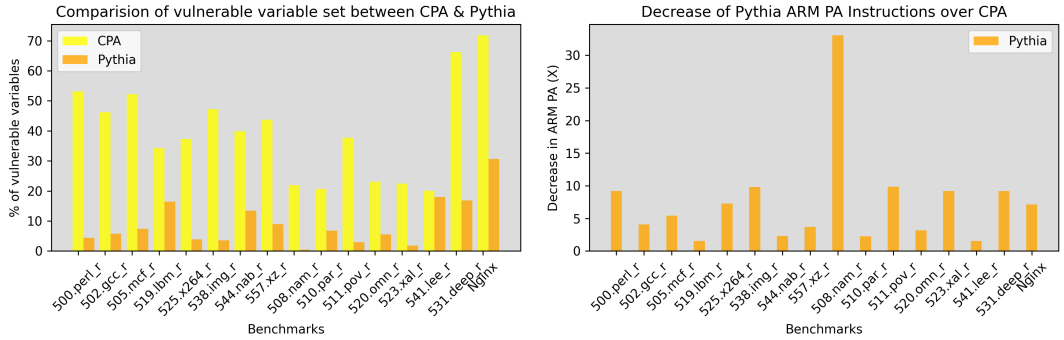

Figure 6. (a) Distribution of vulnerable variables and ARM-PA instructions between CPA and Pythia scheme. (b) ARM PA instructions decrease in Pythia over CPA.

instructions. Practically, in both the schemes only **50%** of instrumented PA instructions are executed dynamically.

⋆ Stack Canaries + Heap Relocation: As mentioned earlier, more than ~99% variables being used in input channels are stack variables (around ~29300). *Pythia* which adds one canary per stack variable, thus adding ~ 29300 canaries across all benchmarks. The CPA scheme requires an encryption during the stack allocation, and then a decryption for loading into input channel function, followed by an encryption at store (Eq. 1). In contrast, the *Pythia* scheme adds one extra layer of encryption before loading the input channel use (Eq. 5). This extra ARM-PA encryption helps *Pythia* save many added instructions in case of register spills. For example, a variable spilled twice in the CPA Scheme would have 7 PA instructions (4 encrypts and 3 decrypts), while the *Pythia* requires only 4 PA instructions (3 encrypts and 1 decrypt right after the input channel). This reduction builds up significantly across all statically allocated program variables.

Compared to statically allocated variables, dynamically allocated vulnerable program variables that get relocated by heap sectioning are significantly less prevalent in the benchmarks. However, our analysis has shown that such variables are usually utilized inside program loops. As a result, the size of the isolated heap section is scalable. Furthermore, benchmarks like 519.*lbm_r* and 505.*mcf_r* which don't have any vulnerable heap variables, incur overheads because of heap sectioning (~ 126*ns* on average).

Attack Distance + Branch Security: Lastly, we compare the attack detection capabilities of *Pythia* with a state-of-the-art Data-flow Integrity (DFI) mechanism [14]. Across all the benchmarks, the average distance of input channels from respective branches is 83.29 LLVM instructions with the longest distance being 500.perl_r with 149.76 LLVM instructions. DFI is unable to reason about pointer arithmetic and field sensitivity cases. Therefore, DFI's average attack distance is about **113.95** LLVM instructions since its backward slice mechanism terminates whenever it encounters pointer arithmetic and field sensitive cases being unable to reason about them regarding their data-flow. On the other hand, *Pythia*'s average attack distance is **127.35** LLVM instructions. Pythia focuses on protecting all the variables encountered using PA authentication rather than relying on verifying underlying data flow leading to longer backward slices. However, in some cases, Pythia cannot extend the backward slice

to the input channel due to complex inter-procedural alias analysis encountered, which we do not tackle currently.

In addition, we compare the branch protection capability of DFI and Pythia (please refer to Fig. 7(b)) as it shows the security strength of both techniques. Recall that a given technique protects a branch if the technique can generate and protect branch's backward slice to the input channel. Due to Pythia's backward slice generation and ARM-PA protection capability, Pythia protects an average of 92% branches across all benchmarks against DFI's 86.6%. Looking at the benchmark breakdowns, we can see Pythia offers more protection than DFI, ranging from 0% to 17% with an average of 5.6%. One important point to note here is that a small percentage difference can result in a huge number of actual conditional branches (2% difference in 502.*gcc_r* results in 7000 more branches, and 7% in 510.*parest_r* leads to 140000 more branches being protected by Pythia). In general, Pythia's protection is stronger for C++ codes mainly due to complex pointer operations in the benchmark that terminate the backward slices of DFI. Pythia offers over 90% protection for 13 benchmarks, whereas DFI offers over 90% protection for only 9 benchmarks. If we look at 100% protection cases (all branches are secured), DFI only provides perfect protection for 519.*lbm_r*, which has only 75 branches. In contrast, Pythia can fully secure three benchmarks (519.*lbm_r*, 505.*mcf_r* and 525.*x*264*_r*), where 525.*x*264*_r* has over 7000 branches. Overall, DFI and Pythia perform similarly in non-pointer arithmetic cases, whereas Pythia has a significant advantage in pointer-heavy and C++ code.

6.3 Real-World Examples

⋆ **Nginx** : We evaluated *Pythia* on *nginx* [57], which is a well-known web server that needs strict security guarantees. Recent DOP attacks [17] has exposed the vulnerabilities of Nginx. In addition, this application is multi-threaded, which will be useful in stress testing the heap sectioning framework. In our experiments, we used the same experimentation scheme described in Blankit [62]. Nginx has a workload generator with 12 threads to create 400 concurrent connections. We use nginx's workload generator to satisfy requests for Wikipedia's home page for 3s, 30s, and 300s. The overheads for nginx are based on the transfer rate degrading or not.

By averaging the performance across the three runs, the CPA runtime overhead is around 49.13%, while Pythia drops it to 20.15%. Nginx also uses a mixture of input channels from glibc and their implementation variations beginning with "ngx_". Despite having significantly fewer variables, it has many input channels (720) with the majority being copy/move input channels (712). In nginx, there is a very high loop in the call chain, so the PA instructions added will be repeatedly executed. As mentioned, Pythia has a significantly higher attack distance than DFI for nginx and also protects 300 more branches.

⋆ **Motivating Examples**: We rewrote the motivating examples (§2.2) so that they could be tested on *Pythia*. We added some extra instructions to prevent code restructing due to optimizations by the compiler.

The first example is the *String-Buffer Overflow* attack as seen in Listing 1. The critical vulnerability is the input channel 'strcpy' on line 14. *Pythia* identifies it as an input channel, and classifies 'someinput' as a stack variable. It will place 'someinput' at the higher address of the current function's stack frame and add a canary after it with a random value. A simple authentication check after the canary determines whether the value has changed.

The second example of ProFTPd Vulnerability in Listing 2 is similar. In this case, the input channel 'sstrncpy' will affect rptr causing the overflow. Like the previous example, it will take the variable to the higher address in the call stack. Pythia creates the canary with a random value and encrypts it initially then re-encrypts before the input channel. There is an authentication after its use in the input channel. Any overflow will crash the program during the canary's check.

The third example in Listing 3 also has an overflow issue. In this case, the overflow happens from 'k' into 'l'. The encryption and authentication will detect the overflow immediately after the input channel.

6.4 Limitations

Pythia cannot detect stack buffer overflows resulting within objects such as sub-fields of a *struct*. If this overflow affects another object, Pythia's stack canaries can detect it immediately. To solve this problem of overflow detection within sub-fields, stack canaries must be inserted within individual fields. Furthermore, with precise alias analysis, the specific fields being used by the input channels can be detected, and canaries can only be created for such fields. This is a focus of our future work.

7 Related Work

Memory Safety: One of the possible way to tackle non-control data attacks (including control-flow bending) is to prevent an adversary from exploiting memory errors by ensuring general memory safety [24, 36, 54–56, 60, 66]. These techniques prevent illegal memory access by introducing non-trivial language extensions. However, memory safety techniques have high overheads compared to Pythia. For example, on legacy applications (only C applications), SoftBound [54] has an average overhead of 67% and Softbound + CETS [55] has an average overhead of 116%. Other bound-checking based techniques that only handle spatial memory errors, such as *ASAN* [68], has an average overheads of 76% (with slowdowns upto 2.67x), heap-only approach *LowFat-Pointer* [26] with an overhead of *113%*, and *LBC* [30] with an overhead of *23%* (legacy applications). Our technique currently has 13.3% overhead with all the added instructions.

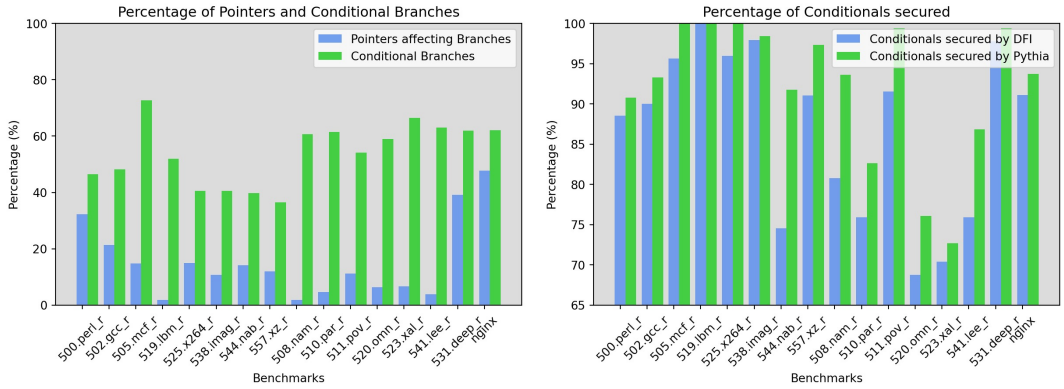

Figure 7. (a) Percentage of variables in the backslice that are pointers. In addition, the percentage of conditional branches in a given benchmark. (b) Comparison between the percentage of conditional branches secured by DFI vs Pythia.

The compiler-based [24, 36, 56] that type-safe pointers are specifically made for C. Our work has been evaluated on both C and C++ benchmarks. Other works focused on bound checking the pointer memory accesses to ensure they don't access unauthorized memory locations [40, 42, 69, 79]. Most of these works require specific hardware processors or extensions. Pythia utilizes hardware extensions already available in commercial ARM chips (seen in Apple products and Graviton servers [4]).

Stack Canaries-based Mechanisms: Majority of defense mechanisms [63, 72, 78] that use stack canaries to protect against non-control data attacks focus exclusively on stacks, and usually do not defend against heap-based vulnerabilities. Moreover, such techniques are also vulnerable to inter-procedural overflows, and stack bypassing where an adversary can try to leak the canary value by performing buffer reads. *Pythia* mitigates this by randomizing the canary value before every input channel, which minimizes the probability of such attacks. In addition, these techniques are stack-specific so they do not solve the heap buffer overflows.

Address Randomization: Another alternative method to combat data attacks is to simply prevent an adversary from locating privileged information by randomizing the data layout [6–8, 28, 44]. Note that such techniques are geared towards making it harder for the adversary to guess the memory addresses - they are not designed to prevent control-flow bending attacks like Pythia. Furthermore, to minimize overheads, such techniques often randomize only a part of the program data [71]. Recent works have also focused on randomizing stack layout to reduce the probability of leaking statically allocated safety-critical data [3, 8, 46, 50, 74]. Some randomization techniques will figure out the actual locations at runtime, or create somewhat of a padding. We simply move the variables around at the compile stage so there is no runtime overhead for randomization and our canaries for vulnerable variables provide the least amount of padding needed for stack protection.

Data-Flow Integrity (DFI) [14]: The goal of DFI is to first compute a static data-flow graph and then verify whether the transfer of dataflow facts at the runtime is permitted by the graph or not. The problem with this approach is that it requires maintaining and checking the dynamic dataflow information at runtime using SETDEF and CHKDEF for every program variable, which causes overheads of up to 2.5x. In particular, DFI is unable to reason about pointer arithmetic and field-based alias analysis which results in its ability to construct backward slices that can cover an input channel.

8 Conclusion

In this work, we proposed *Pythia*, a compiler-guided defense framework that combines traditional compiler analysis with pointer authentication. *Pythia* prevents control flow bending by isolating vulnerable variables, tackling statically allocated variables by re-orienting and adding canaries, and dynamically allocated variables by heap sectioning. In our evaluation, we found that Pythia's performance-aware approach of using the isolation and pointer authentication has an average overhead of 13.07% compared to the complete pointer authentication baseline of 47.88%, without compromising security guarantees. In addition, Pythia can secure 5.6 % branches more than DFI and fully secure 3 applications. Thus, it shows its effectiveness on pointer-intensive applications and C++ codes in terms of coverage of input channels.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ashish Venkat for their invaluable comments to improve the paper significantly.

References

[1] Cwe top 25 most dangerous software weaknesses, 2022.
[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM*

Transactions on Information and System Security (TISSEC), 13(1):1–40, 2009.

[3] Misiker Tadesse Aga and Todd Austin. Smokestack: thwarting dop attacks with runtime stack layout randomization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 26–36. IEEE, 2019.

[4] Amazon graviton2. https://aws.amazon.com/ec2/graviton/, 2019. Accessed: 2021 April 16.

[5] ARM ARM. Architecture reference manual-armv8, for armv8-a architecture profile. *ARM Limited, Dec*, 2017.

[6] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 337–358. Springer, 2018.

[7] Sandeep Bhatkar and R Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[8] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, volume 16, pages 21–24, 2016.

[9] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. 2007.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, New York, NY, 2018. ACM.

[11] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[12] Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou, Ajay Joshi, and Manuel Egele. Efficient context-sensitive cfi enforcement through a hardware monitor. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 259–279. Springer, 2020.

[13] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 161–176, 2015.

[14] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[15] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387. IEEE, 2005.

[16] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[17] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.

[18] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[19] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[20] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307. IEEE, 2014.

[21] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[22] Lucas Davi and Ahmad-Reza Sadeghi. Building control-flow integrity defenses. In *Building Secure Defenses Against Code-Reuse Attacks*, pages 27–54. Springer, 2015.

[23] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 401–416, 2014.

[24] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. *ACM SIGPLAN Notices*, 41(6):144–157, 2006.

[25] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 131–148, 2017.

[26] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[27] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, 2012.

[29] Guang Gong. Exploiting heap corruption due to integer overflow in android libcutils. *Black Hat USA*, 2015.

[30] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[31] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Pacsafe: Leveraging arm pointer authentication for memory safety in c/c++. *arXiv preprint arXiv:2202.08669*, 2022.

[32] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 177–192, 2015.

[33] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1470–1486, 2018.

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[35] Hyerean Jang, Moon Chan Park, and Dong Hoon Lee. Ibv-cfi: Efficient fine-grained control-flow integrity preserving cfg precision. *Computers & Security*, 94:101828, 2020.

[36] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[37] Dongjae Jung, Minsu Kim, Jinsoo Jang, and Brent Byunghoon Kang. Value-based constraint control flow integrity. *IEEE Access*, 8:50531–50542, 2020.

[38] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE, 2019.

[39] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 195–211, 2019.

[40] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1153–1166. IEEE, 2020.

[41] Benjamin A Kuperman, Carla E Brodley, Hilmi Ozdoganoglu, TN Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, 2005.

[42] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[43] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[44] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[45] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*. Citeseer, 2015.

[46] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2559–2575, 2021.

[47] Kyung-Suk Lhee and Steve J Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: practice and experience*, 33(5):423–460, 2003.

[48] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1901–1915, 2022.

[49] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[50] Yu Liang, Xinjie Ma, Daoyuan Wu, Xiaoxiao Tang, Debin Gao, Guojun Peng, Chunfu Jia, and Huanguo Zhang. Stack layout randomization with minimal rewriting of android binaries. In *Information Security and Cryptology-ICISC 2015: 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers 18*, pages 229–245. Springer, 2016.

[51] 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 *USENIX Security Symposium*, pages 177–194, 2019.

[52] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[53] Girish Mururu, Sharjeel Khan, Bodhisatwa Chatterjee, Chao Chen, Chris Porter, Ada Gavrilovska, and Santosh Pande. Beacons: An end-to-end compiler framework for predicting and utilizing dynamic loop characteristics. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):173–203, 2023.

[54] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[55] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[56] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[57] nginx. https://nginx.org/, 2019. Accessed: 2021 Oct 10.

[58] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[59] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584, 2010.

[60] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. Hardscope: Hardening embedded systems against data-oriented attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[61] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–164. Springer, 2015.

[62] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: getting what you want instead of cutting what you don't. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 164–180. ACM, 2020.

[63] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. Privguard: Protecting sensitive kernel data from privilege escalation attacks. *IEEE Access*, 6:46584–46594, 2018.

[64] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX security symposium*, pages 169–186, 2009.

[65] Robert Schilling, Pascal Nasahl, and Stefan Mangard. Fipac: Thwarting fault-and software-induced control-flow attacks with arm pointer authentication. In *Constructive Side-Channel Analysis and Secure Design: 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*, pages 100–124. Springer, 2022.

[66] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[67] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. 2010.

[68] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[69] Rasool Sharifi and Ashish Venkat. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *2020*

ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 762–775. IEEE, 2020.

[70] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[71] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[72] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. Valueguard: Protection of native applications against data-only buffer overflows. In *Information Systems Security: 6th International Conference, ICISS 2010, Gandhinagar, India, December 17-19, 2010. Proceedings 6*, pages 156–170. Springer, 2010.

[73] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940, 2015.

[74] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 727–741, 2016.

[75] Tielei Wang, Chengyu Song, and Wenke Lee. Diagnosis and emergency patch generation for integer overflow exploits. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings 11*, pages 255–275. Springer, 2014.

[76] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, pages 1–14, 2009.

[77] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, Guimin Zhang, and Zhihui Shi. Bci-cfi: A context-sensitive control-flow integrity method based on branch correlation integrity. *Information and Software Technology*, 136:106572, 2021.

[78] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. To detect stack buffer overflow with polymorphic canaries. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 243–254. IEEE, 2018.

[79] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[80] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[81] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[82] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[83] Guimin Zhang, Qingbao Li, Zhifeng Chen, and Ping Zhang. Defending non-control-data attacks using influence domain monitoring. *KSII Transactions on Internet and Information Systems (TIIS)*, 12(8):3888–3910, 2018.

[84] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

[85] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 558–569, 2014.

[86] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Using branch correlation to identify infeasible paths for anomaly detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 113–122. IEEE, 2006.