University of Deusto

New methods for the integrity of the data flow in operating systems and their applications

Dissertation by Irene Díez Franco

within the doctoral program Engineering for the information society and sustainable development

for the degree of Doctor of Philosophy

Candidate	Advisor	Co-advisor
Irene Díez Franco	Dr. Pablo García Bringas	Dr. Xabier Ugarte-Pedrero

Para los tres chospos.

Abstract

Thanks to the widespread deployment of information security techniques that protect applications and operating systems against control flow hijacking attacks, malicious actors face increased difficulties to exploit computer systems. This, however, has a downside, attackers are becoming more imaginative and try to find new and increasingly complex vulnerability exploitation techniques.

One of these new techniques is based on exploiting the non-control data of a program with malicious intent, and the unfortunate news is that neither operating systems nor their applications currently deploy any known defences against this kind of attacks.

In this dissertation we propose a compiler-based optimised defence based on the data-flow integrity property that allows practitioners to compile applications with security mechanisms in order to provide defences against non-control-data attacks.

This defence has been built on top of the GCC compiler, allowing a widespread adoption and usage by any C application that can be compiled with GCC.

Our implementation is set apart from previous works in the granularity and precision of its static analysis, providing broader security guarantees. Moreover, we provide two novel optimisations that on one hand give full control to the users so that they can define which types of non-control-data they wish to protect in their applications, and on the other hand allow to reduce the amount of basic blocks that the GCC compiler needs to protect by 45.8% in average, whilst maintaining the security guarantees.

Resumen

Gracias al amplio despliegue de técnicas de seguridad informática que protegen a los usuarios finales de ataques de subversión del flujo de datos de control en sus aplicaciones informáticas y sistemas operativos, los actores maliciosos se enfrentan cada día a mayores dificultades para explotar aplicaciones informáticas. Esto sin embargo tiene una consecuencia negativa, ya que los atacantes se vuelven más imaginativos y tratan de buscar nuevas técnicas de explotación de vulnerabilidades.

Una de estas nuevas técnicas se basa en explotar los datos de no-control de un programa para fines maliciosos y la desafortunada noticia es que ni los sistemas operativos ni sus aplicaciones pueden desplegar ningún tipo de defensa conocida contra este tipo de ataques.

En esta tesis proponemos una defensa optimizada basada en el propiedad de la integridad del flujo de datos mediante compiladores que permite compilar aplicaciones con mecanismos defensivos contra ataques basados en datos de no-control.

Esta defensa ha sido desarrollada sobre el compilador GCC, por lo que permite una adopción y uso amplio por cualquier aplicación en C que pueda ser compilada con GCC.

Nuestra implementación se diferencia de trabajos previos en la granularidad y precisión de su análisis estático, ofreciendo garantías de seguridad más amplias; además, incluimos dos optimizaciones inéditas que por un lado, permiten dar el control total al

usuario, para que este pueda definir qué tipos de datos de no-control quiere proteger en sus aplicaciones, y por otro lado permiten reducir en una media de un 45.8% la cantidad de bloques básicos mientras que simultáneamente se mantienen las garantías de seguridad.

Acknowledgements

El camino que ha llevado a la conclusión de esta tesis doctoral ha sido largo y lleno de baches. Me gustaría dar las gracias a los directores de esta tesis, Pablo y Xabi, por su apoyo y paciencia durante el camino.

A mi familia por su apoyo incondicional y confianza. A los compañeros de DeustoTech, con los que he compartido muchas comidas, desde la época del *búnker* hasta la cafetería, especialmente a Oihane, Aitor, Aritz y Adrián por la asistencia de último momento.

Eskerrik!
Irene Díez Franco

Publications

Some of the contributions of this dissertation have already been published in the following journals or venues:

International JCR journals

- *"Optimized Data-Flow Integrity for Modern Compilers."*
  Irene Díez-Franco, Xabier Ugarte-Pedrero and Pablo García Bringas. (2024)
  In IEEE Access. (Q1, 2023) doi: 10.1109/ACCESS.2024.3454551.

International conferences

- *"Understanding the Security Landscape of Control-Data and Non-Control-Data Attacks Against IoT Systems."*
  Irene Díez-Franco, Pablo García Bringas and Xabier Ugarte-Pedrero. (2024)
  In 9th International Conference on Smart and Sustainable Technologies (SpliTech), Bol and Split, Croatia, 2024, pp. 01-06, doi: 10.23919/SpliTech61897.2024.10612517.
- *"Feel Me Flow: A Review of Control-Flow Integrity Methods for User and Kernel Space"*
  Irene Díez-Franco and Igor Santos. (2016)
  In Proceedings of the International Conference on Computational Intelligence in Security for Information Systems (CISIS): San Sebastián, Spain, 2016 Proceedings 11. Springer International Publishing, 2017. doi: 10.1007/978-3-319-47364-2_46

- *"Data is flowing in the wind: A review of data-flow integrity methods to overcome non-control-data attacks".*

  Irene Díez-Franco and Igor Santos. (2016)

  In Proceedings of the International Conference on Computational Intelligence in Security for Information Systems (CISIS): San Sebastián, Spain, 2016 Proceedings 11. Springer International Publishing, 2017. doi: 10.1007/978-3-319-47364-2_52,

Table of Contents

List of Figures

List of Tables

List of Listings

Acronyms

ASLR Address Space Layout Randomization

BB Basic block

CFG Control Flow Graph

CFI Control Flow Integrity

COP Call-Oriented Programming

CPI Code Pointer Integrity

DEP Data Execution Prevention

DFG Data Flow Graph

DFI Data Flow Integrity

DMA Direct Memory Access

ICFG Inter-procedural Control Flow Graph

JOP Jump-Oriented Programming

KASLR Kernel Address Space Layout Randomization

KPTI Kernel Page-Table Isolation

LAF Local Address Frame

LBR Last Branch Record

OS Operating System

PLT Procedure Linkage Table

RDT Runtime Definitions Table

ROP Return-Oriented Programming

SAF Shared Address Frame

SDFG Static Data Flow Graph

SMAP Supervisor Mode Access Prevention

SMEP Supervisor Mode Execution Prevention

SSA Static Single-Assigment

TCB Trusted Computing Base

VCL Variable-Context List

W⊕E Write xor Execute

We're here to make magic with words.

R.F. Kuang, Babel

CHAPTER

1

Introduction

Operating systems are the last line of defence against cybersecurity threats, thereby they incorporate a wide variety of defences that are used in conjunction with user space defences to protect applications against threats. As attackers evolve and develop new exploitation techniques, the defences of the operating systems and their user space counterparts must adapt, so that both their own code and the code of their applications remain protected.

Nowadays, memory errors that target the *control data* of a program (e.g. return addresses) are harder to exploit due to core defences provided by every major operating system. Modern compilers provide compilation options to protect user space applications as well. However, these operating systems and applications still remain completely exposed against threats that, as opposed to *control data*, target what is known as *non-control data* or *pure data* instead (e.g. boolean values, configuration data). The objective of this dissertation is to compensate for this limitation, advancing the state-of-the-art in defences that protect *non-control data*.

The rest of this chapter is organised as follows. Section 1.1 gives an historical context and motivation for the execution of the research presented in

this dissertation. Section 1.2 formulates the hypothesis that drives this dissertation, as well as the specific and operational objectives. Section 1.3 explains the methodology followed for the completion of this dissertation. Finally, Section 1.4 outlines the organisation of the remainder of this document.

1.1 Historical context and motivation

Since the inception of computers, malicious users have tried to exploit existing memory errors to subvert the intended usage of applications in order to accomplish a malicious goal, such as stealing private information (Jones, 2024a) or disrupting critical infrastructure (Speed, 2024; Jones, 2024b; Lyons, 2024) often with the intent of causing economical and sometimes personal harm. These exploits have evolved from classic buffer overflows (Spafford, 1989) in the late 80s to advanced code-reuse exploits, but they all have in common that they target *control data*, which lead to the research and development of defence techniques both in user and kernel space that try to protect control data; for instance, one of the earliest results is `StackGuard` (Cowan et al., 1998), a compiler-based defence that has helped prevent buffer overflows in GCC-compiled programs since 1998.

From then on, the research community has been focusing on developing new techniques, each one more complete an efficient, that prevent control-data attacks. However, in the early 2000s Chen *et al.* raised awareness that not only exploiting *non-control data* was feasible, but also that none of the existing security defences at the time was capable of detecting attacks targeting such type of data (Chen et al., 2005). As a result, defence techniques were presented, being the data-flow integrity property (Castro et al., 2006) one of the most promising ones. This technique utilised the now deprecated Microsoft Phoenix Compiler Backend (Microsoft, 2008) to instrument programs, which did not become a mainstream security solution. From that time on, both the industry and the academia did not venture in more research projects trying to prevent non-control-data attacks and the topic remained stagnant.

Fast forward to the early 2010s, a non-control-data attack caused worldwide panic when the OpenSSL “Heartbleed” vulnerability was uncovered in

2014 (US-CERT, 2014; Chirgwin, 2014). From that day the research community has concentrated on demonstrating that non-control-data attacks are indeed a reality (Hu et al., 2015, 2016; Ispoglou et al., 2018; Johannesmeyer et al., 2024), and that their attack surface is broad, being capable of targeting user space applications, the Linux Kernel (Davi et al., 2017; Han et al., 2024) and web browsers (Jia et al., 2016) indiscriminately.

On the defensive side, efforts have been made that present specialised solutions (Song et al., 2016b; Bellec et al., 2022; Davi et al., 2017), that either require to deploy customised hardware, are tailored for a specific type of system, or cannot defend against the full spectrum of non-control-data attacks, thereby, those defences specialise in protecting against a specific subset of attacks or systems. However, the lack of a general-purpose defence against all types of non-control-data attacks that can be widely deployed is still apparent.

Other key factors that must be taken into account for the widespread deployment of defences against non-control-data attacks are the overhead that those defences incur, which is non-negligible in the case of data-flow integrity (from 45% to 105% (Castro et al., 2006)). This is particularly relevant in low resource environments such as IoT devices, where the applicability of such defences needs to be considered.

Our threat models shows that IoT devices are indeed susceptible to non-control-data attacks, and that these attacks have higher stakes since IoT devices are tied to the physical world and might cause direct physical and economical harm (Falliere et al., 2011; Shekari et al., 2022). This further justifies the need to research optimisations to mitigate the high overhead of defences against non-control-data attacks.

The historical landscape outlined in this section demonstrates that optimised generalist defences against non-control-data attacks, such as the one proposed in this dissertation, are key components in nowadays security ecosystem.

1.2 Hypothesis and objectives

Based on the existing problem stated in the previous section we formulate the following hypothesis:

> **Hypothesis.** *Although existing techniques to protect users against non-control-data attacks are specialised solutions, it is possible to develop a generalist and optimised solution that can be applicable to any environment.*

Considering this hypothesis we formulate the general objective of this dissertation:

> **General objective.** *Improve and optimise the defences targeting non-control data in order to provide generalist protections against non-control-data attacks.*

Moreover, this objective can be divided into the following specific objectives:

> **Specific objective 1.** *Conduct an analysis of the current security defences to analyse their general applicability and shortcomings.*

> **Specific objective 2.** *Study the applicability of the current threat models in the case of low-resource IoT systems.*

> **Specific objective 3.** *Improve the generalist defences against non-control-data attacks and explore possible optimisations.*

With the previous objectives defined, we outline the following operational objectives that will guide and help measure the results of this dissertation:

Operational objective 1. *Perform a thorough analysis of the current defence techniques to prevent control-data attacks.*

Operational objective 2. *Perform a thorough analysis of the current defence techniques to prevent non-control-data attacks.*

Operational objective 3. *Evaluate and propose a new threat model for IoT systems.*

Operational objective 4. *Develop and evaluate a general-purpose technique that can protect users against non-control-data attacks.*

1.3 Research methodology

The following section explains the different phases of the research methodology (refer to Figure 1.1) that has been followed for the completion of this dissertation.

1. **Definition of the research problem.** In this first phase a preliminary analysis of the literature is made to identify research opportunities as a consequence of open issues in the research area. During this phase a broad rather than an in depth analysis is essential to gather what are the key ideas of the topic. The output of this phase is the definition of the research area.

2. **Literature review.** Once the research opportunities have been established as a result of the first step, an in depth analysis of the publications in the relevant areas is conducted. In this step it is important to not limit the analysis to articles published in academical venues and journals, but also to include technical reports and relevant publications from the industry. After the completion of this phase, the expected output is the identification of the strengths and areas of improvement of the current approaches.

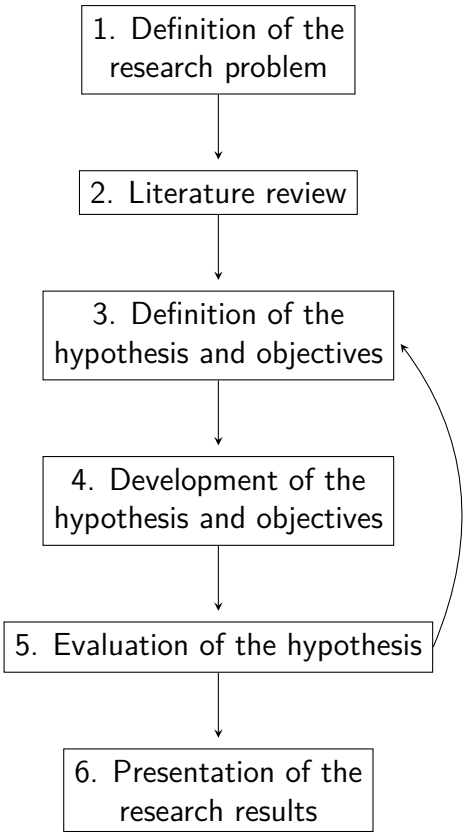


Figure 1.1: Phases of the research methodology followed for the completion of this dissertation.

3. **Definition of the hypothesis and objectives.** After the exhaustive literature review has been completed the research hypothesis are identified, with the subsequent definition of the research objectives that will guide the research towards the validation of the hypothesis. The completion of this step requires specifying a work plan with milestones and deadlines associated with the research objectives.

4. **Development of the research objectives.** Once planning has been completed during this phase we proceed with the development of the research objectives as defined in the work plan. During this phase the required skill-set for the completion of the work plan is learnt, if needed. The result of this phase is the completion of the research objectives.

5. **Evaluation of the hypothesis.** After the development of the research objectives is concluded the evaluation phase begins. During this phase the required tests or experiments for evaluating the hypothesis are defined and executed. The results of the experiments are evaluated against the hypothesis to confirm it. At this point in time two results are possible: (a) the hypothesis is confirmed and thereby we proceed to the next step, or (b) a redesign of the approach is needed, including a possible redefinition of the hypothesis.

6. **Presentation of the research results.** In this final phase, once the results are collected, they are shared with the scientific community to validate them, and to gather feedback from peers and experts in the field. If needed, correction actions will be taken.

1.4 Structure of the document

The rest of this dissertation is structured as follows. The first part of Chapter 2 introduces the background concepts about the differences between control data and non-control data and gives an overview of the security landscape of both control-data attacks and defences as well as non-control-data attacks and defences. The second part of the chapter presents three different reviews: a review of the threat models used in system security research, a review the control-flow integrity techniques and a review of the data-only attacks and defences. Chapter 3 explores the specific threat models that IoT devices incur and the applicability of control-data and non-control-data attacks and defences in these systems. In Chapter 4 we present our optimised data-flow integrity implementation and validate its completeness. Finally, Chapter 5 summarises the results of this dissertation, and defines future research opportunities.

As if any of this makes sense.

Suzanne Palmer, Driving the Deep

CHAPTER

2

Background and related work

Attackers have evolved their exploitation methods from techniques that relied on code injection to more sophisticated code-reuse attacks because the defence techniques have become more refined. The first part of this chapter gives an overview of the different static analysis concepts required to understand the security landscape from the perspective of control-data and non-control-data attacks. The second part of the chapter consists of three reviews of the state of the art: security threat models, control flow integrity methods, and non-control-data attacks and defence techniques.

The remainder of this chapter is organised as follows. First the background concepts of this dissertation are presented in Section 2.1, including the differences between the two types of data that can be found in a computer program: *control data* and *non-control data*, which is outlined in Section 2.1.1, Section 2.1.2 briefly explains the notation of control flow graphs, Section 2.1.3 gives an overview of the different precision types available for points-to analysis algorithms, Section 2.1.4 describes the different types of control-data at-

tacks and defences, and Section 2.1.5 presents the variety of non-control-data attacks and defences. Once the background concepts are presented, Section 2.2 reviews the threat models presented in the systems security field in the last 20 years that are relevant for this dissertation. Section 2.3 reviews the control flow integrity methods for user and kernel space, Section 2.4 reviews the different non-control-data attack variants, and Section 2.5 reviews the two different trends in defences against non-control-data attacks.

Finally Section 2.6 summarises the contents of this chapter.

2.1 Background concepts

2.1.1 Control data and non-control data

The data that computer programs handle can be classified into control data or non-control data. On the one hand, control data is defined as the data used by a program in order to manage its control flow; for instance, the return addresses of function calls, function pointers, addresses of `jmp` instructions and indirect call targets are control data.

On the other hand, non-control data is all the remaining data being used in a program, which does not intervene in the control flow of the program. This non-control data is still considered security-critical (Chen et al., 2005) since its modification could have malicious repercussions. Non-control data consists of user input data (e.g., strings provided in an input form), configuration data (e.g., paths to storage folders), user identity data (e.g., Linux `uid`s and `guid`s) and decision-making data (e.g., boolean variables).

Although the quantity of non-control data in a program is more abundant, attackers have historically exploited memory errors targeting control data; consequently, both the industry and academia have been more focused on preventing attacks targeting control data; however, in recent years non-control-data attacks have become more common (US-CERT, 2014; Wu, Nicolas, 2023).

2.1.2 Control flow graphs

Control flow graphs (CFG) (Allen, 1970) are a graphical representation of the control flow of a program using a graph notation. This representation is commonly used in the compiler and static analysis field s.

These control flow graphs are defined with a directed graph $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$ made of nodes and edges, where $\mathscr{V}$ represents the set of *basic blocks* (BB) of a program (nodes). These basic blocks are a linear sequence of instructions with an entry point and an exit point. $\mathscr{E}$ represents the set of directed edges of the control flow graph. An edge $e = (b_1, b_2) \mid e \in \mathscr{E}$, represents a pair of nodes which indicate that a control flow transfer happens from basic block b_1 to basic block b_2.

In Figure 2.1 a sample `C` function is shown with its equivalent CFG on the right side. Note that this CFG only covers the `main()` function and the called `rand()` function is not covered. Control flow graphs that cover the full range of functions of a given program are named *interprocedural control flow graphs* (ICFG).

The nodes of the CFG can have a sequence of instructions in source code, assembly code (e.g., see Figure 2.2) or three address code representation depending on the usage that the CFG will have, but all representations are equivalent.

2.1.3 Precision of points-to analysis algorithms

Points-to analysis is a static analysis technique utilised to determine to which memory locations pointers can point to. The precision of the points-to analysis can be defined by different properties: (i) flow sensitivity, (ii) context sensitivity and (iii) field sensitivity.

- **Flow sensitive and flow insensitive points-to analysis.** In this case the precision of the points-to analysis is defined by its capability to take control flow changes into account in order to determine to which memory location a pointer may point to.

```
int main()
{
   int i = rand();
   int a;
   if (i)
     a = 0;
   else
     a = i + rand();
   a += rand();
   return a;
 }
```

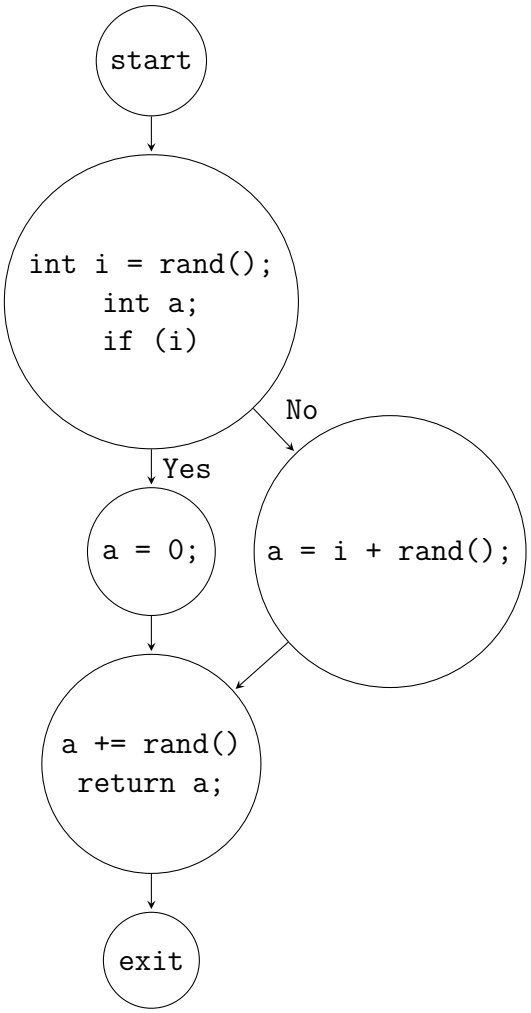


Figure 2.1: Example C program with its equivalent CFG.

```
1 |        a = &x;
2 |        a = &y
```

Using the code example above, depending on the *flow sensitivity* of the algorithm the points-to analysis may conclude the `a` can point to the address locations of `x` and `y` if it is a flow-insensitive algorithm, or that `a` points to the address `x` exactly at line 1, and that it points to the address `y` at line 2.

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
callq  rand
mov    %eax,-0x4(%rbp)
cmpl   $0x0,-0x4(%rbp)
je     400545
movl   $0x0,-0x8(%rbp)
jmp    400554
callq  rand
mov    %eax,%edx
mov    -0x4(%rbp),%eax
add    %edx,%eax
mov    %eax,-0x8(%rbp)
callq  rand
add    %eax,-0x8(%rbp)
mov    -0x8(%rbp),%eax
leaveq
retq
```

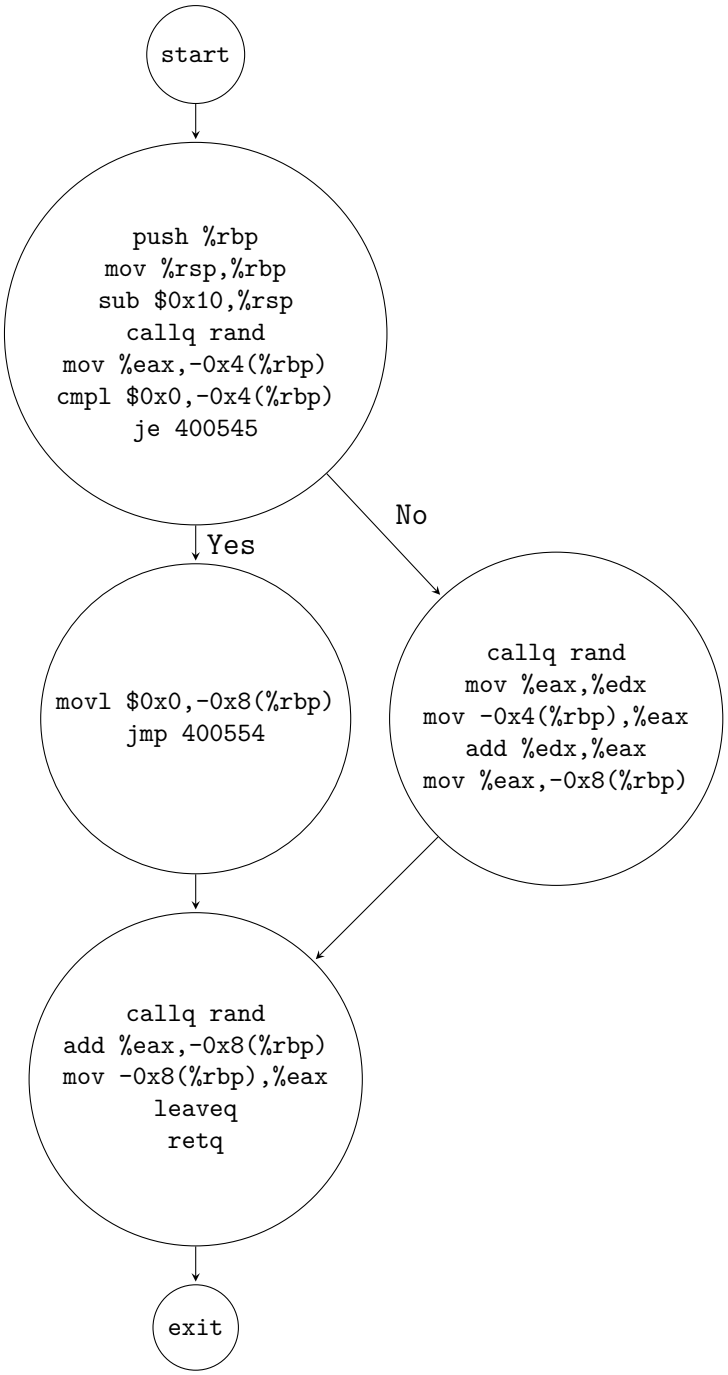


Figure 2.2: Example program in x86_64 with its equivalent CFG.

- **Context sensitive and context insensitive points-to analysis.** In this type of analysis the points-to information needs to distinguish the context between function calls.

The following figure shows a high-level call graph in which the function-call relationships of functions A, B, C, D and E are shown.

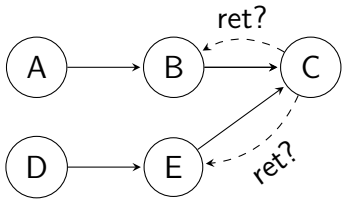


Figure 2.3: Example call graph for functions A, B, C, D and E.

If pointers previously defined are used in function `C` a context-sensitive approach would be able to determine if they were defined in the `B->C` call-site context or in the `E->C` call-site context.

- **Field sensitive and field insensitive points-to analysis.** This type of analysis mainly deals with the granularity of the points-to analysis with regards to structures or objects.

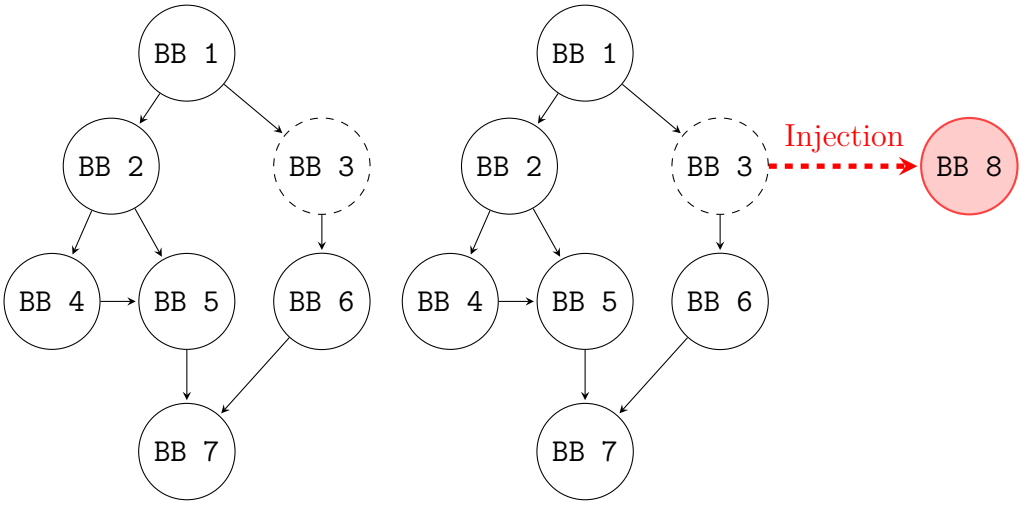
```
struct object {
  int field_a;
  int field_b;
};

struct object x = {
  .field_a = 0,
  .field_b = 1
};

int *y = &x.field_b;
```

Code 2.1: Field sensitive and field insensitive points-to analysis example.

Given the example shown above, a field insensitive algorithm will not be able to determine the offset in which `.field_b` is located based on the starting memory location of `object`, but a field sensitive approach will be capable of doing so.



(a) Original CFG **(b)** CFG with a code injection

Figure 2.4: Code injection attack. (a) shows the original CFG of the program with a memory error on BB 3. (b) shows the CFG modification after the code injection attack to execute BB 8, the dashed line shows the modification in the original CFG.

2.1.4 Control-data attacks and defences

Control-data attacks or control-flow hijacking attacks are those attacks which subvert the intended control flow of given a program for a malicious one. In this section we present the different control-data attacks and the available defences.

2.1.4.1 Code injection

The first known control-data attacks relied on code injection. In these attacks the attackers would exploit a memory error to inject malicious shellcode, then they would forcefully redirect the program counter into the injected code, executing the malicious shellcode.

Figure 2.4 shows the DFG of a sample program that has a memory error in basic block 3 that can be exploited. The attacker would use the memory error to write the malicious shellcode and then redirect the execution of the program to the malicious code (e.g., BB 8 shown in Figure 2.4).

2.1.4.2 Defences against code injection attacks

Code injection attacks were suppressed with the inclusion of Write XOR Execute (W⊕E) and Data Execution Prevention (DEP) (Andersen and Abella, 2004), which allowed to mark page tables as writable or executable, but not both at the same time, which prevented the execution of previously injected data. Stack canaries (Cowan et al., 1998; Dang et al., 2015) also played a role hindering these attacks, which caused runtime errors when data was written beyond the allowed bounds of a data buffer.

Since code injection was rendered virtually impossible, code reuse attacks surfaced.

2.1.4.3 Code reuse

Code reuse attacks leveraged a memory error to redirect the control flow of the program to existing pieces of code in the executable section, which were executed with malicious intent. In the initial forms of this type of attacks, attackers redirected the execution of a program to the virtual addresses of security critical functions already loaded in memory; for instance, in the `return-into-libc` (Nergal, 2001) attack, attackers could redirect the control flow of the program to a fixed function address from the address space of `libc`, a library that is commonly found already loaded in memory due to the popularity of the `C` programming language; if they used the address of the `system` function, attackers could execute any shell command (lib).

In order to hinder these types of attacks which relied on the location of the virtual addresses of well-known fixed library functions, Address Space Layout Randomization (ASLR) (Bhatkar et al., 2003; Team, 2003), and later on kernel Address Space Layout Randomization (Giuffrida et al., 2012) (KASLR) were implemented.

All ASLR-based defences rely upon the randomization of the locations of memory areas, user space ASLR ranzomizes the base addresses of the locations of the stack and heap memory areas, as well as the locations of the loaded libraries. These ASLR-based defences are known as statistical defences because their security is closely tied to the number of available bits (Shacham

et al., 2004), which makes ASLR-based defences limited on 32-bit systems. Furthermore, if attackers are able to deduce information about the privileged memory space layout (Hund et al., 2013) ASLR-based defences can also be bypassed, as well as by leveraging just-in-time code-reuse attacks (Snow et al., 2013).

Code reuse attacks became more sophisticated after the introduction of return-oriented programming (ROP) (Shacham, Hovav, 2007), which instead of redirecting the control flow of the program to well-known functions, reused existing small sequences of instructions ending in a `ret`, named *gadgets*. These gadgets were searched within the existing program scope using automated tools in order to find and emulate operations that can be commonly found in real-world programs, such as loading constants, loading and storing data from memory, performing arithmetic operations etc. with the resulting range of operations being Turing-complete (Tran et al., 2011).

This variety of operations comes from the realisation that in order to construct ROP chains the instructions do not need to appear in the program as we require them to be, since we can start reading the instructions from whichever byte position we wish. For instance:

```
f7 c7 07 00 00 00    test $0x00000007, %edi
0f 95 45 c3          setnzb -61(\%ebp)
```

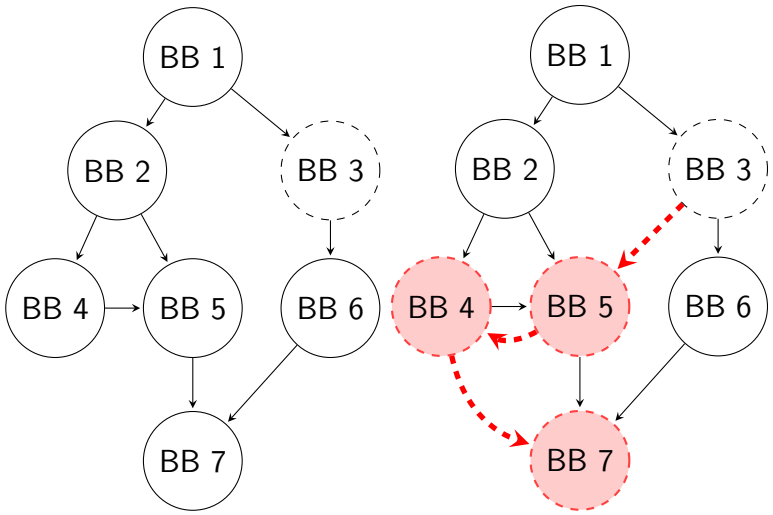
Code 2.2: Sample x86 instructions as it might appear in a program, opcode bytes on the left side and instructions on the right side.

can be used as a ROP gadget since we have a `c3` (`ret`) instruction in the code, but we need to start reading the instructions from a different byte:

These gadgets were then chained together, which allowed to perform arbitrary computations. Figure 2.5 shows the CFG graph of a sample program with a memory error in basic block 3. As opposed to injecting a new basic block with new code, code reuse attacks first identified these sequences of gadgets (in the example in basic blocks 4, 5 and 7), to then force the execution of the program from unintended code paths, as shown in Figure 2.5.

```
c7 07 00 00 00 0f   movl $0x0f000000, (%edi)
95                  xchg %ebp, %eax
45                  inc \%ebp
c3                  ret
```

Code 2.3: Re-arranged sample x86 instructions by reading one opcode byte forward on the left side, with the resulting instructions on the right side.



(a) Original CFG **(b)** CFG with gadgets

Figure 2.5: Code-reuse attack. (a) shows the original CFG of the program with a memory error on BB 3. (b) shows the modified CFG with three gadgets, BB 4, BB 5 and BB 7. The dashed lines show the modifications to the original CFG.

Since the inception of ROP attacks, more variants have surfaced throughout the years that use different mechanisms to chain together pieces of code that allow to perform arbitrary computations, both in user (Bletsch et al., 2011; Checkoway et al., 2010; Bosman and Bos, 2014; Schuster et al., 2015; Checkoway et al., 2010; Bosman and Bos, 2014), as well as in kernel space (Hund et al., 2009; Zeng et al., 2023).

2.1.4.4 Defences against code reuse attacks

Even thought the conjunction of W⊕E/DEP along with K/ASLR reduced the attack surface of code reuse attacks, these code reuse attacks are still a reality. In recent years both the academia and the industry have been developing new defence techniques based on the control-flow integrity (CFI) (Abadi et al., 2005) property.

Control-flow integrity is a complex runtime defence that on its more fine-grained approach can theoretically prevent all types of control-flow hijacking attacks on its own.

Although CFI is one of the most complete defences against control-flow hijacking attacks, it has a non-negligible performance cost (21%) (Abadi et al., 2005); moreover, full backward edge protection requires the implementation of a shadow stack to track the context of the function calls, which results in additional overhead (~16%) (Dang et al., 2015; Abadi et al., 2005).

Section 2.3 covers a more thorough review of the different control flow integrity implementations, outlining their strengths and weaknesses.

2.1.5 Non-control-data attacks and defences

An orthogonal problem to control-flow hijacking attacks are non-control-data attacks, which do not subvert the intended control flow of a program, and thereby remain undetected against state-of-the-art defences such as CFI or CPI (Kuznetsov et al., 2014).

2.1.5.1 Non-control-data attacks

Efforts to tamper with non-control data to launch viable attacks were first discussed by Chen et al. (Chen et al., 2005) who identified different types of security-critical non-control data and provided memory error examples of real-world programs that could be exploited to launch data-only attacks. Both Hu et al. and Ispoglou et al. independently demonstrated that non-control-data attacks could be automatically constructed using two different methods (Hu et al., 2015; Ispoglou et al., 2018), the former method stitches together two

or more existing dataflows given a program with a memory error, the variable it can initially influence, its trace, and the target variable to modify; whereas the later provides a domain specific language to write exploits and leverages symbolic execution techniques to search the basic blocks with the desired capabilities. A subsequent contribution by Hu et al. (Hu et al., 2016) proved that data-only attacks exhibited Turing-complete capabilities if atomic data-oriented gadgets and a gadget dispatcher could be chained together in a technique known as data-oriented programming (DOP), which can be thought as return-oriented programming (ROP) (Shacham, Hovav, 2007) or other of its variants or weird machines (Checkoway et al., 2010; Bletsch et al., 2011; Bosman and Bos, 2014; Schuster et al., 2015), but specifically tailored for the non-control-data plane.

Exploitation based on non-control data has diversified, and has become a stage to launching attacks that give the attacker more manoeuvrability. Small changes to non-control data have proven to be useful to subvert other integral defence techniques such as control-flow integrity (CFI) (Carlini et al., 2015) thereby allowing any form of control-flow hijacking; they can also disable `W⊕E` in the Linux kernel (Davi et al., 2017), bypass same-origin-policy enforcement in Chrome (Jia et al., 2016) or enable remote code execution in Mozilla's JavaScript code engine (Park et al., 2020).

Section 2.4 covers a more thorough review of the different non-control-data attacks.

2.1.5.2 **Non-control data defences**

Specialised solutions have been proposed to hinder specific attacks that leverage data-only attacks, Song et al. (Song et al., 2016a) utilised DFI and write integrity testing (WIT) (Akritidis et al., 2008) to prevent memory-corruption-based privilege escalation attacks in the Android Linux kernel, Davi et al. (Davi et al., 2017) deployed page table randomisation to prevent data-only attacks against page tables, whereas other approaches focus on providing memory safety (Song et al., 2016b; Carr and Payer, 2017; Frassetto

et al., 2018; Proskurin et al., 2020) instead by using different methods for selective memory isolation.

Data-flow integrity (DFI) (Castro et al., 2006) is another promising defence that can protect against non-control-data attacks. DFI is a runtime defence that has some design similarities to CFI, DFI also works in two phases, a static analysis phase uses reaching definitions to first compute the uses and definitions of variables and then a runtime phase performs the security checks. Due to the amount of extra read and write operations required by a fine-grained DFI implementation, the overhead of this defence can reach an average of 104% (Castro et al., 2006), which is not an overhead tolerable by many commercial software applications

Another trend proposed several statistical defences (Belleville et al., 2018; Bhatkar and Sekar, 2008; Davi et al., 2017). For instance, Data Space Randomization (DSR) (Bhatkar and Sekar, 2008) is one of the first defences to be proposed to hinder non-control-data attacks. DSR is a statistical defence against non-control-data attacks that relies on modifying the representation of data objects in memory by encrypting values in memory with a xor operation. The downsides of DSR are that similarly to other statistical defences, DSR is also vulnerable to information leak attacks that could help determine the masks used in the cryptography operations.

To the best of our knowledge none of the defences that protect against non-control-data attacks have been implemented outside academic circles.

Section 2.5 covers a more thorough review of the different available defences against data-only attacks.

2.1.6 **Background concepts summary**

In this section we have presented a general overview of the concepts needed to understand the differences between control data and non-control-data (Section 2.1.1), the control flow graph terminology (Section 2.1.2) and the differences between the different points-to analysis types (Section 2.1.3)). Moreover we have also presented a high-level overview of the different control-data at-

tacks and defences (Section 2.1.4) as well as the non-control-data attacks and defences (Section 2.1.5).

2.2 Threat models

The OWASP foundation (Drake, 2024) defines *threat modeling* as the process of capturing, organising and analysing the information that affects the security of an application in a structured representation. Microsoft (Microsoft, 2024) however, defines threat modeling as an engineering technique that allows to identify attacks, threats, vulnerabilities and countermeasures that may affect security applications.

In system security research, threat models are presented as agreeable theoretical models in which the system to be defended or attacked is defined in terms of which defence techniques it has deployed, which security shortcomings it presents and which assumptions are made. This allows to define the adversary model that a theoretical security defence system would face to model the different types of attacks that it would be able to sustain; similarly, when a new attack technique or an improvement to a existing attack technique is being presented the threat model would list the adversarial capabilities and the defensive assumptions that the attack relies on being deployed.

In this section we review the threat models available (if any) in selected systems security research works which are relevant for this dissertation (e.g. control flow hijacking attacks and defences as well as data-only attacks and defences, with related work) over the last 20 years (2005-2024) in order to identify the common trends in the assumptions made in the threat models.

Section 2.2.1 identifies the adversary models of defence techniques for user and kernel space, and Section 2.2.2 identifies the defensive assumptions of attack techniques against user and kernel space; finally, Section 2.2.3 presents our conclusions.

2.2.1 Adversary models for defence techniques in user and kernel space

When identifying the threat model of a defence technique the threat modeling focuses on identifying the attack model that the system to be defended would be facing, and the minimum set of defensive techniques that the system would need in order to withstand the attack.

In this scenario, when a novel defence technique is being presented, there are several observations that need to be considered: (i) the default defences being deployed in the system, (ii) the capacity or incapacity of the novel defence technique to coexist with the default defences without reducing and/or interfering with the defensive capabilities of the system, (iii) the minimum set of defences that the system needs to withstand an attack, (iv) the security defects that the system would still exhibit that could be exploited by an adversary and, (v) any peculiarities of the deployment of the system that might interfere with existing defences or boost attack capabilities.

Tables 2.1 and 2.2 list the common assumptions that novel user space and kernel space defensive techniques employ, respectively. In these reviewed cases observations (i) through (iv) mentioned above are taken into account, but observation (v) i.e. *"any peculiarities of the deployment of the system that might interfere with existing defences or boost attack capabilities"*, is not mentioned and it is assumed to be taken into account by security engineers when the system is deployed.

In the threat models of the reviewed user space defences (see Table 2.1), the baseline is at least W⊕E/DEP protection being deployed and that the adversary has some sort of arbitrary read and write capabilities. W⊕E/DEP is also commonly identified as the default defence technique available in the majority of the systems, ASLR and some type of CFI defences (whether partial CFI or full CFI) are also commonly identified as default defence techniques, but with less prevalence than W⊕E/DEP.

The reviewed user space defence threat models do not mention any incompatibilities with existing default defence techniques, however, even if they do not consider some default defences in their threat model (such as ASLR) this

Table 2.1: Attack models and defensive assumptions in the threat models of user space defences. The top part of the table specifies the defence capabilities and the bottom part of the table the attack capabilities. *Mem. corr.: R+W* refers to memory corruption vulnerability allowing to read and write arbitrary memory.

Assumption	**Works**
W⊕E/DEP	(Abadi et al., 2005), (Belleville et al., 2018), (Carr and Payer, 2017), (Cheng et al., 2014), (Frassetto et al., 2017), (Frassetto et al., 2018), (Kuznetsov et al., 2014), (Mohan et al., 2015), (Niu and Tan, 2015), (Zhang et al., 2016), (Zhang et al., 2015), (Zhang et al., 2013), (Zhang and Sekar, 2013)
No ASLR	(Cheng et al., 2014)
ASLR	(Frassetto et al., 2017), (Zhang et al., 2015), (Zhang et al., 2013), (Zhang and Sekar, 2013)
No side channel attacks	(Belleville et al., 2018)
Side channel attacks out of scope	(Carr and Payer, 2017)
Data-only attacks out of scope	(Zhang et al., 2016), (Zhang et al., 2015)
No flaws in hardware	(Belleville et al., 2018)
No physical access	(Belleville et al., 2018)
CFI	(Akritidis et al., 2008), (Frassetto et al., 2017), (Frassetto et al., 2018), (Zhang and Sekar, 2013)
Partial CFI	(Zhang et al., 2016), (Zhang et al., 2015)
CPI	(Frassetto et al., 2018)
Secure boot	(Frassetto et al., 2017)
OS, compiler and instrumentation are TCB	(Carr and Payer, 2017), (Kuznetsov et al., 2014), (Niu and Tan, 2015)
Safe registers	(Bounov et al., 2016)
Safe stack	(Bounov et al., 2016)
Mem. corr.: R+W	(Belleville et al., 2018), (Carr and Payer, 2017), (Kuznetsov et al., 2014), (Mohan et al., 2015), (Niu and Tan, 2014b), (Niu and Tan, 2015), (Zhang and Sekar, 2013)
Mem. corr.: R+W with memory access permissions	(Frassetto et al., 2018), (Zhang et al., 2016),(Zhang et al., 2015)
Mem. corr.: R+W to known addresses	(Frassetto et al., 2017)
Mem. corr.: limited R	(Zhang et al., 2013)
Corruptible heap	(Bounov et al., 2016)

is commonly because these defences are considered orthogonal to the security problem to be solved, and thereby do not serve to gain any additional security guarantees against the type of attack that the specific defence technique is trying to protect against, for instance, CFI cannot withstand non-control-data attacks, and such defence technique, even though nowadays it is becoming more common in some operating systems to protect user-space applications, is not useful in that scenario and it is strictly irrelevant to be included in such adversary model. Nevertheless, orthogonal security defences are often added to the threat model to avoid broadening the attack capabilities in the case that some defences are successfully disabled. For instance, since non-control-data attacks are considered difficult to craft, they are often used just to disable control-flow hijacking defences, such as CFI, and then less complex attacks can be used once CFI is disabled.

Related to the security defects that the system would have, the majority of the reviewed works assume some kind of memory corruption error that could grant read and write capabilities to the end user. These read and write capabilities range from full arbitrary read and write access to restrictions to which addresses the attacker is allowed to write to.

Regarding the threat models for kernel space defences (see Table 2.2), due to the reduced scope of existing attack types against Linux operating systems and their derivatives, and the attacks being very specialised, the range of assumptions made in the defensive side has less variety. There is a smaller range of available defences, and those that exist are more complex that their user space counterparts if any exist. The most common assumptions rely on some kind of secure boot mechanism in the system, attackers having no physical access to the system, and W⊕E/DEP protections. In this case W⊕E/DEP protections are also commonly identified as the default defence technique, as well as to some extent, secure boot.

Table 2.2: Attack model and defensive assumptions in the threat models of kernel space defences. The top part of the table specifies the defence capabilities and the bottom part of the table the attack capabilities. *Mem. corruption* refers to memory corruption vulnerability allowing to: *W* write arbitrary memory, *R+W* read and write arbitrary memory, or *limited W* write to memory with some limitation.

Assumption	**Works**
W⊕E/DEP	(Davi et al., 2017), (Ge et al., 2016), (Proskurin et al., 2020)
KASLR	(Davi et al., 2017), (Proskurin et al., 2020)
SMAP/SMEP	(Proskurin et al., 2020)
DMA protection	(Davi et al., 2017)
Side channel attacks out of scope	(Davi et al., 2017)
Data-only attacks out of scope	(Ge et al., 2016)
User space pages not accessible when CPU is in kernel mode	(Davi et al., 2017)
No physical access	(Criswell et al., 2014), (Ge et al., 2016)
CFI	(Davi et al., 2017), (Proskurin et al., 2020)
CPI	(Davi et al., 2017)
Shadow stack	(Proskurin et al., 2020)
Secure boot	(Criswell et al., 2014), (Davi et al., 2017), (Ge et al., 2016), (Song et al., 2016a)
Mem. corruption: W	(Ge et al., 2016), (Criswell et al., 2014)
Mem. corruption: limited W	(Kemerlis et al., 2012)
Mem. corruption: R+W	(Davi et al., 2017), (Proskurin et al., 2020), (Song et al., 2016a)
Full user space control	(Davi et al., 2017)

2.2.2 Defensive assumptions for attack techniques in user and kernel space

Presenting a new attack, or a refined version of a previously existing attack type requires slightly different observations to the threat model than those assumptions made for threat models of defensive techniques previously made (see Section 2.2.1): (i) the default defence techniques deployed in the system,

Table 2.3: Defensive assumptions in the threat models of user space attacks. The top part of the table specifies the defence capabilities and the bottom part of the table the attack capabilities.

Defensive assumptions	**Works**
W⊕E/DEP	(Bletsch et al., 2011), (Carlini et al., 2015), (Carlini and Wagner, 2014). (Checkoway et al., 2010), (Chen et al., 2005), (Conti et al., 2015), (Göktaş et al., 2014a), (Johannesmeyer et al., 2024), (Ispoglou et al., 2018), (Rogowski et al., 2017), (Snow et al., 2013)
ASLR	(Conti et al., 2015), (Göktaş et al., 2014a), (Johannesmeyer et al., 2024), (Ispoglou et al., 2018), (Rogowski et al., 2017), (Snow et al., 2013)
Limited ASLR	(Carlini and Wagner, 2014)
CFI	(Carlini et al., 2015), (Johannesmeyer et al., 2024), (Ispoglou et al., 2018)
Limited CFI	(Checkoway et al., 2010), (Davi et al., 2014), (Göktaş et al., 2014a)
Shadow stack	(Conti et al., 2015), (Ispoglou et al., 2018)
Huge library loaded	(Bletsch et al., 2011), (Davi et al., 2014)
W capability	(Carlini et al., 2015), (Morton et al., 2018)
R+W capability	(Conti et al., 2015), (Ispoglou et al., 2018), (Rogowski et al., 2017)
Limited R capability	(Morton et al., 2018), (Schuster et al., 2015)
Control flow can be subverted without `rets`	(Checkoway et al., 2010)

(ii) the quality of the deployed defence techniques, (iii) the ability of the attack to subvert existing defences, (iv) the security defects needed for the attack to work and, (v) any peculiarities of the deployment of the system that might assist the attacker.

Tables 2.3 and 2.4 list the common assumptions made in the works that we have reviewed for user space and kernel space attacks. Similarly to the assumptions made in the adversary models for defensive techniques (see Sec-

Table 2.4: Defensive assumptions in the threat models of kernel space attacks. The top part of the table specifies the defence capabilities and the bottom part of the table the attack capabilities.

Defensive assumptions	**Works**
W⊕E/DEP	(Hund et al., 2013), (Zhou et al., 2024)
KASLR	(Hund et al., 2013), (Zeng et al., 2023), (Zhou et al., 2024)
SMEP/SMAP	(Hund et al., 2013), (Zeng et al., 2023), (Zhou et al., 2024)
Kernel CFI	(Han et al., 2024), (Zhou et al., 2024)
Hypervisor protection	(Han et al., 2024)
Page level write protection	(Han et al., 2024)
Read-only safe area	(Han et al., 2024)
KPTI	(Zeng et al., 2023)
Page table randomization	(Zeng et al., 2023)
Kernel mode code cannot be run	(Hund et al., 2013)
Kernel mem. R+W	(Han et al., 2024)
Kernel mem. limited W	(Zhou et al., 2024)
Control-flow hijacking allowed	(Zeng et al., 2023)
Kernel address leakage	(Zeng et al., 2023)

tion 2.2.1) these threat models also forgo to mention the peculiarities (if any) of the deployment of the system (observation v), but they usually rely on real-world defence implementations rather than relying on theoretical models, since the quality of the implementation of the defensive technique (observation ii) often plays a central role in the capacity of the attack to bypass a defence, because the perfect implementation of some defensive techniques is often impractical in the real world, and thereby trade offs have to be made which benefit the attacker. For instance, CFI has two core security characteristics, its capacity to protect forward control-flow transfers, and its capacity to protect backward control-flow transfers. The former is easier to implement and several real-world implementations exist already, however, the later is computationally expensive and incurs in high performance overheads, thereby

some threat models (see Table 2.3) rely on the system deploying *limited CFI* rather than CFI, understood as theoretically perfect CFI, because it is not yet feasible.

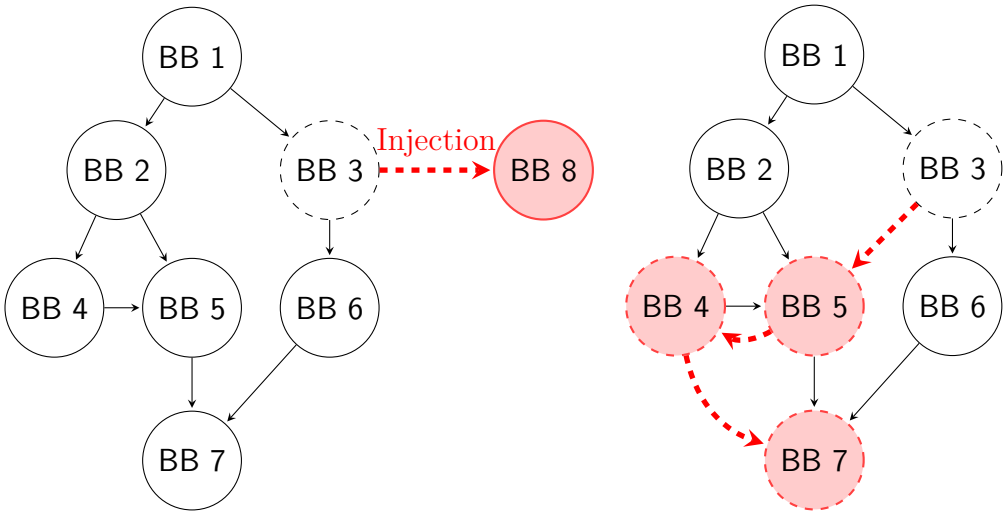
2.2.3 Conclusions

The threat models describing defensive techniques often rely on attackers having strong capabilities, such as arbitrary read and write, in order to better showcase the benefits of the novel defensive technique. Some threat models also include defences that are not applicable to the type of attack that they need to protect against, to clarify that they do not interfere with existing defence techniques.

In the case of threat models to describe new or improved attack techniques, researchers either assume perfect defence techniques that can be bypassed by their new types of attacks, or rely on the existing flaws of real-world techniques to argue that improvements or refinements are still needed in these existing techniques in order to prevent already existing types of attacks.

During the threat model process, the majority of the works do not mention which are the computational capabilities of the system, if any special security needs are needed for the system to be considered secure, however, in some cases, the requirements of no physical access to the system are given, whereas for the vast majority of threat models no mention to the deployment model is given and to the best of our knowledge the system is assumed to be part of a cloud deployment or a data center for historical reasons.

As we will discuss in Chapter 3, knowing the threat model that specific types of systems will face helps understand which are the core security techniques needed to protect systems in each scenario, as well as showcase if certain types of attacks are feasible, practical, or not, against these types of deployments.



(a) Code injection attack **(b)** Code-reuse attack

Figure 2.6: Control-data attacks subverting the original control flow transfers of a program.

2.3 A review of control flow integrity methods for user and kernel space

Attackers have evolved classic *code-injection attacks*, such as those caused by buffer overflows to sophisticated Turing-complete *code-reuse* attacks. Control-Flow Integrity (CFI) is a defence mechanism to eliminate control-flow hijacking attacks caused by common memory errors. CFI relies on static analysis for the creation of a program's control-flow graph (CFG), then at runtime CFI ensures that the program follows only the legitimate path. Thereby, when an attacker tries to execute malicious shellcode, CFI detects an unintended path and aborts execution. CFI heavily relies on static analysis for the accurate generation of the control-flow graph, and its security depends on how strictly the CFG is generated and enforced. This section reviews the CFI schemes proposed over the last ten years and assesses their security guarantees against advanced exploitation techniques.

Operating systems must ensure that both their own code and the code of their applications remain incorruptible and consequently secure and reli-

able against attackers. Since code-injection attacks are widely known, adversaries nowadays commonly exploit memory corruption bugs to subvert the *control flow* of the operating system or the applications being executed (see Figure 2.6).

Rather than focusing on protecting the integrity of code, with complete memory safety or developing safe dialects of `C/C++`, modern defences try to protect the *control flow integrity* (CFI) of these systems.

Since CFI (Abadi et al., 2005) was introduced to avoid these problems and issues, different implementations and versions of these techniques have been proposed by the community that try to make it practical, while ensuring the completeness of its protection. In addition, new attacks have been proposed that limit the effectiveness of these methods. Due to the raising relevance of CFI methods for the system security community, in this work we present the first comprehensive literature review and discussion of control flow integrity defences and the attacks that try to subvert them.

2.3.1 The rationale of Control flow integrity

`C/C++` code goes hand in hand with memory corruption bugs, which allow an adversary to launch attacks that exploit those memory errors. *Code injection* attacks due to stack-based or heap-based overflows, dangling pointers/use-after-free, and format string vulnerabilities are common, and can be prevented using defences such as write-xor-execute (W⊕E) / Data Execution Prevention (DEP) (Andersen and Abella, 2004) and stack canaries (Cowan et al., 1998) which are included in modern compilers and operating systems. Nevertheless *code-reuse* attacks, like return-into-libc (Nergal, 2001), return-oriented programming (ROP) (Shacham, Hovav, 2007), jump-oriented programming (JOP) (Checkoway et al., 2010; Bletsch et al., 2011) and call-oriented programming (COP) (Carlini and Wagner, 2014) still can not be fully prevented. Operating systems themselves are not exempt from code-reuse attacks, such as return-to-user (ret2usr) (Kemerlis et al., 2012), a kernel level variant of return-into-libc, and sigreturn oriented programing (SROP) (Bosman and Bos, 2014), which exploits the signal handling capabilities of UNIX like systems to deploy

gadgets in the same manner that ROP and JOP do with `ret` and `jmp` instructions respectively.

Operating systems deploy statistical defences to protect user space and kernel space against code-reuse attacks; namely address space layout randomisation (ASLR) (Team, 2003), and Kernel ASLR (Giuffrida et al., 2012). However, these defences can be circumvented due to information leakage and just-in-time code-reuse attacks both for user (Snow et al., 2013) and kernel (Hund et al., 2013) space.

Taking into account these problems, Abadi et al. introduced *control flow integrity* (CFI) (Abadi et al., 2005), a defence mechanism to prevent code-reuse attacks, which try to subvert the legitimate execution flow of a program.

2.3.2 Technical overview of Control flow integrity

CFI works in two phases, firstly, it computes the Control Flow Graph (CFG) of the program by static analysis, either using its source code or its binary; afterwards, during program execution, CFI enforces that the program follows through the legitimate execution path; otherwise the program is aborted.

In the computation phase, CFI is concerned with *points-to analysis*, the static analysis that deals with the possible values of a pointer, because it affects the precision in which a CFG is generated (Burow et al., 2017) and consequently, the precision in which the enforcement phase will guarantee the legitimate execution path. Taking into account the precision, CFI implementations can be categorised into (i) *flow-sensitive* or *flow-insensitive* and (ii) *context-sensitive* or *context-insensitive*. On the one hand, flow-sensitive algorithms use the control flow information of a program to determine the possible values of a pointer, whereas flow-insensitive algorithms compute a set of values that are valid for all program inputs (Hind, 2001; Hardekopf and Lin, 2009). On the other hand, context-sensitive algorithms take into account the context when analysing a function, preventing values from propagating to impracticable paths and thus guaranteeing that the context of a call remains independent from other call contexts; in contrast, context-insensitive algo-

rithms allow a function to return to the computed set of all callers (Wilson et al., 1995; Hind, 2001).

In the enforcement phase, CFI may take into account *forward* (e.g. indirect calls or jumps) and *backward* (e.g. return instructions) control-flow transfers. CFI solutions that provide just a forward enforcement of control-flow transfers have been found insecure (Göktaş et al., 2014a; Carlini and Wagner, 2014; Davi et al., 2014), whereas solutions that enforce the backward transfers usually rely on a shadow stack, a structure that holds copies of the return addresses present in each of the stack frames of the original stack, causing up to a **~ 10%** increase in the program overhead (Dang et al., 2015), or use the last-branch record registers (LBR) (Intel, 2016c; AMD, 2013) which are only available to a subset of CPUs and have a limited storing capacity. Processor vendors have started to implement frameworks to support CFI, such as Intel's Control Flow Enforcement Technology (CET) (Intel, 2016a), which provides specific instructions (e.g. `RSTORSSP`, `SAVESSP`) to manage shadow stacks and track control transfers.

2.3.3 Control flow integrity implementations

The vast majority of CFI implementations aim to protect the user space, and come in the flavours of compiler extensions, source code or binary code patching frameworks and kernel modules, whereas a small subset intend to secure the operating system deploying kernel modifications or new kernel modules.

2.3.3.1 User space implementations

The original CFI (Abadi et al., 2005) operates on x86 binaries by machine-code rewriting. For the forward control-flow transfers, the rewriting process includes an ID insertion at each destination, and an ID-check before each source; then at runtime the source ID and the destination ID must coincide. To ensure that a function call returns to the appropriate call site, namely, that a backward control-flow transfer is secure, the implementation uses a shadow call stack relying on x86's segmentation capabilities. CFI requires:

1. The code to be non-writable, to prevent attackers from rewriting the ID-check.

2. The data to be non-executable, to prevent attackers to execute data generated with the expected ID.

The first requirement is true in modern OSes, excluding the loading time of dynamic libraries and runtime code-generation, and the second requirement is enforced with W⊕E. This implementation makes the assumption that two destinations are equivalent if they are called from the same source, thus introducing imprecision in the CFG and thereby in the enforcement phase.

- **MoCFI.** MoCFI (Davi et al., 2012) provides CFI protection on iOS devices' applications running on ARM processors. It addresses the special issues of ARM architecture (e.g. the nonexistence of dedicated return instructions). As the original CFI, it also operates on binaries. The authors generate a CFG of the application and a patchfile containing metadata of the indirect branches and function calls in the application; dynamic libraries used in the application are not protected. In the runtime enforcement phase, the patchfile is used by the MoCFI shared library, generating a patched application which is executed within the CFI policy. MoCFI uses a shadow stack to protect calls and returns. For the forward control-flow transfers however, it cannot protect indirect jumps/calls whose destination cannot be identified on the static analysis; thereby they can target any valid address within the function, or any valid function respectively.

- **CCFIR.** Unlike MoCFI, CCFIR (Zhang et al., 2013) is a binary CFI implementation that includes protection for libraries. CCFIR works on Windows x86 PE executables, with partial support for libraries. It builds upon Abadi et al.'s approach and incorporates a third new ID-check for returns to sensitive and non-sensitive functions. This implementation suffers from the same imprecision as Abadi et al.'s for forward edges and introduces it in backward edges.

- **Bin-CFI.** Bin-CFI (Zhang and Sekar, 2013) is another binary implementation that protects stripped Linux x86 binaries including shared libraries. This approach is similar to the original CFI scheme and has lower security guarantees than CCFIR. Recent studies have found both Bin-CFI and CCFIR protections insufficient (Göktaş et al., 2014a; Davi et al., 2014), since grouping destinations into equivalence classes is not strong enough to prevent them for being used as ROP/JOP gadgets.

- **kBouncer**. kBouncer (Pappas et al., 2013) is a hardware based Windows toolkit that relies on Intel Nehalem architecture's LBR registers to retrieve the sequence of the latest 16 indirect branch instructions at critical points (e.g. system calls). In total, kBouncer protects the execution of 52 Windows API functions.

- **ROPecker.** ROPecker (Cheng et al., 2014) is a Linux x86 kernel module that utilises the LBR register to prevent code-reuse attacks. Both kBouncer and ROPecker depend on chain length and gadget length heuristics to prevent such attacks. Nevertheless they can be bypassed by choosing the right sized gadget-chain length (Carlini and Wagner, 2014; Davi et al., 2014; Göktaş et al., 2014b).

- **O-CFI.** O-CFI (Mohan et al., 2015) is a binary based x86/x86_64 CFI implementation that combines code randomisation with CFI checking. O-CFI first computes the permissible destination addresses for each indirect branch, then it transforms the policy that indirect branches must reach to a valid destination into a bounds-checking problem; thereby O-CFI has to check that the destination address exists within min/max address boundaries. These boundaries are protected using code randomisation and then checked making use of Intel's memory protection extensions (MPX) (Intel, 2016b). O-CFI uses a relaxed version of forward and backward control-flow transfer checks and consequently, can be bypassed.

All the previously presented binary level approaches (Abadi et al., 2005; Davi et al., 2012; Zhang et al., 2013; Zhang and Sekar, 2013; Pappas et al.,

2013; Mohan et al., 2015) fail to capture *complete* context sensitivity; whereas just some of them support partial (backward) context sensitivity due to the use of a shadow stack (Abadi et al., 2005; Davi et al., 2012).

- **PathArmor.** PathArmor (van der Veen et al., 2015) is the first binary level scheme to tackle context sensitivity for forward and backward edges. Context-sensitive CFI methods need to keep track of the paths of the executed control-flow transfers, to later on enforce that the execution follows the legitimate path. Instead of using a shadow stack, PathArmor employs LBR registers to emulate a path monitoring mechanism limited by the number of LBR registers (just 16). PathArmor outperforms all previous protection schemes for forward edge transfers. However, shadow stack based approaches are still more reliable for backward edges due to the limitations that current hardware imposes.

The following CFI implementations utilise source code to perform their static analysis:

- **VTV/IFCC.** Tice et al. (Tice et al., 2014) present two different forward-edge protection mechanisms integrated in production compilers, Virtual-Table Verification (VTV) and Indirect Function-Call Checks (IFCC) for GCC and LLVM respectively. Stack based attacks have been found effective bypassing VTV/IFCC (Conti et al., 2015) and the subsequent compilers have been patched.

- **SafeDispatch.** SafeDispatch (Jang et al., 2014) is a earlier LLVM compiler extension, and like VTV, aims to protect virtual tables (vtables) for `C++` virtual calls; both VTV and SafeDispatch fail to provide full control-flow protection since they focus just on forward edges

Further research has been done with the objective of protecting vtables, resulting in two binary level implementations, VfGuard (Prakash et al., 2015) and VTint (Zhang et al., 2015); which unfortunately are also limited to partial control-flow protection.

One of the recent variants of code-reuse attacks, Counterfeit Object-oriented Programming (COOP) (Schuster et al., 2015) can generate Turing-complete attacks using gadgets of `C++` virtual functions. COOP is effective against the original CFI, bin-CFI, CCFIR, VTint and partially against IFCC, VfGuard and PathArmor. In contrast, COOP can be prevented at binary level by TypeArmor (van der Veen et al., 2016), and at source code level with the compiler extensions SafeDispatch, VTV, VTrust (Zhang et al., 2016) and VTI (Bounov et al., 2016).

- **MCFI and RockJIT.** Niu and Tan introduced Modular CFI (MCFI) (Niu and Tan, 2014a), a new scheme which extends CFI with modular compilation. Building upon MCFI the authors present RockJIT (Niu and Tan, 2014b) which enforces CFI in Just-In-Time compilers; both MCFI and RockJIT induce some imprecision in the edge generation since they apply the same assumption as the original CFI for equivalent targets.

- **πCFI**. πCFI (per-input CFI) (Niu and Tan, 2015) is Niu and Tan's follow-up contribution, which introduces the highest security guarantees for a source code based CFI solution. πCFI differs from all previous CFI implementations in the way it addresses the CFG generation. Conservative CFI implementations utilise static analysis to compute the CFG before the enforcement phase, this analysis is considered hard since it has to take into account *all* the possible input values for the given program; moreover, CFI's security guarantees are strictly bounded to the CFG's precision. Niu and Tan point out than even if a perfect CFG were possible, it would still include unnecessary edges for a given input. Thereby they tackle the CFG generation in the following way; firstly, they generate the conservative CFG for all program inputs (building upon MCFI and RockJIT), then during program execution, given an input, πCFI generates CFG edges on the fly, but just those which comply with the conservative all-input CFG are enforced. This innovative scheme provides less backward edge protection compared to shadow

Table 2.5: Comparison of user space CFI implementations by precision. *B* stands for binary, *S* source-code, *KM* kernel module, *VMM* virtual machine monitor; *CS* context-sensitive, *EC* equivalent classes, *H* heuristics, *HW* hardware, ∅ the policy is not enforced.

		Precision	
	Scheme	**Forward**	**Backward**
Original CFI	B	EC	CS
MoCFI	B	EC	CS
CCFIR	B	EC	EC
Bin-CFI	B	EC	EC
kBouncer	B	H	H
ROPecker	KM	H	H
O-CFI	B	EC	EC
PathArmor	B	Hardware limited CS	Hardware limited CS
VTV	S	CS	∅
IFCC	S	CS	∅
SafeDispatch	S	CS	∅
TypeArmor	B	EC	∅
MCFI	S	EC	EC
RockJIT	S	EC	EC
πCFI	S	Limited CS	Limited CS

stack approaches, but higher guarantees that other backward edge approaches. Concerning forward edges, πCFI has stronger assurance that original CFI due to the per-input mechanism.

2.3.3.2 Kernel-space implementations

The community has focused these last years on securing the user space with different CFI implementations. However, due to the critical relevance of protecting the kernel space, some approaches have been presented in the last few years.

- **SBCFI.** State-based CFI (SBCFI) (Petroni Jr and Hicks, 2007) is a CFI implementation for Xen and VMware Workstation virtual machine

monitors. Unlike CFI enforced in userland, kernel space CFI is not able to comply to every requirement. Firstly, the generated CFG cannot be guaranteed to be read only. Second, since an attacker that has gained access to the kernel is obviously capable to access or modify page tables, there is no possible manner to confirm that the data is still non-executable. Thereby, SBCFI enforces a relaxed CFI by periodically checking the current kernel's CFG against the initial kernel's CFG. This implementation provides light security guarantees since it does not enforce backward edges and the support for forward edges is limited.

- **Hypersafe** (Wang and Jiang, 2010) is a LLVM framework extension that targets hypervisors. Hypersafe introduces the concepts of *non-bypassable memory lockdown* and *restricted pointer indexing* to introduce CFI on hypervisors. The former method is responsible of guaranteeing the integrity of the hypervisor's code and static data; the later delimits the contents of the targets of the control data (function pointers and return addresses) into a target table, to then rewrite each function pointer or return address to a pointer index to the target table. Using the restricted pointer indexing, Hypersafe can either allow light security guarantees by allowing a function to return to any address entry on the target table, or a more strict scheme, by generating a target table for each function and allowing the function to return to a *subset* of all returns, made specifically for that function. Hypersafe implements backward edge enforcement policies but not as safe as those provided by shadow stack schemes, and for forward edges, in its strict scheme, a policy more accurate than the orignal CFI but less that the most strict user space implementations (PathArmor and πCFI).

- **kGuard** (Kemerlis et al., 2012) is a GCC compiler extension whose aim is to protect the kernel against ret2usr attacks. kGuard combines CFI with program shepherding. *Program shepherding* (Kiriansky et al., 2002) is a technique that permits to implement arbitrary restrictions to code origins and control-flow transfers. Upon compiling a kernel

with kGuard, Control Flow Assertions (CFA) are introduced before each control-flow transfer. These assertions are comparable to the original CFI checks, but unlike them, CFAs are not checked against a CFG to enforce a valid edge, they just ensure that the target address exists within kernel space instead. This security mechanism cannot withstand ROP/JOP like attacks since it is comparable to just enforcing weak forward control-flow transfers, like the traditional CFI, and also a weak policy for backward transfers.

- **KCoFI** (Criswell et al., 2014) provides CFI for commodity OSs utilising the infrastructure provided by the Secure Virtual Architecture (SVA) (Criswell et al., 2007) virtual machine. This infrastructure is used to handle low level operations regarding the MMU, general I/O, signal dispatch and context switching. KCoFI is built on top of the SVA virtual machine, and thereby requires the OS and applications to instrument to be compiled to the virtual instruction set provided by the SVA architecture. As the original CFI, KCoFI enforces a CFI policy that is not context-sensitive.

- Ge et al. (Ge et al., 2016) propose a semi-automated CFI implementation for kernel software. For the static CFG generation the authors utilise static taint analysis to compute the targets of indirect calls, making the following two assumptions: (i) the operations of function pointers are limited to dereferencing for calls and assignment, this is, an assigned function pointer will not be modified, and (ii) data pointers to function pointers (e.g. `void *` to `int (*function)(int)`) do not exist. To compute return targets, and to handle the special cases of assembly functions that may present a tail-call optimisation or fall-through functions, Ge et al. use intra-procedural control-flow analysis to find the callees of a caller. They emulate the execution of each function in a recursive fashion until a return instruction or another call is found. Finally, the enforcement of the generated CFG is based on Hypersafe's restricted pointer indexing.

Table 2.6: Comparison of kernel space CFI implementations by precision. *B* stands for binary, *S* source-code, *KM* kernel module, *VMM* virtual machine monitor; *CS* context-sensitive, *EC* equivalent classes, *H* heuristics, *HW* hardware, ∃ exists, ∅ the policy is not enforced.

		Precision	
	Scheme	**Forward**	**Backward**
SBCFI	VMM	CFG comparison	∅
Hypersafe	S	Limited CS	Limited CS
kGuard	S	∃ in kernel space	∃ in kernel space
KCoFI	S	EC	EC
Ge et al.	S	Limited CS	EC

2.3.4 Control flow integrity methods: a discussion

Tables 2.5, 2.6, 2.7 and 2.8 summarise the implementations reviewed in this section from an implementation precision and known attacks perspective for both kernel and user space.

Regarding user space CFI implementations, on the one hand, binary schemes are more common, nevertheless these schemes are known to be less secure than their source-code based counterparts. On the other hand, some source code schemes (VTV, IFCC, SafeDispatch, TypeArmor) tend to focus on just forward edges, and thereby are prone to ROP attacks; while others enforce policies that fall into the *equivalent classes* paradigm which just can partially prevent ROP/JOP.

Concerning kernel space, the implementations enforce modified CFI methods due to the peculiarities that protecting a kernel involves. The majority of them are source code based approaches. Hypersafe is the strongest implementation followed by the implementation of Ge et al. The former provides limited context-sensitivity for both edges, whereas the later falls into the equivalent classes paradigm for the backward edges. Both implementations are based on the technique of restricted pointer indexing.

In summary, the strongest implementations provide some level of context-sensitivity for both edges. PathArmor utilises the LBR registers for a hard-

Table 2.7: Comparison of user space CFI implementations by known attacks.

User space CFI implementation	**Known attacks**
Original CFI	COOP
MoCFI	Limited JOP
CCFIR	(Göktaş et al., 2014a), COOP
Bin-CFI	(Davi et al., 2014; Göktaş et al., 2014a), COOP
kBouncer	(Davi et al., 2014; Göktaş et al., 2014b; Carlini and Wagner, 2014)
ROPecker	(Davi et al., 2014; Göktaş et al., 2014b; Carlini and Wagner, 2014)
O-CFI	Theoretical ROP/JOP
PathArmor	History flush
VTV	COOP
IFCC	ROP
SafeDispatch	ROP
TypeArmor	ROP
MCFI	Theoretical ROP/JOP
RockJIT	Theoretical ROP/JOP
πCFI	Limited ROP/JOP

Table 2.8: Comparison of kernel space CFI implementations by known attacks.

Kernel space CFI implementation	**Known attacks**
SBCFI	ROP
Hypersafe	Limited ROP/JOP
kGuard	ROP/JOP
KCoFI	Theoretical ROP/JOP
Ge et al.	Theoretical ROP/JOP

ware limited context-sensitivity, πCFI builds upon the modular CFG idea and Hypersafe uses restricted pointer indexing to provide a limited context-sensitivity.

Future trends of work are focusing on more precise context-sensitive schemes and addressing the implementation of safer CFI schemes for commodity OSs. On the one hand, having a full context sensitivity for backward edges still remains an open problem due to the high performance cost of the shadow stack, and the incomplete security of hardware assisted methods. On the other hand, work trends are showing an specialisation on the forward control-flow transfer enforcements with methods that perform at their best for forward edges and do not implement any security policy for backward control-flows.

2.4 A review of non-control-data attacks

Systems security researchers have been focusing on developing mitigation and protection mechanisms against *code-injection* and *code-reuse* attacks. Modern defences focus on protecting the legitimate control-flow of a program, nevertheless they cannot withstand a more subtle type of attack, *non-control-data* attacks, since they follow the legitimate control flow, and thus leave no trace.

Programs are formed by control data (e.g. return addresses, function pointers) and non-control data (e.g. variables, constants). Even though non-control data is more abundant, both attackers and defenders have focused their efforts into exploiting or protecting control data. Different attacks have been repeatedly applied to the control flow, such as complex *code-injection* and *code-reuse* attacks. Therefore, memory integrity methods have been proposed in order to secure operating systems and userland programs (e.g. safe dialects of `C/C++`, secure virtual architectures, and control-flow integrity methods).

Even though non-control-data attacks are not new (Chen et al., 2005) and their importance has not decreased, modern operating systems and their programs remain vulnerable against these type of attacks.

2.4.1 Logical grounds for non-control-data attacks

The most common memory corruption vulnerabilities, namely *code-injection* and *code-reuse* attacks, have been tackled by the community, creating defences for commodity operating systems and compilers. On the one hand,

stack canaries (Cowan et al., 1998) and write-xor-execute (W⊕E)/DEP (Andersen and Abella, 2004) defence schemes try to prevent *code-injection* attacks resulting from stack, heap or buffer overflows, use-after-free and format string vulnerabilities, whereas Control-Flow Integrity (CFI) (Abadi et al., 2005) and program shepherding (Kiriansky et al., 2002), along with the statistical defences provided by ASLR (Team, 2003) and Kernel ASLR (Giuffrida et al., 2012) concentrate on *code-reuse* attacks arising from classic return-oriented programming (ROP) (Nergal, 2001; Shacham, Hovav, 2007), or its newer variants (Checkoway et al., 2010; Bletsch et al., 2011; Bosman and Bos, 2014; Carlini and Wagner, 2014).

Due to the attention given to code-injection and code-reuse attacks, Chen et al. (Chen et al., 2005) raised awareness of the *pure data* or *non-control-data* attacks, since all the previous approaches (Cowan et al., 1998; Andersen and Abella, 2004; Abadi et al., 2005; Kiriansky et al., 2002; Team, 2003; Giuffrida et al., 2012) focus just on *control-data*, and thereby cannot endure more subtle non-control-data attacks.

A non-control-data attack differs from a control-data attack because it does not affect the control-flow of a program. Control-data attacks are based on rewriting control data (e.g. return addresses), leaving a trace in the form of an unintended control-flow transfer. This transfer can be detected and prevented at runtime. On the contrary, non-control-data attacks follow legitimate control-flow transfers since they are based on modifying the program's logic or decision-making data. Consequently, they remain invisible to defence techniques which only focus on control data.

2.4.2 Security critical non-control-data targets for non-control-data attacks

Chen et al.'s work (Chen et al., 2005) identifies the following types of security-critical data that may be subject to non-control-data attacks:

- **Configuration data**. Many applications, such as web servers, need configuration files in order to define access control policies and file path

directives to specify the location of trusted executables. If an attacker was capable of overwriting such configuration data, it would be possible to launch unintended applications (e.g. root shells), and moreover bypass the access controls of the web server.

- **User input data**. A well known practice in software engineering is to distrust user input data, and only after that data has been validated it can be used. If an attacker could change the input data after the validation process, she could execute the program with a malicious input.

- **User identity data**. UIDs and GIDs are stored in memory while authentication routines are executed. If such IDs were tampered with, an attacker could impersonate a user with administration privileges.

- **Decision-making data**. Boolean values (e.g. authenticated or not) are usually used to make decisions in an application, an attacker could change those decision making values to redirect the flow of a program through unintended branches.

Moreover, Hu et al. (Hu et al., 2015) enhanced the previously presented types of security-critical data with the following items:

- **Passwords and private keys**. The disclosure of passwords and private keys could give an attacker full access to a system.

- **Randomised values**. Tags for CFI enforcement, random canary words and randomised addresses are used in many security related mechanisms (e.g. CFI, ASLR, SSP). If an attacker knows the random canaries placed in the stack she can use stack-smashing attacks without alerting the Stack Smashing Protector (SSP) (Cowan et al., 1998).

- **System call parameters**. Tampering with the parameters of security critical system calls (e.g. `execve`, `setuid`) can lead to privilege escalation or unintended program execution.

```
struct passwd {
  uid_t pw_uid;
  ...
} *pw;
...
int uid = getuid();
pw->pw_uid = uid;
...
// memory error that allows to control pw->pw_uid
...
void passive(void) {
  ...
  // set root uid
  seteuid(0);
  ...
  // set non-root uid
  seteuid(pw->pw_uid);
}
```

Figure 2.7: Vulnerable code from the `wu-ftpd` web server. Taken from (Hu et al., 2015).

If an adversary abuses any type of the above mentioned types of non-control data due to a memory error, a non-control-data attack can be launched. For instance, Figure 2.7 shows a vulnerable piece of code from the `wu-ftpd` daemon adapted from (Chen et al., 2005). In this example a non-control-data attack can be crafted in the `getdatasock` function due to an exploitable format string error present in the vulnerable function `vuln`, which allows an attacker to control the value of `pw->pw_uid`. In a non-malicious use case of the `getdatasock` function, the example shows how the `wu-ftpd` daemon calls the `vuln` function to temporary store the user ID into the `pw->pw_uid` variable, to then perform privileged operations by switching to root (`uid` 0). Once the privileged operations have been completed, the daemon drops the privileges back using the cached value at `pw->pw_uid`, and then resumes normal operation.

However, an attacker could use the memory error present at the `vuln` function to set the value that they wish to `pw->pw_uid`, for instance 0; in

this way the `seteuid` call is rendered useless because `pw->pw_uid` no longer caches the user ID and the privileges are not dropped, granting root privileges to the attacker without subverting the intended control-flow of the program, and thereby, bypassing control-flow guarding defences.

Modern web browsers are a common target (Jia et al., 2016; Rogowski et al., 2017) for such attacks; in (Jia et al., 2016) for instance, Jia *et al.* demonstrate that a data-only attack targeting a couple of bytes can disable the process based isolation policy of modern web browsers such as Google Chrome, allowing to bypass the same origin policy. Once this policy has been disabled attackers are free to mount arbitrary cross-origin reads and writes (e.g., reading the contents of any file or modifying DOM elements), or access the persistent browser storage to manipulate web sessions and cookies.

This pattern of using non-control-data attacks as the first stage of the exploitation process has also been used in kernel space (Davi et al., 2017; Zhou et al., 2024). Davi *et al.* (Davi et al., 2017) prove that a data-only attack can be used to change the permissions of a memory page holding a security critical function to writable (e.g., the `sys_setns` (set) system call, which allows to change the namespace of a process); then use the write permissions to write shellcode into the function, change the permissions of the page back to read+execute, and finally execute the function, which would allow a privilege escalation and leave the system vulnerable.

Another real world non-control-data attack is the famous OpenSSL *Heartbleed* vulnerability (US-CERT, 2014). Heartbleed allowed a remote attacker to expose sensitive data, namely *private keys*, using a non-control-data attack. On the OpenSSL 1.0.1 and 1.0.2β heartbeat request/response protocol, an attacker could request a heartbeat using a legitimate payload but with a payload length field larger than the real payload (up to 65,535 bytes). The heartbeat protocol would read this request and craft a response copying the original payload in a buffer, this buffer was allocated of the size indicated by the payload length field (controlled by the attacker). Since the payload length field was not correctly verified against the length of the real payload, a memory leakage was possible because the attacker could specify the number

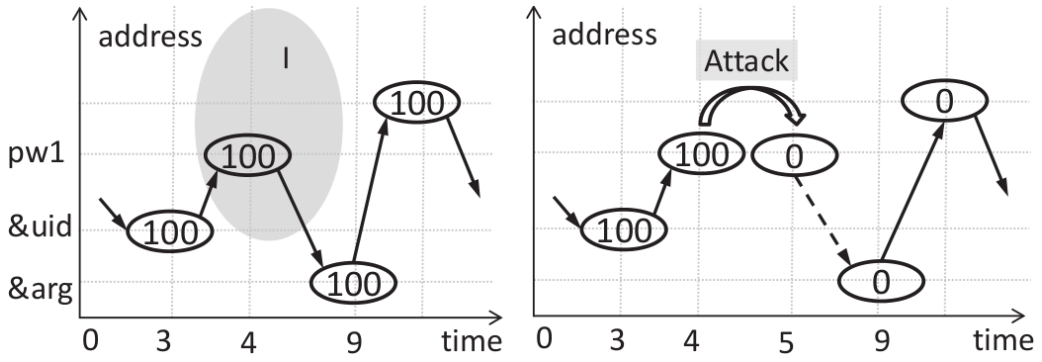


Figure 2.8: Original 2D-DFG of the vulnerable code of `wu-ftpd` web server (left) and resulting 2D-DFG after a non-control-data attack (right). `&arg` is the stack address of `setuid`'s argument. Taken from (Hu et al., 2015).

of bytes that the protocol would return, and thus giving the ability to leak private keys.

These attacks have remained invisible to the aforementioned defence techniques that focus on control data, since both attacks leveraged *non-control data* and do not tamper with the legitimate control-flow of the program.

2.4.3 Non-control-data attack variants

Some argue that the limitations of non-control-data attacks (i.e., their inability by definition to subvert the control-flow of a program) signified that they are commonly limited to exploits targeting information leak or privilege escalation, however, as we have shown in an example in the previous section, this is not a limitation, since data-only attacks can also be used as a stepping stone for more complex attacks.

Moreover, the following sections show how data-only attack can be automatically constructed (Section 2.4.3.1), and that they can perform Turing-complete computations (Section 2.4.3.2).

2.4.3.1 **Data-flow stitching**

Hu et al. (Hu et al., 2015) demonstrated that non-control-data attacks can be automatically constructed using a technique named *data-flow stitching*. Given a memory error as an input, this technique is capable of redirecting the data-flow of the given program to the intended target as to tamper with security critical data (e.g. UUIDs) or to leak sensitive data (e.g. private keys) by *stitching* together several data-flows under the influence of the memory error.

Hu et al. introduced the concept of *two-dimensional data-flow graph* (2D-DFG) in order to represent the existing data dependencies in a program which was executed with a concrete input. A 2D-DFG is a directed graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ the set of edges. A vertex v ($v \in \mathcal{V}$) is a variable instance with a value, and it is represented as a point (a, t) in the two dimensions of the 2D-DFG, addresses and time; thereby, a refers to the address or register name of the variable, and t to the execution time when the variable instance is created. A *vertex* $v = (a, t)$ is created when an instruction writes to memory value a at time t; a *data edge* (v', v) is created when the instruction takes v' as the source and v as the destination operands, and finally, an *address edge* (v', v) is created when an instruction uses v' as the address of the operand v.

In order to generate a non-control-data attack, data-flow stitching requires a program with a memory error. The set of memory locations this memory error can write to are called the *influence* I. The new data-flow that wants to be created consists of two vertices, namely source vertex (v_s), and target vertex (v_t); resulting in a data-flow path from v_s to v_t. This new data-flow path would have a new 2D-DFG $\mathcal{G}' = \{\mathcal{V}', \mathcal{E}'\}$, where $\mathcal{V}'$ and $\mathcal{E}'$ are generated by the memory error exploit. The goal of data-flow stitching is to discover a data-flow edge set $\overline{\mathcal{E}}$, where $\overline{\mathcal{E}} = \mathcal{E}' - \mathcal{E}$, to add in the resulting 2D-DFG of the memory error ($\mathcal{G}'$), allowing new data-flow paths from v_s to v_t.

Following the examples given by Chen et al. and Hu et al., Figure 2.7 shows a vulnerable piece of code from the `wu-ftpd` web server, and Figure 2.8 (left) shows the original 2D-DFG of such program. The format string vulnerability on `wu-ftpd` server can overwrite `pw->pw_uid`, since such vertex is under the influence of the memory error; thereby, a privilege escalation attack is possible

using a non-control-data attack generated by data-flow stitching that inserts an edge in the DFG, overwriting `pw->pw_uid` with root's UID, as shown in Figure 2.8 (right).

This example requires only the addition of a single edge in the new 2D-DFG, nevertheless data-flow stitching can also generate advanced attacks that need stitching more edges.

2.4.3.2 **Data-oriented programming**

Data-oriented programming (DOP) is a technique proposed by Hu et al. (Hu et al., 2016) to perform computations on a program's memory respecting its legitimate control-flow, it is the first technique that demonstrated that non-control-data attacks could be automatically constructed. DOP's computations are based on non-control-data attacks resulting from memory errors and have been proven to be Turing-complete. DOP is comparable to the computations made using gadgets on code-reuse attacks by ROP (Shacham, Hovav, 2007), JOP (Checkoway et al., 2010; Bletsch et al., 2011), COP (Carlini and Wagner, 2014) and sigreturn-oriented programming (Bosman and Bos, 2014); however, unlike all the previous approaches, DOP is based on *data-oriented gadgets* that have a small number of differences compared to classic gadgets.

DOP requires the use of (i) *data-oriented gadgets* and (ii) a *gadget dispatcher*. On the one hand, data-oriented gadgets form the virtual instructions (i.e. arithmetic, logical, assignment, load, store, jump and conditional jump) required to simulate a Turing machine. Data-oriented gadgets can simulate these operations using the x86 instruction set the same way ROP, JOP and COP do. In contrast, data-oriented gadgets need to use just memory and not memory or registers to generate its operations. In addition, data-oriented gadgets must follow the legitimate control flow. On the contrary, one of the benefits of data-oriented gadgets is that they can be scattered and consequently, there is no need for them to be executed one after the other.

On the other hand, the gadget dispatcher allows the chaining of the data-oriented gadgets and simulating control operations. This dispatcher allows an attacker to choose the sequence of data-oriented gadgets (e.g. creating loops),

resulting in *interactive* and *non-interactive* DOP attacks. Interactive attacks use loops and at every loop iteration, a selector controlled by the memory error selects the sequence of data-oriented gadgets that must be executed. Non-interactive attacks require a single payload where all the data-oriented gadgets must be chained.

Other works, such as (Ispoglou et al., 2018) and (Johannesmeyer et al., 2024) further demonstrate that automatic data-only attacks are a reality.

2.5 A review of the two defence trends against non-control-data attacks

The two current trends to protect users against non-control-data attacks follow very distinguishable paths, on the one hand the data-flow integrity property tries to detect malicious changes in data-flow paths to prevent attacks. On the other hand, the second type of defences either rely on randomisation schemes or in data obfuscation to prevent non-control-data attacks.

2.5.1 Data-flow integrity techniques

Castro et al. introduced *data-flow integrity* (DFI) (Castro et al., 2006), a defensive technique that aims to protect programs against non-control-data attacks. DFI targets x86 architecture ensuring that a given program's data stays within the permitted data-flow paths. Firstly, DFI generates the data-flow graph (DFG) of the program by static analysis, secondly, it instruments the program introducing data-flow integrity checks, and finally, it enforces at runtime that the data-flow of the program is allowed by the DFG, otherwise the execution is aborted.

DFI relies on *reaching definitions analysis* (Aho et al., 2006) for the static DFG generation. Reaching definitions analysis is a static analysis technique used by modern compilers to deploy global code optimisation (e.g. dead code elimination), based on data-flow analysis. Data-flow analysis tries to extract information about the flow of data from program execution paths (Aho et al.,

2006), and reaching definitions analysis concretely deals with the definition (i.e. assignment) and use (i.e. read) of variables. Using reaching definitions analysis, DFI can compute a DFG that contains a set of definitions, assigns an identifier to each definition, and maps those identifiers to instructions. In this way, the DFG shows the instructions that assigned a value to each used variable.

DFI uses two different static analyses to generate reaching definitions, (i) a *flow-sensitive intra-procedural* analysis and (ii) a *context-insensitive inter-procedural* analysis. The former is used to compute the reaching definitions of variables that have no definition outside the function in which they are declared, whereas the later computes the reaching definitions of variables with definitions outside the function in which they are declared. This separation is done to increase the performance of the analysis, since flow-insensitive algorithms have less computing overhead.

Once the static DFG has been generated, DFI instruments at runtime the program to check before every variable use that its definition is within the statically generated reaching definitions identifiers. If not, the data-flow integrity property does not hold, and the program must be terminated. Castro et al.'s approach makes use of a *runtime definitions table* (RDT) to keep track of the last definition of each identifier. In order to check if the data-flow integrity holds, the last value of the RDT for a given identifier must be checked against the static DFG.

In order to be effective, DFI itself must remain safe against sabotages, requiring (i) the integrity of the RDT, (ii) the integrity of the code, and (iii) the integrity of DFI's instrumentation. RDT integrity is achieved ensuring that the definitions are within the memory boundaries defined for the RDT, code integrity is accomplished using modern operating systems' W⊕E check on pagination. The integrity of the instrumentation performed by DFI can be ensured relying on DFI (e.g. instrumenting uses and definitions of control-data made by the compiler) or using additional defences, such as CFI (Abadi et al., 2005) and program shepherding (Kiriansky et al., 2002).

2.5.1.1 **Kernel data-flow integrity**

The operating system is the first line of defence against attacks based on memory corruption on userland applications. However the OS itself is not safe against non-control-data attacks. If an attacker were capable of successfully gaining control of the OS, all the defences deployed in userland applications would become futile.

Song et al. (Song et al., 2016a) utilise DFI in order to enforce kernel security invariants related to access control mechanisms against memory-corruption attacks in a system named KENALI. This system protects two security invariants, (i) *complete mediation*, attackers have to be prevented from bypassing access control checks, and (ii) *tamper proof*, the integrity of the data and code of the reference monitors must be maintained.

As to enforce these invariants KENALI uses two techniques: *InferDist* and *ProtectDists*. InferDist is used to distinguish the *distinguished regions*, which are the regions that have essential security critical data for enforcing the security invariants. ProtectDists enforces DFI over these regions and due to invariant (i) complete mediation (ensuring the integrated of the data and the code), CFI must also be enforced.

Furthermore, InferDist uses the kernel CFI mechanisms proposed by Criswell et al. (Criswell et al., 2014) to protect control-data. Regarding non-control data, KENALI enforces that if a security check fails, it will return the `-EACCESS` error code (permission denied). InferDist retrieves these error codes and, through dependency analysis on the conditional variables of the security checks, is able to discover which are the distinguished regions.

Finally, to enforce the DFI over the inference result regions, KENALI distinguishes three types of data-flow (i) within non-distinguishing regions, (ii) between two different types of regions and (iii) within distinguishing regions. KENALI protects the distinguishing regions using a two-layer scheme. The first layer is a lightweight data-flow isolation mechanism to protect the second type of data-flow (between two different types of regions), and a more heavy DFI enforcement mechanism when the two regions are distinguishing.

2.5.2 Diversity and obfuscation techniques

Diversity-based defences are those relying on some sort of randomisation to make harder for an attacker to locate the values of security critical data, such as return addresses, code pointers, variables etc. Compared to DFI diversity is a statistical defence which is deployed at an earlier stage, thereby if diversity is bypassed, DFI-based defences would be applied. Due to the statistical nature of diversity it is highly unlikely for the same attack to work twice in the same program, either the same attack to work across machines.

Defences based on diversity can be applied in a broad range of granularity levels (Forrest et al., 1997; Larsen et al., 2014): instruction, basic block, loop, function, program or system, or at a combination of those levels. The randomisation can happen in different stages, resulting in schemes that generate the diversity at implementation, compilation, load, link, installation, execution, update time or in various phases.

Obfuscation is the method in which a program is transformed in a way that it becomes difficult to understand while at the same way, the original semantic is maintained (Bhatkar et al., 2003). Obfuscation based methods modify a program with the objective of making the exploitation of a memory error harder. Diversity and obfuscation are commonly mixed to implement defences.

Methods for diversity or obfuscation can be categorised into (Larsen et al., 2014): static data randomisation, constant blinding (e.g. XORing immediate values), structure layout randomisation, heap layout randomisation and library entry point randomisation.

In this section we review the state of the techniques that either deploy some kind of obfuscation or diversity methods at program granularity level, concretely, those aiming to diversify the data of a program, in order to protect non-control data.

2.5.2.1 Diversity-based techniques

- **DieHard** (Berger and Zorn, 2006) deploys a custom memory manager to provide probabilistic memory safety in the heap randomising the lo-

cation of the variables. Using its *replica* mode, different replicas of the same program are run using a distinct random seed, when a variable is accessed the resulting value is checked against all replicas, if the values are not the same DieHard will terminate the program. Using DieHard, attacks relying on relative addresses on the heap are avoided.

- **DieHarder** (Novark and Berger, 2010) is an enhanced version of DieHard based on OpenBSD's memory allocator. Unlike, DieHard, which allocates large continuous memory regions, DieHarder's allocator uses a *sparse page layout* scheme, which allows small objects to be allocated in a heap that is disseminated randomly among multiple pages. DieHarder's randomisation increases the available entropy of systems with low ASLR entropy, allocating large virtual address spaces and randomising the address of the heap objects. Finally, DieHarder fills the freed objects with random data. Thereby, DieHarder has stronger security guarantees than its predecessor.

2.5.2.2 Obfuscation-based techniques

- **PointGuard** (Cowan et al., 2003) is a GCC compiler extension to protect pointers. PointGuard encrypts the values of the pointers when they are in memory and decrypts them before they are loaded into registers. The encryption/decryption scheme is done XORing a random key generated when the process starts. When an attacker overwrites a pointer, PointGuard applies the pointer decryption upon its dereference, XORing the malicious pointer value with the secret key, this operation results in an access to a random position in memory that very likely causes the program to crash. PointGuard can thus prevent attacks that rely on absolute addresses.

- **Data Randomisation** (DR) (Cadar et al., 2008) is another technique that uses encryption to protect data. Using points-to-analysis (Heintze and Tardieu, 2001), DR divides instruction operands into *equivalence classes* (two operands belong to the same equivalence class if they may

refer to the same object in a memory safe operation), these equivalence classes have assigned a random mask which is used to encrypt the values and decrypt them. This scheme ensures the integrity of the data written and read from memory.

- **Data Space Randomisation** (DSR) (Bhatkar and Sekar, 2008), as it name suggests, works by randomising the data of a program. DSR is a statistical defence against non-control-data attacks that relies on modifying the representation of data objects in memory. DSR encrypts values in memory with a xor operation before each write, and decrypts them before each read. These extra operations required before each read and write cause an average performance overhead of 15%. DSR is an improvement of PointGuard, it uses the same XORing mechanism but, unlike PointGuard, DSR can also protect non-pointer data. DSR is compiler extension to generate a source code to source code transformation.

  DSR requires all the modules to be compiled in DSR-mode, meaning that all external libraries need to be compiled in the same way, making code sharing difficult.

- `librando` (Homescu et al., 2013), a diversification library for Just-in-time (JIT) compilers. `librando` decompiles the code generated by the JIT compiler and then uses constant blinding to protect operands from unauthorised reads and writes. `librando` does not use the common XOR scheme to encrypt the operands, instead the following scheme is followed: $value = \mod_{32}(encrypted + key)$, where the key is a randomly generated value.

2.5.2.3 Mixed techniques

- **Address Obfuscation** (AO) (Bhatkar et al., 2003) is a technique to obfuscate the absolute locations of code and data, and the relative distances between data items. AO randomises the base address of the stack and heap segments, the starting address of dynamic libraries and

the locations of the functions and the static data. Moreover, AO permutes the order of locals in stack frames, static variables, functions in the executable or in shared libraries. In addition to randomisation, AO injects some random padding in stack frames, between variables in the static segment, between the memory allocated by `malloc` and between instructions in functions (with the corresponding jump instructions to avoid breaking the program).

Using AO, attacks that rely on absolute addresses (e.g. overwriting a data pointer) or relative addresses (e.g. non-pointer data) can be stopped; concretely AO provides strong security guarantees for pointer data stored on the stack or heap, limited security guarantees for non-pointer data stored on the stack or heap, and theoretical defences for static non-pointer data.

- **Data Structure Layout Randomisation** (DStructLR)(Lin et al., 2009), is a GCC compiler extension that randomises the layout of data structures (e.g. `struct`, classes and stack variables). Since some structures cannot be randomised due to implementation constraints, DStructLR relies on source code annotations made by the programmer to specify which structures can be randomised. DStructLR is enabled when a structure is marked with the attribute `__obfuscate__`, and structure reordering and/or garbage field addition will be deployed if the `__reorder__` or `__garbage__` keywords are specified respectively.

2.5.2.4 Summary of diversity and obfuscation techniques

Table 2.9 shows a comparison of the diversity and obfuscation methods previously discussed to prevent non-control-data attacks. More than half of the implementations are based on compiler extensions that generate either binary code or more code for its final compilation, whereas a minority work at runtime replacing existing memory managers.

Techniques that replace memory managers, such as DieHard and its improved variant, DieHarder, provide more fine grained diversity methods but

Table 2.9: Comparison of diversity and obfuscation techniques for data protection. *C* stands for compiler, *MM* memory manager, *B* binary rewriting, *SCT* source code transformation, *L* library. Regarding protections, a '✓' represents that the scheme protects the target in all possible program segments, otherwise the protected sections are listed.
Additional notes. ✓*: only on structures.

		Target Data		Method	
Scheme		**Pointer**	**Non-Pointer**	**Diversity**	**Obfuscation**
PointGuard	C	✓			X
AO	C	✓	✓	X	X
DieHard	MM	Heap	Heap	X	
DieHarder	MM	Heap	Heap	X	
DR	C	✓	✓		X
DSR	SCT	✓	✓		X
DStructLR	C	✓*	✓*	X	X
librando	L	Stack	Stack		X

only work for heap objects. Regarding compiler-based techniques, both Address Obfuscation and Data Struct Layout Randomisation can provide diversity and obfuscation at some kind of level, being Address Obfuscation the one that gives more security guarantees.

2.6 Summary

In this chapter we have given the background concepts to understand the differences between control data and non-control data, control flow graphs, and we have given a small overview of the static analysis concepts that define the precision of the points-to analysis algorithms. Then, a small overview of control data attacks and defences was presented, followed by another small overview of non-control-data attacks and defences.

The second part of the chapter introduced the state-of-the-art reviews of systems security threat models, control flow integrity methods for user and kernel space, types of non-control-data attacks, and the current trends in defences against non-control-data attacks.

In the next chapter we present an analysis of the security landscape of IoT devices to identify the exact threat models needed by these devices, these threat models demonstrate that non-control-data attacks are an important attack vector in all scenarios.

None of the things one frets about ever happen. Something one's never thought of does.

Connie Willis, Doomsday Book

CHAPTER

3

Understanding the security landscape of control-data and non-control-data attacks against IoT systems

With the recent tendency of deploying software applications in the edge and the popularity of home-based IoT system deployments there are more IoT devices than ever. Consequently, the exploitation of these devices has become more common. Security researchers have focused on understanding the threat model that targets the data centre or the cloud, but their research has not focused yet on understanding the unique security challenges faced by IoT systems and the applicability of the existing threat models for these IoT systems. The remainder of this chapter is organised as follows. Section 3.1 gives an overview of

the rationale of the chapter, section 3.2 presents the proposed IoT device classification, and section 3.3 introduces the threat models. Finally, section 3.4 summarises the contributions of this chapter.

3.1 Introduction

Applications that not long ago were deployed on an environment, such as a data centre or in the cloud, had the security guarantees provided by the results of decades of research done on a well-known threat model. Nowadays the deployment of these applications is shifting towards the edge, where the common threat model does not fit anymore.

In the current landscape the data gathering, data processing and decision making of applications are made as close as possible to the source of those data points, offloading some responsibilities from the traditional data centre or cloud into a heterogeneous mix of IoT devices (Kumar et al., 2019) that are deployed in a vast variety of scenarios, and thereby, complicate the understanding of the threat model (Yuan et al., 2020; Zhou et al., 2019).

The security challenges of this new landscape emerge from the restricted capabilities of IoT devices (e.g., limited CPU, GPU, RAM, storage, networking capabilities) as well as from restrictions related to the environment and location in which these IoT devices are deployed in comparison to deployments in data centres or the cloud, namely the lack of physical control access, which invalidates the common assumption that attackers do not have physical access to the device, and the sometimes remote location in which IoT devices are deployed, which makes human intervention complicated in case of an emergency.

Moreover, even though these IoT devices are being targeted by security threats akin to desktop computers (Antonakakis et al., 2017; Alrawi et al., 2019, 2021), they are also regarded as a high value target not only due to the high amount of personal information that they may hold, but also because they are often tied to the physical world; consequently, a malicious use of these devices can cause direct physical and economical harm (Shekari et al., 2022; Falliere et al., 2011).

Thereby, if these IoT devices are targeted by advanced exploitation techniques unprecedented problems may be caused; for instance, a single byte change in a boolean variable could unlock a smart lock, open the valves in a dam or turn off the cooling system of a nuclear reactor. Attacks specialised in exploiting the non-control data of an application and leaving the control-flow of the application untouched are known as *data-only* or *non-control-data attacks* (Chen et al., 2005).

In this chapter we identify the threat model that IoT devices are facing and the applicability of the previously presented attacks and defences in the IoT landscape.

3.2 The heterogeneous IoT landscape

In order to understand the security threats that the heterogeneous landscape of IoT device types may face, we classify the IoT devices based on the usage that they are going to face to then determine the threat model that they are facing and the possible attacks and countermeasures.

3.2.1 IoT device classification

IoT device classification is made based on the environment and the usage that is going to be given to the device. We chose this classification instead of classifying devices by type or capabilities because the environment in which they are deployed and the usage that they are going to be given modifies the threat model that they face adding more or less risk. For instance, an IP camera may be deployed in a home environment to monitor the behaviour of a dog while their owners are away, whereas the same IP camera may be used to monitor the physical status of a machine at a manufacturing factory. In the first scenario the threat model may be more broad, and more attack scenarios may be possible because the users intend to access the IP camera over the internet, but at the same time, an attack may have less economical and physical repercussions, whereas in the second scenario the IP camera may

be just used in a local closed network and consequently the threat model would be more relaxed, although the stakes could be higher.

Based on this usage classification we identify *consumer IoT devices*, *industrial IoT devices*, and *enterprise IoT devices* as follows:

- **Consumer IoT devices.** These IoT devices are commonly deployed on a home-based environment were physical control access to the devices is in place, however remote access to these devices is commonly enabled. The users of these IoT devices are predominantly not specialised technicians, and thereby deploy the device as is, where default passwords may be still available. These devices are not commonly configured and factory settings may still apply. Security updates and firmware updates depend upon the capabilities of the manufacturer and the ability of the user to apply them. Some sample IoT devices development in this environment are voice assistants (e.g., Amazon Alexa), IP cameras, smart-TVs and smart home appliances (e.g., Roomba) and media players, among others.

- **Industrial IoT devices.** Industrial devices are those that are integrated into the industrial operations by actively participating, monitoring or supporting them; these devices are also used around quality reporting or human monitoring. Industrial IoT devices can also be considered a part of critical infrastructure. These devices usually have limited networking capabilities and are deployed in an air-gap secure network by specialised technicians, moreover there is a day-two operations plan to ensure that security patches are applied and the device is monitored throughout its life-cycle.

- **Enterprise IoT devices.** Enterprise IoT devices share some characteristics with industrial IoT devices: they are deployed and maintained throughout their life-cycle by specialised technicians; the differences are that they are usually deployed in commercial operations, they sometimes have a high degree of human interactivity and have network access. They are also not part of critical infrastructure.

Some example IoT devices of this category are information kiosks (e.g., smart-TVs showing arrival and departure information on airports), self-payment checkout systems and infotainment stations.

3.3 IoT device threat model by device classification, attacks and countermeasures

Based on the previous classification of IoT devices in this section we identify their specific threat models.

3.3.1 Threat model for consumer IoT devices

We assume that an attacker has no physical access to the device but the device is connected to the internet. The device has also been left with the factory-default configurations and updates are unlikely. The operating system of the device is running the available modern mitigations, including W⊕E, KASLR, and kernel CFI.

Attacks. Due to the lack of updates these devices are often left with weak and outdated crypto algorithms, thereby they are vulnerable to Man-in-the-Middle attacks. Due to the weak passwords used in default configuration (Kumar et al., 2019) an attacker can also gain control of the device. There is a chance that these devices will become part of a botnet (Antonakakis et al., 2017).

Countermeasures. In this case the great majority of the attack surface can be reduced with software updates patching CVEs as they are discovered. Most of this work falls upon the responsibility of the vendor (Alrawi et al., 2019). Another security gap is the usage of default passwords and configuration shipped by default in these systems, which needs to be avoided. Up to date documentation and a set-up manager so that the users can change the default configuration can solve these issues. Nevertheless, even if the software is up to date and with the latest security patches these systems are still vulnerable to data-only attacks that could lead to losing the control of the device since the defences against this type of attacks are still academic proposals.

3.3.2 Threat model for industrial IoT devices

We assume that an attacker has no physical access to the device, and that the device is deployed in a secure LAN, the device has no internet access by itself, but other systems in the network do. The devices are running W⊕E, KASLR, and kernel CFI.

Attacks. If the devices are deployed in an air-gap network the only option for attackers is to employ an insider threat to gain access to other systems in the network to deploy malware. This malware could spread in an scenario similar to the `stuxnet` (Falliere et al., 2011) malware to eventually reach the security-critical IoT systems, where advanced exploitation techniques such as data-only attacks or day-0 vulnerabilities can be used.

Countermeasures. In this threat model strong network separation directives should be employed and physical control accesses for employees is needed. If those measures are breached, then the devices are vulnerable to data-only attacks.

3.3.3 Threat model for enterprise IoT devices

We assume that the attacker has physical access to the device, but this is running full encryption and data integrity mechanisms such as secure boot and the Linux kernel Integrity Measurement Architecture (IMA) (Safford et al., 2014) are enabled. The device has internet access, and it is running W⊕E, KASLR, and kernel CFI.

Attacks. In this case the applications running in the device are the main attack vector. If there is a memory error in any them the attackers could use user input to craft a data-only attack in order to gain access to the device or leak security-critical information, such as private keys.

Countermeasures. In this case there are no countermeasures since the device is vulnerable to data-only attacks. Damage-control mindset should be employed to minimise the damage that the device may cause. The device should be granted the minimal amount of permissions and access to linked systems to fulfil its role.

3.4 Conclusions

In this chapter we have conducted an analysis of the security landscape faced by IoT devices identifying the threat model that they are facing based on the environment that the devices will be deployed into. Furthermore, we explain the security capabilities that devices are expected to have in such environments and identify the attacks that they will be facing, as well as defences that they should implement.

This shows that non-control-data attacks are an important attack vector in all scenarios where IoT devices are deployed. While other attack vectors can be easily mitigated, there are still many challenges for the adoption of data-flow integrity in these environments.

Once that the importance of non-control-data attacks has been also demonstrated in the corner case of IoT systems in this chapter, in the following chapter we propose our generalised data-flow integrity approach that entails a series of optimisations that will push the state of the art closer to a generalist solution, which will be also be relevant and applicable to IoT systems.

I didn't get where I am by having reasonable goals.

Ann Leckie, Ancillary Justice

CHAPTER

4

Optimised data-flow integrity for modern compilers

Non-control-data attacks are those attacks that purely target and modify the non-control data of a program, such as boolean values, user input or configuration parameters, and leave the control flow of a program untouched. These attacks were considered a niche due to the high difficulty in crafting attacks that do not modify the control flow. However, in recent years researchers have already demonstrated that non-control-data attacks can be automatically constructed and that they pose a significant threat because they can compromise critical and widely used software, such as web browsers and the Linux kernel. Moreover, they can also be used to disable or bypass state-of-the-art software security techniques, such as control-flow integrity. The most promising technique to protect against non-control-data attacks is data-flow integrity, however, modern compilers do not implement this protection yet. In this work we present an optimised data-flow integrity implementation for modern compilers that

reduces the amount of basic blocks that need to be protected in an average of 45.8%, it also has broader security guarantees due to its more precise static analysis. Finally, we evaluate the completeness of our optimised data-flow integrity implementation.

4.1 Introduction

Since Chen et al. first raised awareness (Chen et al., 2005) of the theoretical dangers of attacks targeting the non-control-data of a program, security researchers have not only demonstrated that non-control-data attacks are feasible and that they can be automatically constructed (Hu et al., 2015, 2016; Ispoglou et al., 2018), but also that they are capable of hindering critical software, such as modern browsers (Jia et al., 2016; Rogowski et al., 2017; Park et al., 2020), and the Linux kernel (Davi et al., 2017; Zhou et al., 2024). Moreover, some of these works demonstrate (Carlini et al., 2015; Davi et al., 2017) that non-control-data attacks are also being used to bypass different implementation variants of control-flow integrity (CFI) (Abadi et al., 2005). Nowadays CFI is considered the state-of-the-art technique against control-flow hijacking attacks, and it is present in the two most prominent compilers (i.e., GCC and LLVM) (Tice et al., 2014). By using non-control-data attacks as a stepping stone to disabling CFI, attackers are free to utilise a wider range of advanced exploitation techniques, such as return-oriented programming (Shacham, Hovav, 2007) or other variants.

In order to eradicate non-control-data attacks, statistical defences focused on randomising or encrypting the layout of data in memory have been proposed (Bhatkar and Sekar, 2008; Belleville et al., 2018); however, due to the statistical nature of these defences, they are often vulnerable to information leak attacks. At the same time, the most prominent runtime defence technique to hinder non-control-data attacks is data-flow integrity (DFI) (Castro et al., 2006). DFI is a runtime defence technique that checks which instructions are allowed to write to different memory addresses, and thus is able to prevent non-control-data attacks. However, DFI is limited by the quality of

the static analysis that guides its runtime checks, and by the performance penalties incurred by using such defence at runtime.

Since DFI was originally presented nearly two decades ago, specialised solutions have arisen to solve particular instances of non-control-data attacks (Akritidis et al., 2008; Song et al., 2016a); however, to the best of our knowledge there have not been efforts to optimise and broaden the security guarantees of the original DFI.

The contributions of this paper are as follows:

- We present a design, and practical implementation of DFI in modern compilers.
- We augment the security guarantees provided by the original DFI proposal by strengthening the static analysis adding field-based points-to analysis and taking call-context into account.
- We propose a first optimisation that allows users to define their own critical basic blocks so that those can be automatically chosen to be instrumented.
- We further optimise DFI by providing a control dependent data optimisation to reduce the subset of basic blocks that need to be instrumented.
- We evaluate the completeness of DFI in the presence of compiler optimisations such as tail calls.

The rest of the chapter is organised as follows: we briefly discuss the rationale of our work (§ 4.2), we overview the basic details of DFI in modern compilers (§ 4.3), we present the threat model applicable to our optimised DFI implementation (§ 4.4), we present the design of our DFI implementation (§ 4.5), we explain our basic-block optimisations (§ 4.6), we discuss the challenges of implementing DFI in a modern compiler (§ 4.7), we evaluate the completeness of our implementation (§ 4.8), to then finally present our conclusions (§ 4.9).

4.2 Rationale

Security critical data in any program can be classified as *control data* or *non-control data*. *Control data* is used to drive the control-flow of a program (e.g., return addresses, function pointers). In contrast, while *non-control data* does not explicitly drive the control-flow of a program, it can still serve as a decision point for control-flow statements (e.g., boolean values, user input, configuration parameters).

Memory errors were first exploited to inject malicious data that will later on be executed as code. In these scenarios it is possible to modify control data and then force the execution to such malicious code. Modern mitigations like DEP/W⊕X (Andersen and Abella, 2004) prevent these exploitation techniques, and as a consequence nowadays most attacks try to reuse existing control data to subvert the control-flow of a program, and thereafter perform malicious actions (Szekeres et al., 2013). However, due to the effort made by the security industry and research community many promising defences (Abadi et al., 2005; Kuznetsov et al., 2014; Team, 2003; Tice et al., 2014; Niu and Tan, 2014b; van der Veen et al., 2015; Prakash et al., 2015; Zhang et al., 2015; van der Veen et al., 2016) have been proposed that make control-flow hijacking attacks increasingly difficult.

4.3 Data-flow integrity instrumentation in compilers

Data-flow integrity (DFI) (Castro et al., 2006) is a runtime security mechanism that prevents the use of corrupted memory variables that could lead to non-control-data attacks.

In an offline stage, DFI computes the *reaching definitions* (Aho et al., 2006; Nielson et al., 2015) of each variable, a data-flow analysis that shows where a given variable may have been defined when program execution reaches a precise point. At runtime, it tracks the last writer to each memory position in a runtime definitions table (RDT); when a variable is used (read), it checks

whether the last definition (write) made to that variable *reaches* the program at that point (the definition is *live*). If the last definition is not in the statically determined *reaching definitions* set, then the latest write to that variable is considered illegal and an exception is thrown, as the use could potentially be malicious. Otherwise, the use is allowed and execution continues.
It can also prevent some control-flow hijacking attacks since it instruments the target addresses of indirect control-flow transfers added by the compiler (*ret* addresses).

DFI is implemented at the compiler level to instrument the uses and writes made to the variables. It uses two high-level instructions (i) `SETDEF addr, id` and (ii) `CHECKDEF addr, set-id`; which write the given `id` to the RDT in the given address position, and check that the last identifier in the given address is in the set of *reaching definitions* statically determined, respectively. These high-level instructions are specifically lowered into x86 assembly. The RDT is protected by being in a well-known memory location and its bounds are checked to prevent tampering with it.

DFI comes in two flavours (i) *intraproc* DFI, which only instruments uses of local variables and control data, with an average 45% overhead, and (ii) *interproc* DFI, which handles local variables, local address-taken variables and static and global variables resulting in an average 104% overhead. For both cases the average space overhead is 50%.

4.4 Threat model of our optimised data-flow integrity implementation

Although the original DFI implementation could protect against some control-flow hijacking attacks, the main goal of our work is solely to protect programs against non-control-data attacks. Protecting against control-flow hijacking attacks and other exploitation techniques that employ control-data is an orthogonal problem, we thus leave those defences to other specialised solutions such as CFI (Abadi et al., 2005), CPI (Kuznetsov et al., 2014), and others.

Our threat model has reasonable assumptions and it is consistent with other works (Frassetto et al., 2017); it is defined as follows:

- **Standard security mechanisms: DEP/W⊕E, ASLR.** We assume that the operating system provides standard security mechanisms protecting against code-injection attacks using DEP/W⊕E (Andersen and Abella, 2004), and hinders the localisation of addresses to employ code-reuse attacks by employing some form of Address Space Layout Randomisation (ASLR) (Team, 2003).

- **Control-flow integrity.** Modern defences against code-reuse attacks, both for forward-edge and backward-edge targeting attacks are in place by some form of fine-grained CFI (Abadi et al., 2005) with a shadow stack.

- **Memory-corruption vulnerability.** We assume that user-space programs have memory-corruption vulnerabilities that could be used to read and write user-space non-control data.

- **RDT and static data-flow graph protection.** We assume the runtime definitions table employed by DFI to keep track of the runtime definitions as well as the static data-flow graph to whom those definitions are compared to is protected by high-level protection mechanisms (e.g., memory isolation), and thus the attacker cannot read or modify those. Such protection mechanisms could vary depending on the system that DFI is being deployed in.

4.5 Design

In this section we introduce the general design of our system as shown in Figure 4.1. The (i) *static analysis* component (§ 4.5.1) collects security sensitive data, such as pointer information, uses and definitions, called functions, equivalences, global variables and context information from the source code of the program to be instrumented, the output is the static data-flow graph

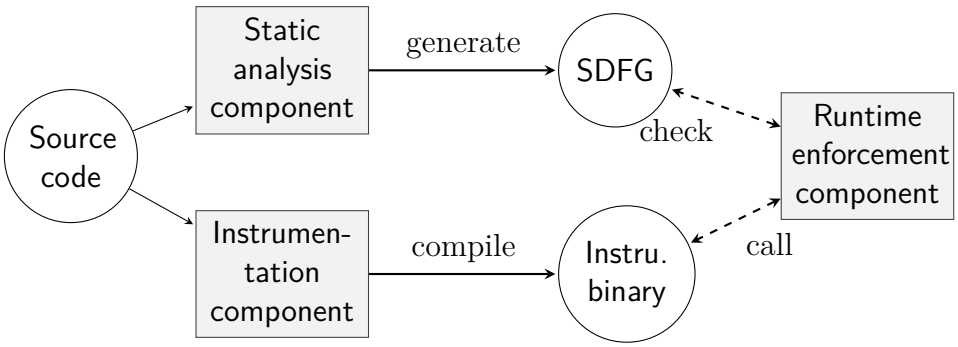


Figure 4.1: High level overview of the components of our optimised DFI implementation

(SDFG) of the target program and other relevant metadata; the (ii) *instrumentation* component (§ 4.5.2) uses some metadata generated by the static analysis component and instruments the target program inserting high-level custom instructions to provide DFI. The (iii) *runtime enforcement* component (§ 4.5.3) is in charge of handling changes made to the RDTs and enforces the DFI property, to do so it utilises the SDFG generated as part of the static analysis.

The instrumented binary will have our new optimisations for the instrumentation of security-critical non-control-data, which we discuss in § 4.6.

4.5.1 Static analysis

The main goal of the static analysis component is to generate the static data-flow graph (SDFG), which will be consulted at runtime to check in a given point whether a variable is live.

For each source file the static analysis component (i) utilises an intra-procedural flow-sensitive and field-sensitive analysis to gather the uses and definitions of local and global variables in each function. Only variables that are used in memory or spilled to memory from registers are considered.

An inter-procedural points-to analysis (ii) generates alias information of security critical variables. At each function call the parameters are recorded and the alias values updated in the callee; additionally we check whether the return value of the called function becomes aliased with any of the function

parameters performing backward slicing (Weiser, 1984) on the return value of the callee.

The static analysis component also (iii) collects which functions are part of the whole compilation unit, since only the context of internal functions will be handled in the instrumentation component.

These three procedures are performed in an iterative mode per source file. In a given point the static analysis may not be complete since further functions from other additional source files could be still needed to complete the alias information. If these information gaps are not filled by the end of the analysis, supplementary information is passed to the instrumentation component to avoid instrumenting those variables taking account their context and reverting to local-only instrumentation. At this point the static analysis may also determine that the given program is not sound (e.g., the target of a function call has not been defined), and consequently, the analysis is interrupted early.

When the static analysis component finishes, reaching definition set identifiers are assigned to each unique reaching definitions set, and equivalent sets are assigned the same identifier; then, each variable is assigned one set identifier and the information is written to a file, plus other complementary metadata.

4.5.1.1 **Comparison to the original data-flow integrity**

The original DFI (Castro et al., 2006) uses a field-insensitive points-to analysis based on Andersen's algorithm (Andersen, 1994). Since the instrumentation relies on this type of analysis, it cannot detect attacks that overwrite different fields in data structures. A recent study has shown (Morton et al., 2018) that it is feasible to identify low-level memory structures mapped to high-level C structs to then modify them by means of a non-control-data attack, thereby the original DFI cannot protect against attacks focused on this level of granularity. In comparison, our approach uses field-based static analysis that fully differentiates different fields in data structures and thus can detect manipulations to field-based data structures.

Moreover, the original DFI does not take call-context into account, whereas our analysis computes alias equivalences at each function call to pass a call context in each function call.

In conclusion, the security guarantees that our implementation provides are broader since our reaching definitions data-flow analysis has been constructed with a more complete static analysis.

4.5.2 Instrumentation

Our approach follows some of the design guidelines of the original DFI to implement the two high-level instructions (`SETDEF`, `CHECKDEF`) described there (Castro et al., 2006), however our work has some key differences and consequently implements some additional high-level instructions. Since our static analysis gathers context information and is field-sensitive, it demands to differentiate between instrumenting normal variables and field-based data structures, resulting in different types of `SETDEF`/`CHECKDEF` instructions; moreover, additional instructions to handle context are needed and an instruction to load the static data-flow graph (SDFG) into memory to employ comparisons is also required.

The code fragment in 4.4 shows the pseudo-code of a sample program taken from (Castro et al., 2006) that depicts a sample data-only vulnerability in SSH when an attacker overwrites `auth` using an overflow in `packet`.

Our approach would instrument the vulnerable SSH program (Listing 4.4) as shown in the pseudo-code of Listing 4.5, where our instrumentation is denoted within brackets (`[]`).

The set of high-level instructions used to instrument the program is explained in the following section.

First, we define the data structures needed in our approach (§ 4.5.2.1), the different instructions injected in the code to provide the DFI property (§ 4.5.2.2, § 4.5.2.3, § 4.5.2.4, § 4.5.2.5), as well as how are are special cases such as global variables (§ 4.5.2.6), recursive calls (§ 4.5.2.8) or indirect calls (§ 4.5.2.7) handled.

```
1  int auth = 0;
2  char packet[1000];
3
4  while (!auth) {
5    PacketRead(packet);
6
7    if (Authenticate(packet)) {
8      auth = 1;
9    }
10 }
11
12 if (auth) {
13   ProcessPacket(packet);
14 }
```

Code 4.4: Pseudo-code of a data-only vulnerability in SSH.

4.5.2.1 Data structures

Apart from the runtime-definitions table (RDT) where the address of each definition is stored with the given ID, we have included another RDT for global variables; furthermore, we use two additional lists of frames to store local addresses and shared addresses per call-frame: the Local Address Frame (LAF) and the Shared Address Frame (SAF), respectively. Another structure, the Variable-Context List (VCL) stores for each defined address, which are the different set-IDs that are used to define its context.

The contents of these data structures are modified during runtime based on calls made to `SETDEF`/`CHECKDEF` instructions, or by specific instructions that handle context.

The DFI checks rely on information gathered during compile time, which is stored in two data structures: a map that holds the definitions that are allowed for each set-ID, and a two-tier nested map that holds, for each set-ID of a field-based data structure, the different offsets at which it has a field, and the corresponding set-ID assigned for each field (Field-Offset Index).

In contrast to the original DFI, the set-IDs in our implementation may not refer to unique lists of definitions when the set-ID is used to reference a

```
1  [ load_statics() ]
2  [ create_frame() ]
3  [ set_def(&auth, 1) ]
4  int auth = 0;
5  [ set_def(&packet, 2) ]
6  char packet[1000];
7
8  [ check_def(&auth, set_id_11) ]
9  while (!auth) {
10   [ check_def(&auth, set_id_11) ]
11   [ create_shared_frame() ]
12   [ share_var(set_id_12, &packet) ]
13   PacketRead(packet);
14   [ unshare_frame() ]
15
16   [ check_def(&packet, set_id_12) ]
17   [ create_shared_frame () ]
18   [ share_var(set_id_12, &packet) ]
19   if (Authenticate(packet))
20   [ unshare_frame() ]
21   {
22     [ set_def(&auth, 8) ]
23     auth = 1;
24   }
25 }
26
27 [ check_def(&auth, set_id_11) ]
28 if (auth) {
29   [ check_def(&packet, set_id_12) ]
30   [ create_shared_frame() ]
31   [ share_var(set_id_12, &packet) ]
32   ProcessPacket(packet);
33   [ unshare_frame() ]
34 }
35 [ delete_frame() ]
```

Code 4.5: Pseudo-code of the vulnerable SSH program instrumented with our implementation.

set of definitions for a field-based variable. In our approach it is primordial to uniquely differentiate those field-based variables and their fields, thereby those set-IDs are unique even if they refer to the same set of definitions.

4.5.2.2 Static data-flow graph loading instruction

A single `load_statics` instruction is injected after the prologue of the entry point of the instrumented program. It loads the required static information to handle DFI checks: the static data-flow graph (SDFG) and the Field-Offset Index.

The protection of shared libraries or those programs without a main is out of the scope of this work.

4.5.2.3 Context handling instructions

The following instructions are used to create and delete LAFs and SFAs:

- `create_frame`/`delete_frame`. At the beginning of each function a `create_frame` is inserted, which will create a LAF, and a `delete_frame` is placed at the end of the function. This new LAF will hold the addresses defined by every `set_def` instruction. When the function is about to exit, `delete_frame` will be called and each of the addresses defined in this LAF will be compared against the addresses of variables that were shared for this frame (if any) and stored in the SAF. If those addresses are local, they will be deleted from the RDT; finally, the LAF is deleted.

- `create_shared_frame`/`unshare_frame`. Before each function call (direct or indirect) considered internal (i.e., calls within the global compilation unit or indirect calls whose call destination is unknown) a `create_shared_frame` is placed before the call and a `unshare_frame` after the call. These instructions simply create and delete a SAF respectively, where the addresses of the shared variables and their set-IDs will be stored.

- `share_var/share_var_lvl setid, addr`.

  When a variable is passed as a parameter in a function call we also need to share it with this instruction. `share_var` adds the address and set-ID of the shared variable to the previously created SAF; moreover, it adds the set-ID to the VCL, updating the list of set-IDs that make the whole context of the variable.

4.5.2.4 **Instructions for handling non-field-based variables**

These are the custom high-level specialised instructions introduced by our approach in order to handle non-field based variables:

- `set_def/set_def_lvl addr, id`. These instructions handle definitions of variables, the address is used as a key in the RDT and the ID is added as the value taken, the ID is defined as the line (given by the compiler) in which the variable is defined, as the original DFI did. We also take note of the address in the LAF.

- `check_def/check_def_lvl addr, setid`.

  These instructions check that the given address exists in the RDT, and that the ID of the last writer is in the given set-ID. This last check is preemptively allowed to fail since the given set-ID does not take into account the whole context in which the variable is living. We get the remaining context of the variable using the VCL and check again, if this check fails a DFI error is thrown.

Instructions with a `lvl` keyword (`share_var_lvl`, `set_def_lvl`, `check_def_lvl`) perform the same actions as their fellow instructions without `lvl`, but they perform a given number of runtime dereferences beforehand, to then execute the common actions. This is done to handle pointers that require dereferences, see § 4.7.2.1.

4.5.2.5 Instructions for handling field-based variables

These instructions are used to handle variables that might have other associated addresses (i.e. structs) which need to be handled as a batch:

- `share_fvar setid, addr, offset`. In addition to the steps taken by its non-field based counterpart, this instruction also shares the set-ID of the variables that the given set-ID has attached to (i.e. all the fields of a given struct). The required addresses are calculated using the Field-Offset Index, and all possible nested variables are also handled.

- `set_def_fvar addr, offset, id`. Apart from the same functionality of its non-field based counterpart, the base address of the variable (*addr* - *offset*) is also added to the LAF and the base address is added to the RDT, with a dummy definition.

- `check_def_fvar addr, offset, setid`.

  This instruction checks that the given address exists in the RDT, and also that the ID of the last writer is included in the sets definitions specified by given set-ID. This first DFI check is allowed to fail on first instance when the offset of the given variable is 0, and also since further checks due to possible aliases and context handling must be taken into account.

  Nested structs placed in offset 0 will share the same address but will be assigned unique set-IDs, thereby all the possible nested set-IDs with the given set-ID as is outermost variable will be retrieved and checked against, if any of those checks is successful the DFI property will be considered satisfied.

  To check if the definitions were made in a previous context the VCL is consulted.

4.5.2.6 Handling global variables

To handle the definitions and checks when global variables are used, we also differentiate between field-based and non-field-based variables but the

definitions are stored in a specific RDT for global variables and no context handling is required.

4.5.2.7 **Handling indirect calls**

The LAF creation and deletion has been modelled after the function prologue and epilogue of assembly languages. We delete the SAF in the caller after the callee has been executed, rather than in the callee itself. This allows to handle every indirect-call target inside the compilation unit.

When an indirect-call to a function outside the compilation unit is made, the required variables are still shared and once the indirect-call returns, the shared frame will be deleted; similarly if the target of the indirect-call is inside the compilation unit, it will execute, perform any DFI related checks it might have and return; this causes a slight space overhead due to the nesting nature of real-world programs, but the alternative is to forfeit context information and revert to basic local DFI checks wherever *any* indirect-call is made, resulting in looser security guarantees.

4.5.2.8 **Handling recursive calls**

Before a direct recursive call, any `share_var` instructions and the `create_shared_frame` / `unshare_frame` instruction pair are exceptionally not injected because we claim that the security guarantees would remain equivalent and the overhead is lower (23%) according to our experiments.

The rationale behind this design decision falls within the nature of recursive calls (which can be reduced to loops), and the requirements of DFI to instrument every definition and use of in-memory variables. After the call from an outside function to the recursive function is made, the recursive function will have a SAF with the addresses passed from the previous context, and any checks made against variables passed down as arguments will continue having context-sensitive security guarantees. On the other hand, checks made against local variables, both local by the current context of the recursive call and local in previous recursive-contexts, will continue to be made.

4.5.3 Runtime enforcement

The runtime enforcement component handles all frame/context management operations and propagates *setdefs* made in a shared-RDT before it is deleted. It also charges the SDFG into memory, takes note of definitions made by `SETDEF` instructions and performs the relevant checks required for `CHECKDEF` instructions. All the high-level instructions described above (see § 4.5.2) have their implementation in this component.

4.6 Optimisations

Although DFI remains as the main reference to prevent non-control-data attacks, its performance downside has prevented its wide usage in real-world applications, even though it includes several optimisations (Castro et al., 2006).

To improve the prospects of DFI's usability in real-world applications, we propose a new series of optimisations that aim to remove all `CHECKDEF` instructions concerning non-control-data that has not been considered security critical.

4.6.1 Rationale

The non-control-data of given program may have different purposes; for example, it may be used to make computations, hold user-input parameters or decide which execution path the program will take. From a security standpoint, we should only be concerned about *security-critical* non-control-data. The original DFI implementation instrumented *all* non-control-data. Instead, we propose to *only* instrument the subset of security-critical non-control-data.

4.6.2 Implementation of the optimisations

In order to determine which non-control data can be considered security-critical we refer to Chen et al. (Chen et al., 2005), where their work identified that configuration data, user input and user identity data along with decision-making data should be considered security-critical.

Configuration, user input and user identity data are highly program-dependent, whereas decision-making data can be broadly defined as that data that is used to trigger changes in the control-flow. These types of security-critical non-control-data are taken into account when designing the following optimisations:

4.6.2.1 **Critical data types**

We comprise configuration, user input and user identity data as the term *critical data types*, and it will concern data whose types (as in storage data types) have been determined critical by an expert (e.g., `task_struct` structs in the Linux kernel or `ngx_command_t` structs in Nginx). This optimisation is applied per basic block and removes the DFI checks to every non-critical var type.

4.6.2.2 **Control dependent data optimisation**

Control dependencies exist between two statements, *S1* and *S2*, whenever the predicate of *S1* controls whether *S2* is executed; if so, *S2* would be control dependent on *S1*. Following this example we can say that the legitimate execution of *S1*'s predicate is security critical for *S2*'s legitimate execution.

Given a control-flow graph (CFG), we can also establish this relationship between basic blocks using the algorithm described by Ferante et al. (Ferrante et al., 1987) to determine the control dependences of a program.

To generate this control dependency relationship we first construct the post-dominator tree for the given CFG, then we check each edge of the CFG and take note of those edges whose destination basic block does not post-dominate the source basic block; in order to accomplish this, we use the previously generated post-dominator tree.

The source basic blocks of the resulting edges are the basic blocks to whom the rest of the blocks in the CFG may be dependent on. Those basic blocks to whom other basic blocks are control dependent are considered *security critical*, and their *checkdefs* will be kept. The other *checkdefs* will be removed. Nonetheless, we cannot split a CFG in two types of basic blocks, those which

Table 4.1: Reduction in the instrumentation size per basic block and program using the Control dependent data optimisation.

Benchmark	Basic Blocks	Critical Basic Blocks	Reduction %
401.bzip2	2550	1249	51.01
403.gcc	159049	88143	44.58
429.mcf	437	239	45.30
433.milc	3246	1662	48.79
445.gobmk	30192	18580	38.46
456.hmmer	10181	5458	46.39
458.sjeng	5046	2847	43.57
462.libquantum	956	506	47.07
464.h264ref	16789	8388	50.03
470.lbm	181	102	43.64
482.sphinx3	5005	2753	44.99

are dependent on others and those which are not; thereby, we also calculate the exact dependencies between blocks and mark those which are not part of any of those categories as security critical as well. This is also done using the algorithm provided by Ferante et al., which traverses post-dominator tree backwards from the destination basic block of each of the edges selected beforehand up to the source basic block's parent. Every basic block in between, excluding the parent, is control-dependent on the source basic block.

For CFGs with more that one exit (e.g., exits that handle exception paths) we cannot apply this optimisation since the post-dominator calculation requires a single exit in the CFG.

4.6.3 Results

Table 4.1 shows the per program reduction in the instrumentation size after running an experiment with the control dependent data optimisation on SPEC CPU2006 using C benchmarks. The results show that our approach can remove the `CHECKDEF` instructions in 45.8% of the basic blocks on average.

4.7 Implementation

In order to select a compiler to implement our optimised DFI variant, we focused on selecting the compiler that could handle the broader amount of use cases, both in user and kernel space, to be able to build upon this implementation in future works. Thereby, we chose GCC due it's variety of targets, as well as for being the official compiler being used to build the Linux Kernel.

Our work uses `gcc` version 5.4.0; this version has been selected to match the `gcc` version available in our lab environment which is used to run the completeness experiments shown in section 4.8.1.

The static analysis and instrumentation components are implemented in two plugins for the GCC compiler collection, which consist of 6,400 and 14,300 lines of C/C++ code respectively, whereas the runtime enforcement component is a shared library consisting of 1,100 lines of C/C++ code.

These plugins allow users to add DFI to their C/C++ programs with their existing GCC distribution without the need to recompile the whole compiler. The plugins attach the new compilation passes defined by our work to the existing compilation infrastructure, the user just needs to add the parameter `-fplugin` to load the required plugin and specify some plugin related parameters (i.e., folder locations to store/load the SDFG). The plugins operate in GCC's middle-end, where machine-independent optimisations are made, thereby our implementation is also machine-independent.

The following sections describe the challenges and peculiarities of implementing the static analysis component (§ 4.7.1) and the instrumentation component (§ 4.7.2) in GCC, finally we show how the runtime library (§ 4.7.3) has been implemented.

4.7.1 Static analysis component

The static analysis component is a GCC plugin which works on top of GCC's machine-independent intermediate three-address code representation, GIMPLE, to gather uses and definitions of security critical variables (see § 4.5.1).

The static analysis plugin consists of two inter-procedural passes and an intra-procedural pass which are attached to the middle-end right after the static single-assignment (SSA) form has been created. The intra-procedural pass traverses the SSA form to gather field-sensitive and flow-sensitive use-def chains which are directly translated into uses and definitions of variables; one of the inter-procedural passes performs the context-sensitive analysis following the procedure described in § 4.5.1 building upon the context-insensitive information gathered in the previous pass. The remaining inter-procedural pass recovers metadata of all the functions of the compilation unit.

4.7.1.1 **Bit-fields**

Due to the variety of compilation targets that GCC supports, the compiler marks as addressable memory operations that take addresses smaller that a byte. Our work is framed to handle standard C with target x86-64 due to the equipment chosen to run the security experiments that we have run to validate our claims, thereby we ignore uses and definitions to those bit-fields since we cannot address anything smaller than a byte in standard C (ISO 9899:2024(en), 2024).

Nevertheless, adapting our implementation to other architectures or environments that may require bit-field addressing is trivial.

4.7.1.2 **Variadic functions**

Functions which take a variable number of arguments represent a challenge since, by definition, at compile time we cannot distinguish variables part of the optional argument list among themselves. Thereby, optional arguments of variadic functions are handled by the static analysis algorithms as a single argument.

4.7.2 **Instrumentation component**

The instrumentation component is implemented as the second GCC plugin, which consists of five passes following the design specified in § 4.5.2 and § 4.6.

```
1 | x = &y
2 | x = *y
3 | *x = y
```

Code 4.6: Valid pointer assignment operations in SSA form.

The aim of these five passes is threefold: (i) instrument the code, (ii) check that the instrumentation remains consistent after the standard GCC compiler optimisations are made, and (iii) perform our last-minute optimisations.

The passes handling the instrumentation are hooked after the SSA form has been built and before the main machine-independent optimisations have finished. The instrumentation pass is hooked as early as possible to prevent tampering with standard compiler optimisations. Once the code has been instrumented and the machine-independent optimisations are done, we hook some other passes to check the integrity of the instrumentation (e.g., frame creation and deletion must come in pairs) just before the SSA form is translated into the machine-dependent register transfer language (RTL).

Apart from the functionality described in § 4.5.2 there are other aspects that we have considered to implement DFI in GCC. We discuss them below.

4.7.2.1 **N-level Indirection Pointers**

Our approach concerns the safety of non-control data, thereby when a pointer with multiple levels of indirection appears in the r-value of a SSA statement, we inject a `CHECKDEF` to the address of the bottom-most pointer. Due to the design of the SSA form, valid basic pointer assignment operations apart from indexed copy instructions have one of the forms (Aho et al., 2006) shown in Listing 4.6.

Thereby, pointers with more than one level of indirection require one statement per dereference level. Instead of making multiple dereferences to take note of the target address for the required `CHECKDEF`, our implementation takes the r-value of the topmost SSA statement, calculates at compile time the number of indirection levels that the target address requires, and then, at runtime, the required numbers of dereferences are made. Moreover, if multiple

```
1 | p.4_18 = p;
2 | p.5_19 = p.4_18 + 1;
3 | p = p.5_19;
4 | // omitted
5 | p.1_12 = p;
```

Code 4.7: SSA code from `ngx_utf8_length` function (`ngx_string.c`) from nginx

`CHECKDEF`s were to be made as a consequence of multiple SSA dereferences, only the topmost would remain and the rest would be optimised out as per the basic optimisations (Castro et al., 2006) of DFI.

4.7.2.2 "Dynamic" pointers

As per the definition of the SSA form, all variables are distinguishable from each other since each definition to a variable uses a different name (Aho et al., 2006); commonly, instead of renaming the variable, the GCC compiler keeps its original identifier and its version changes.

During our experiments we identified some cases where assignments in SSA form did not update the version of the variable. Since our static analysis component traverses the SSA form to get information about uses and definitions of variables, this resulted in inaccurate static information that could lead to false positives if those specific non-updated variables were used.

The example shown in Listing 4.7 shows a code extract in SSA form from the `ngx_utf8_length` function (`ngx_string.c`) from nginx.

The code shows how p is not updated. When a *checkdef* is made to check DFI on variable p in statement 5, a DFI error is thrown since the address location where p resides is not defined, it has been overwritten in statement 3.

In order to prevent these false positives we instrument the code with a special instruction (`check_dfg_update`) before and after the address change; at runtime, the first instruction before the address change takes note of the current address for the variable, and after the address is changed, we get that

```
1 | p.4_18 = p;
2 | p.5_19 = p.4_18 + 1;
3 | check_dfg_update_before (p);
4 | p = p.5_19;
5 | check_dfg_update_after (p);
6 | // omitted
7 | p.1_12 = p;
```

Code 4.8: SSA code from `ngx_utf8_length` function (`ngx_string.c`) from nginx with `check_dfg_update_before` instrumentation

new address and update the different RDTs replacing the old address with the new one, as shown in the updated Listing 4.8.

The updated example at Listing 4.8 in SSA form shows how the write to `p`, now in statement 4, has a pair of `check_dfg_update` instructions before and after the variable p is updated, which allow us to catch the address change and update the RDTs.

4.7.2.3 Calls to external known-to-be-problematic functions

Even in the case of a fully *sound* and *complete* points-to analysis, DFI can still not detect attacks that happen outside the instrumentation scope, for instance, when a user misuses a function from an external library.

Many non-control-data attacks occur when the bounds of objects in memory are surpassed in a malicious way to overwrite other locations that store non-control-data variables; there are functions with a widely spread usage from common libraries, such as `libc`, that could lead to this insecure behavior (e.g., `memcpy`, `strcpy`). When the instrumentation component encounters one of such known-to-be-problematic functions, it calculates the addresses that will be overwritten by the specific function and inserts the required *set-defs* for every location that will be modified before the actual call to the `libc` function.

4.7.2.4 Reducing undefined behaviour

Using automatic variables before they have been initialised is one of the most common examples of undefined behaviour in C/C++. In these cases, the user is solely responsible for writing code without undefined behaviour and the compiler might yield some warnings. In the case of GCC, the compiler provides the optional `-Wuninitialized`, `-Winit-self`, and `-Wmaybe-uninitialized` warning options (GNU, b).

Our work gives more assurances to the user preventing this type of undefined behaviour in C. Due to the nature of DFI, when a variable is used a DFI check will be done, and if the variable has not been defined a data-flow integrity error will be yielded, removing the uncertainty of undefined behaviour in such cases.

4.7.2.5 ϕ-functions

ϕ-functions are used within the SSA form to define that a variable can be defined in more than one control-flow path. The result of the ϕ-function is the value of the argument that has the control-flow path chosen at runtime (Aho et al., 2006). For instance, given the following statement:

$x_3 = \phi(x_1, x_2)$

the variable x_3 would be either x_1 or x_2, depending on the control-flow of the function, but the compiler will always define a new variable x_3, and continue with the optimisations using this new variable.

In our experiments we have seen that the results of the ϕ-functions are often optimised out to registers, moreover since it is impossible to statically determine which of the arguments of the ϕ-function were used when we need to share the context of the result of a ϕ-function, we set the same set-ID for all the possible ϕ-function address arguments.

This conservative solution focuses on eliminating the false positives that would have been yielded if no set-IDs were shared when the context resided within the result of a ϕ-function.

4.7.3 Runtime enforcement library

When our DFI related high-level instructions are lowered by the compiler they will invoke the routines of our runtime enforcement C/C++ shared library, thereby programs need to be linked against our shared library. Unlike the original DFI, which lowered its high-level instructions to x86 assembly, our work is software-based to be cross-compatible.

4.8 Completeness of DFI

The completeness of the DFI integrity checks relies on the validity of the instrumentation directives, this completeness depends on (i) code execution integrity, and (ii) compiler code generation.

Code execution integrity is provided by DEP/W⊕E in modern operating systems, thereby it is guaranteed under our threat model (see § 4.4); whereas code generation heavily depends on the compiler optimisations chosen to compile a piece of code, which previous studies show (Lin and Gao, 2021) that might impact the injected checks.

The use of some compiler optimisations is critical to the correctness of any DFI algorithm implementation. Specifically, the usage of obscure tail-calls (e.g., tail calls that jump to an arbitrary position of the callee, instead of the beginning of a well defined function) may hinder the security guarantees that this implementation offers if the optimisations that generate these tail calls are enabled.

DFI implementations are based on the assumption that functions are atomic and independent units of code, with a well defined entry point and one or more exit points (return instructions). DFI assumes that the control flow of the program will respect these rules, inserting instrumentation code at strategic points that implement checks that guarantee data-flow integrity. If any optimisation subverts these assumptions, the completeness of DFI is no longer guaranteed.

In this section we analyse the usage of tail calls, to evaluate how common they are, and to which extent they can affect the completeness of our DFI implementation.

4.8.1 Experiments

Our experiments were conducted using Nucleus (Andriesse et al., 2017), the state of the art function identification tool which, to the best of our knowledge, is currently the tool that provides the best precision and recall. This tool helped us with the identification of function boundaries and `call`/`jmp` instructions between them.

Our test suite has been crafted to contain a sample of the most widely used programs in the world, and it includes `binutils-2.32` and `coreutils-8.31`, which serve as a sample of common and widely used Unix programs, as well as `nginx 1.4.0`, a well-known web server which is the most widely used server in the world (Web Tecnology Surveys, 2024).

The programs are compiled with `gcc` version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.12).

4.8.2 Completeness results

Initially, our experiment pointed out (refer to row *Target unknown* of Tables 4.2, 4.3 and 4.4) 129 cases of tail calls to unknown targets in `binutils` and 18 cases in `coreutils`, both compiled with -O0. These cases (marked with * in Tables 4.2, 4.3 and 4.4) refer to targets of addresses outside the boundaries of any function.

We inspected these cases and found certain errors and inconsistencies on the results provided by Nucleus where the function boundaries did not include the last basic block of the function. The majority of these cases were observed in the presence of jump tables (corresponding to switch clauses). These special cases were reported by the Nucleus authors, and affect the inter-procedural

Table 4.2: Tail call experiments for the `binutils` suite. *Target start* refers to tail calls that jumped to the beginning of the target function, *Target not start known* jumped to known locations that were not the beginning of the function, *Target unknown* jumped to unknown locations, *Target PLT* jumped to the PLT and *Target cannot compute* are jumps whose target cannot be statically determined.

	binutils			
#Binaries	15			
Optimisation	-O0	-O1	-O2	-O3
#Functions	26608	21242	19468	18277
#Normal calls	162504	148647	136594	158888
#Tail calls	168	0	2406	1590
Target start	0	0	1677	1026
Target not start known	0	0	40	0
Target unknown	129*	0	0	0
Target PLT	39	0	689	564
#Target cannot compute	1381	1423	1996	2222

CFG generation which is later on used to detect the function boundaries. We inspected these cases and filtered out these false positives.

4.8.2.1 Tail calls to the PLT

Similarly, a number of tail calls (refer to row *Target PLT* in Tables 4.2, 4.3 and 4.4) do not jump to a function of the program, but to the PLT, which is in charge of resolving and jumping to a target function in a dynamically loaded function (e.g., system library calls such as a program exit function). These particular cases do not affect the instrumentation added by DFI techniques.

4.8.2.2 Call/Jmp targets that cannot be statically computed

During these experiments we also observed cases where it was not possible to statically compute the target of a function call (refer to row *Target cannot compute* of Tables 4.2, 4.3 and 4.4). In many of these cases, the target

Table 4.3: Tail call experiments for the `coreutils` suite. *Target start* refers to tail calls that jumped to the beginning of the target function, *Target not start known* jumped to known locations that were not the beginning of the function, *Target unknown* jumped to unknown locations, *Target PLT* jumped to the PLT and *Target cannot compute* are jumps whose target cannot be statically determined.

	coreutils			
#Binaries	108			
Optimisation	-O0	-O1	-O2	-O3
#Functions	17394	13188	11613	11676
#Normal calls	64948	59821	51137	76896
#Tail calls	344	214	3864	2366
Target start	0	0	1939	119
Target not start known	3	0	4	1
Target unknown	18*	0	0	0
Target PLT	323	214	1921	2246
#Target cannot compute	1232	1230	979	1070

depended on the value of a register that must be computed dynamically at run-time (e.g., `jmp rax`). We describe the different cases we found after manual inspection:

- **Jump tables.** Jump tables are an assembly construct typically used to implement switch statements. In switch statements, the control flow depends on the value of a variable, which occasionally is an incremental `int` variable. In these cases, the jump table can use this variable (stored in a register) as an index to access a table that stores the address of the different clauses in the switch statement. In this way, the `jmp` instruction will first access the table by using an offset (base address of the table, and an index multiplied by the size of each entry in the table). These cases do not represent an issue for a DFI implementation as these `jmp`s never point outside the function boundaries.

Table 4.4: Tail call experiments for the `nginx` webserver. *Target start* refers to tail calls that jumped to the beginning of the target function, *Target not start known* jumped to known locations that were not the beginning of the function, *Target unknown* jumped to unknown locations, *Target PLT* jumped to the PLT and *Target cannot compute* are jumps whose target cannot be statically determined.

	nginx			
#Binaries	1			
Optimisation	-O0	-O1	-O2	-O3
#Functions	1292	1062	1017	978
#Normal calls	6075	5425	4832	5290
#Tail calls	4	0	129	99
Target start	0	0	99	75
Target not start known	0	0	18	12
Target unknown	0	0	0	0
Target PLT	4	0	12	12
#Target cannot compute	42	42	64	65

- **Polymorphism and usage of function pointers.** There are many different programming paradigms and styles, but languages like C, and specially C++, allow to call functions given a pointer to its entry point. This pointer can be defined at runtime, and this it can depend on the data and/or execution path followed by the program. This is the main underlying concept behind polymorphism, and although C is not object oriented, it allows to implement similar strategies by using function pointers. These cases do not affect DFI implementations, and these function pointers should only point to valid function entry points, as long as pointer arithmetics are well implemented and control flow and data flow integrity are preserved.

4.8.3 Discussion

The results show that tail call optimisations introduced by the GCC compiler have different targets. Although this is a common phenomenon across the different test cases, the number of tail calls is very low when compared to the total number of function calls.

Even though current data-flow integrity implementations cannot support this type of construct, the impact of removing these optimisations should not affect the overall performance, as it is possible to apply the rest of the compiler optimisations (107+) by disabling the specific one that enables tail calls (`-fno-optimize-sibling-calls` in GCC (GNU, a)).

Thereby, the completeness of the data-flow integrity checks can be ensured if tail-call optimisations are disabled when the target is compiled with -O0, -O1, -O2 or -O3.

4.9 Conclusions

In this chapter we have provided a design to implement the data-flow integrity technique in modern compilers, and we have tested our implementation with the GCC compiler.

We have augmented the security guarantees of the original data-flow integrity by using a static analysis approach that is field-based, and by taking the call-context into account. Moreover, we have introduced two new optimisations that aim to reduce the number of basic blocks that are needed to be instrumented by selecting the security-critical basic blocks, these optimisations result on an average reduction of 45.8% of the basic block instrumentation count.

Finally, we have evaluated the completeness of the DFI integrity checks by analysing the underlying compiler optimisations that could hinder the completeness of DFI, we identified that disabling tail calls (using the GCC option `-fno-optimize-sibling-calls`) is the only prerequisite that our DFI implementation requires, allowing the compiler to utilise the remaining optimisations (107+).

It is good to have an end to journey toward; but it is the journey that matters, in the end.

Ursula K. Le Guin, The Left Hand of Darkness

CHAPTER

5

Conclusions

In the last chapters of this dissertation we have presented three comprehensive comparisons and reviews of the different control flow integrity and data flow integrity implementations as well as the common threat models in systems security research, we have studied the threat models for IoT devices and the relevance of non-control-data attacks against these systems, finally we have presented our general purpose optimised data-flow integrity solution.

This chapter summarises the results of this research and measures the achievement of the research objectives.

The remaining of this chapter is organised as follows. Section 5.1 enumerates the contributions of this dissertation. Finally, Section 5.2 outlines future research opportunities.

5.1 Main contributions

In previous chapters we have presented a comprehensive review of the different attacks and defences against control-data and non-control-data attacks, defined the threat model for the special case of IoT devices, and presented our new optimised data-flow integrity implementation.

Now, we revisit our initial hypothesis, defined in Chapter 1.

> **Hypothesis.** *Although existing techniques to protect users against non-control-data attacks are specialised solutions, it is possible to develop a generalist and optimised solution that can be applicable to any environment.*

Then, we summarise the main contributions presented in this dissertation from Chapter 2 to Chapter 4.

- **Chapter 2. Background and related work - Threat Models**.

  1. **We review the threat models relevant to state-of-the-art control flow hijacking attacks and defences, as well as relevant to non-control-data attacks and defences; both for user and kernel space.**

     This review identifies the common assumptions made in threat models of security research works relevant for this dissertation, as well as the common observations made when constructing adversary models or defensive assumptions. This knowledge is later on utilised in Chapter 3.

- **Chapter 2. Background and related work - A review of control flow integrity methods for user and kernel space**.

  1. **We evaluate the different control flow integrity implementations by analysing the requirements and precision of the static analysis schemes and known attacks.**

     This comprehensive analysis summarises the control flow integrity implementations proposed in the first decade of the existence of the technique. This review presents the differences between the implementations categorising them by scheme type required to apply the static analysis, the precision of the runtime forward edge and backward edge protection and any known attacks.

- **Chapter 2. Background and related work - A review of non-control-data attacks**

    1. **We evaluate the different variants of non-control-data attacks.**

       This review analyses the types of non-control-data that are targeted by non-control-data attacks as well as the different methods to construct data-only attacks, focusing on their complexity and applicability to real-world scenarios.

- **Chapter 2. Background and related work - A review of the two defence trends against non-control-data attacks**

    1. **We evaluate the different defence techniques against non-control-data attacks.**

       This review analyses the two current trends to protect users against non-control-data attacks, namely data-flow integrity methods and diversity/obfuscation-based methods. We evaluate the security guarantees that each of the defence types provides and their possible shortcomings.

- **Chapter 3. Understanding the security landscape of control-data and non-control-data attacks against IoT systems.**

    1. **We identify the threat model that IoT devices are facing and the applicability of the previously presented attacks and defences in the IoT landscape.**

       One of our research goals is to develop a defence against non-control-data attacks that can be used in the widest range of systems and applications. Considering that the IoT landscape has evolved since the first data-flow integrity approach was proposed, we study the threat model that IoT devices face.

       This threat model allowed us to understand which are the key differences between the traditional server/cloud based deployments and the deployments made in the Edge with IoT devices.

Chapter 4. Optimised data-flow integrity for modern compilers.

1. **We present a design and a practical implementation of DFI in modern compilers.**

   To the best of our knowledge, this is the first data-flow integrity implementation that has been implemented in a mainstream compiler, GCC, allowing all `C` applications and operating systems compatible with the compiler and source language to be instrumented with our DFI implementation.

2. **We augment the security guarantees provided by the original data-flow integrity by strengthening the static analysis by adding field-based points-to analysis and taking call-context into account.**

   The more precise static analysis provided by our implementation broadens the security guarantees of the original data-flow integrity by allowing users to be protected against fine-grained byte-precision exploits that target the fields of C structs, and generally provides more precision Moreover, the additional call-context provides more accuracy in the detection of data-flow integrity transgressions.

3. **We provide an optimisation that allows users to define what they consider as critical data types so that they can guide the data-flow integrity property to the data types that they deem security critical for their applications.**

   We provide one of the most configurable data-flow integrity implementations by giving full control to the users and allowing them to define which data types they define as security critical so that DFI can focus on protecting those types of specific data.

4. **We provide a second optimisation based on control data dependencies that reduces the subset of basic blocks that need to be instrumented by 45.8% in average.**

In this second novel optimisation we identify control dependences between basic blocks, the basic blocks to whom other basic blocks are considered control dependent. These basic blocks are considered security critical and the *checkdef* instructions of the DFI property are maintained, whereas the others are removed. This allows us to protect just security critical non-control data that affects the control flow of the program.

5. **We evaluate the completeness of our optimised data-flow integrity implementation in the presence of compiler optimisations such as tail calls.**

   Our DFI implementation relies on the standard DEP/W⊕X properties of OSes to guarantee the integrity of the instrumentation, but we also require the instrumentation to be followed in a certain order; thereby, we evaluate the applicability of our implementation with the presence of the the most common set of compiler optimisations (-O0, -O1, -O2, -O3) and study the prevalence of the different types of tail calls, which would hinder our implementation.

With the aforementioned contributions we have accomplished the specific objectives defined in Chapter 1 for this dissertation.

- Conduct an analysis of the current security defences to analyse their general applicability and shortcomings.
- Study the applicability of the current threat models in the case of low-resource IoT systems.
- Improve the generalist defences against non-control-data attacks and explore possible optimisations.

Moreover, we have also fulfilled the operational objectives previously defined in Chapter 1 to conduct this research.

- Perform a thorough analysis of the current defence techniques to prevent control-data attacks.

- Perform a thorough analysis of the current defence techniques to prevent non-control-data attacks.
- Evaluate and propose new threat models for IoT systems.
- Develop and evaluate a general-purpose technique that can protect users against non-control-data attacks.

With the successfully accomplishment of these specific and operational objectives we believe that we have *improved and optimised the defences targeting non-control data in order to provide generalist protections against non-control-data attacks*, the general objective of this dissertation. Consequently, we consider that this work validates the hypothesis.

5.1.1 Summary of contributions

We summarise the list of contributions made in this dissertation (detailed in the previous Section) into the following items:

1. To the best of our knowledge, we contribute the first generalist DFI implementation in a mainstream modern compiler, GCC (Díez-Franco et al., 2024b). The only known DFI implementation in a compiler up to this date is the original DFI contribution by Castro et. al. (Castro et al., 2006) made for the Phoenix Compiler Backend, which has been deprecated and unusable since 2008 (Microsoft, 2008).
2. We enhance the original DFI implementation and advance the state-of-the-art by improving the static analysis properties of DFI. We introduce a field-sensitive static analysis to DFI (the original DFI is field-insensitive), and also context sensitivity by taking the call context into account (the original DFI is context-insensitive). These static analysis improvements enhance the security guarantees of DFI and are able to withstand attacks that the original DFI fails to prevent (Morton et al., 2018).

3. We also contribute to the state-of-the art by adding two new optimisation techniques: (i) the novel control dependent data optimisation, which leverages the control dependency relationship (Ferrante et al., 1987) between basic blocks, and it is able to reduce the number of basic blocks that DFI needs to instrument by an average 45.8%, and a (ii) type-based optimisation which allows the end user to select which types variables need to be instrumented, allowing more fine-tuned usage in complex applications, such as OS.

4. We review the threat models utilised in the systems security field in the last 20 years, identify which assumptions were made and review whether the assumptions made can be used in the low-resource IoT case. We demonstrate that non-control-data attacks still apply to these low-resource systems (Díez-Franco et al., 2024a) and that optimisations such as the ones presented in this dissertation (previous item) are needed to bring state-of-the-art defences to these systems as well.

5. We made a comprehensive review of the state-of-the-art non-control-data attacks and defences (Díez-Franco and Santos, 2016a), and a review and comparison of the different CFI implementations (Díez-Franco and Santos, 2016b) and their security guarantees.

5.2 Future lines of research

During the work performed to complete this dissertation several future lines of research arose, which we will outline in this section.

- **Further research in the limitations of IoT systems and the applicability of defences against non-control-data attacks.**

  In this dissertation we have identified the threat model that IoT systems are facing and evaluate whether non-control-data attacks are realistic threats against them.

  Our threat model showed that non-control-data attacks are applicable against IoT systems, thereby we propose to enhance the research in

this area by evaluating the specific limitations of these devices (e.g, CPU, RAM), and how these limitations affect the the applicability of defences against non-control-data attacks, such as defences based on the data-flow integrity property as well as defences based on diversity and obfuscation techniques.

- **Further research in hybrid defence techniques against non-control-data attacks.**

  This dissertation proposes two optimisations for the data-flow integrity technique in order to obtain a generalist defence against non-control-data attacks.

  Another way of developing generalist defences could be taken by considering hybrid approaches that could bring the best parts of data diversification techniques and data-flow integrity, only applying one technique or another by classifying the relative criticality of non-control data and applying the technique that fits best in each case.

Bibliography

setns(2) — Linux manual page. https://man7.org/linux/man-pages/man2/setns.2.html. 47

system(3) — Linux manual page. https://man7.org/linux/man-pages/man3/system.3.html. 16

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity: Principles, Implementations and Applications. In *Proceedings of the 12th ACM conference on Computer and Communications Security*, pages 340–353. ACM. 19, 24, 31, 32, 33, 35, 36, 44, 52, 70, 72, 73, 74

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools. 51, 72, 89, 90, 92

Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. (2008). Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 20, 24, 71

Allen, F. E. (1970). Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM. 11

Alrawi, O., Lever, C., Antonakakis, M., and Monrose, F. (2019). SoK: Security evaluation of home-based IoT deployments. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 1362–1380. IEEE. 62, 65

Alrawi, O., Lever, C., Valakuzhy, K., Court, R., Snow, K., Monrose, F., and Antonakakis, M. (2021). The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 3505–3522. USENIX Association. 62

AMD (2013). AMD64 Architecture Programmer's Manual Volume 2: System Programming. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`. 33

Andersen, L. O. (1994). *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen. 76

Andersen, S. and Abella, V. (2004). Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. 16, 31, 44, 72, 74

Andriesse, D., Slowinska, A., and Bos, H. (2017). Compiler-agnostic function detection in binaries. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 94

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. (2017). Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security symposium (USENIX Security 17)*, pages 1093–1110. USENIX Association. 62, 65

Bellec, N., Hiet, G., Rokicki, S., Tronel, F., and Puaut, I. (2022). RT-DFI: Optimizing data-flow integrity for real-time systems. In *ECRTS 2022-34th Euromicro Conference on Real-Time Systems*, number 34, pages 1–24. 3

Belleville, B., Moon, H., Shin, J., Hwang, D., Nash, J. M., Jung, S., Na, Y., Volckaert, S., Larsen, P., Paek, Y., et al. (2018). Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer. 21, 24, 70

Berger, E. D. and Zorn, B. G. (2006). DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 54

Bhatkar, S., DuVarney, D. C., and Sekar, R. (2003). Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association. 16, 54, 56

Bhatkar, S. and Sekar, R. (2008). Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 21, 56, 70

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. 18, 20, 27, 31, 44, 50

Bosman, E. and Bos, H. (2014). Framing signals-a return to portable shellcode. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 243–258. IEEE. 18, 20, 31, 44, 50

Bounov, D., Kıcı, R. G., and Lerner, S. (2016). Protecting C++ dynamic dispatch through vtable interleaving. In *Annual Network and Distributed System Security Symposium (NDSS)*. 24, 37

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33. 32

Cadar, C., Akritidis, P., Costa, M., Martin, J.-P., and Castro, M. (2008). Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008. Cited on. 55

Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. (2015). Control-flow bending: On the Effectiveness of Control-Flow Integrity. In

Proceedings of the 24th USENIX Security Symposium (USENIX Security 15), pages 161–176. USENIX Association. 20, 27, 70

Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399. USENIX Association. 27, 31, 33, 35, 42, 44, 50

Carr, S. A. and Payer, M. (2017). Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (Asia CCS)*, pages 193–204. 20, 24

Castro, M., Costa, M., and Harris, T. (2006). Securing software by enforcing data-flow integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2, 3, 21, 51, 70, 72, 76, 77, 84, 90, 104

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, pages 559–572. 18, 20, 27, 31, 44, 50

Chen, S., Xu, J., and Sezer, E. C. (2005). Non-Control-Data Attacks Are Realistic Threats. In *14th USENIX Security Symposium (USENIX Security 05)*. USENIX Association. 2, 10, 19, 27, 43, 44, 46, 63, 70, 84

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng H., R. (2014). ROPecker: A generic and practical approach for defending against ROP attack. In *Annual Network and Distributed System Security Symposium (NDSS)*. 24, 35

Chirgwin, R. (2014). Running OpenSSL? Patch now to fix CRITICAL bug. 'Heartbleed' leaks data from memory. The Register. `https://www.theregister.com/2014/04/08/running_openssl_patch_now_to_fix_critical_bug/`. 3

Conti, M., Crane, S., Davi, L., Franz, M., Larsen, P., Negro, M., Liebchen, C., Qunaibit, M., and Sadeghi, A.-R. (2015). Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 952–963. 27, 36

Cowan, C., Beattie, S., Johansen, J., and Wagle, P. (2003). PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the USENIX Security Symposium (USENIX Security 03)*. USENIX Association. 55

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H. (1998). Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security 98)*, volume 98, pages 63–78. San Antonio, TX, USENIX Association. 2, 16, 31, 44, 45

Criswell, J., Dautenhahn, N., and Adve, V. (2014). KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 292–307. 26, 40, 53

Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. (2007). Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 351–366. 40

Dang, T. H., Maniatis, P., and Wagner, D. (2015). The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (CCS)*, pages 555–566. 16, 19, 33

Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. (2012). MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Annual Network and Distributed System Security Symposium (NDSS)*. 34, 35, 36

Davi, L., Gens, D., Liebchen, C., and Sadeghi, A.-R. (2017). PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Annual Network and Distributed System Security Symposium (NDSS)*. 3, 20, 21, 26, 47, 70

Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416. USENIX Association. 27, 33, 35, 42

Díez-Franco, I., Bringas, P. G., and Ugarte-Pedrero, X. (2024a). Understanding the Security Landscape of Control-Data and Non-Control-Data Attacks Against IoT Systems. In *2024 9th International Conference on Smart and Sustainable Technologies (SpliTech)*, pages 01–06. IEEE. 105

Díez-Franco, I. and Santos, I. (2016a). Data is flowing in the wind: A review of data-flow integrity methods to overcome non-control-data attacks. In *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16: San Sebastián, Spain, October 19th-21st, 2016 Proceedings 11*, pages 536–544. Springer. 105

Díez-Franco, I. and Santos, I. (2016b). Feel me flow: A review of control-flow integrity methods for user and kernel space. In *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16: San Sebastián, Spain, October 19th-21st, 2016 Proceedings 11*, pages 477–486. Springer. 105

Díez-Franco, I., Ugarte-Pedrero, X., and García-Bringas, P. (2024b). Optimized Data-Flow Integrity for Modern Compilers. *IEEE Access*, 12:124171–124182. 104

Drake, V. (2024). OWASP Threat Modeling Project. `https://owasp.org/www-community/Threat_Modeling`. 22

Falliere, N., Murchu, L. O., and Chien, E. (2011). W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29. 3, 62, 66

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349. 85, 105

Forrest, S., Somayaji, A., and Ackley, D. H. (1997). Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 54

Frassetto, T., Gens, D., Liebchen, C., and Sadeghi, A.-R. (2017). JitGuard: Hardening Just-in-time Compilers with SGX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 24, 74

Frassetto, T., Jauernig, P., Liebchen, C., and Sadeghi, A.-R. (2018). IMIX: In-Process Memory Isolation EXtension. In *In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97. USENIX Association. 20, 24

Ge, X., Talele, N., Payer, M., and Jaeger, T. (2016). Fine-grained control-flow integrity for kernel software. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. 26, 40

Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. (2012). Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490. USENIX Association. 16, 32, 44

GNU. Using the GNU Compiler Collection, Options That Control Optimization. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. 98

GNU. Using the GNU Compiler Collection, Options to Request or Suppress Warnings. `https://gcc.gnu.org/onlinedocs/gcc-11.1.0/gcc/Warning-Options.html`. 92

Göktaş, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014a). Out of control: Overcoming control-flow integrity. In *Proceedings of the IEEE*

Symposium on Security and Privacy (SP), pages 575–589. IEEE. 27, 33, 35, 42

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014b). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the USENIX Security Symposium (Usenix 14)*. USENIX Association. 35, 42

Han, S., Kim, S.-J., Shin, W., Kim, B. J., and Ryou, J.-C. (2024). Page-Oriented programming: Subverting Control-Flow integrity of commodity operating system kernels with Non-Writable code pages. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, pages 199–216, Philadelphia, PA. USENIX Association. 3, 28

Hardekopf, B. and Lin, C. (2009). Semi-sparse flow-sensitive pointer analysis. In *ACM SIGPLAN Notices*. 32

Heintze, N. and Tardieu, O. (2001). Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN onference on Programming Language Design and Implementation (PLDI)*. 55

Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. 32, 33

Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. (2013). Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 993–1004. 56

Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z. (2015). Automatic Generation of Data-Oriented Exploits. In *Proceedigns of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192. USENIX Association. vii, 3, 19, 45, 46, 48, 49, 70

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. (2016). Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE. 3, 20, 50, 70

Hund, R., Holz, T., and Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the USENIX Security Symposium (USENIX Security 09)*, pages 383–398. USENIX Association. 18

Hund, R., Willems, C., and Holz, T. (2013). Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 191–205. IEEE. 17, 28, 32

Intel (2016a). Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`. 33

Intel (2016b). Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. `https://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. 35

Intel (2016c). Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html`. 33

ISO 9899:2024(en) (2024). Information technology — Programming languages — C. Standard. 88

Ispoglou, K. K., AlBassam, B., Jaeger, T., and Payer, M. (2018). Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1868–1882. 3, 19, 27, 51, 70

Jang, D., Tatlock, Z., and Lerner, S. (2014). SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Annual Network and Distributed System Security Symposium (NDSS)*. 36

Jia, Y., Chua, Z. L., Hu, H., Chen, S., Saxena, P., and Liang, Z. (2016). "The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 791–804. 3, 20, 47, 70

Johannesmeyer, B., Slowinska, A., Bos, H., and Giuffrida, C. (2024). Practical Data-Only Attack Generation. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, pages 1401–1418, Philadelphia, PA. USENIX Association. 3, 27, 51

Jones, C. (2024a). National Public Data tells officials 'only' 1.3M people affected by intrusion. The Register. `https://www.theregister.com/2024/08/19/national_public_data_breach/`. 2

Jones, C. (2024b). Two Russians sanctioned over cyberattacks on US critical infrastructure. The Register. `https://www.theregister.com/2024/07/22/russians_sanctioned_over_cyberattacks/`. 2

Kemerlis, V. P., Portokalidis, G., and Keromytis, A. D. (2012). kGuard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 459–474. USENIX Association. 26, 31, 39

Kiriansky, V., Bruening, D., Amarasinghe, S. P., et al. (2002). Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium (USENIX Security 02)*. USENIX Association. 39, 44, 52

Kumar, D., Shen, K., Case, B., Garg, D., Alperovich, G., Kuznetsov, D., Gupta, R., and Durumeric, Z. (2019). All Things Considered: An Analysis of IoT Devices on Home Networks. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1169–1185. USENIX Association. 62, 65

Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. (2014). Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163. 19, 24, 72, 73

Larsen, P., Homescu, A., Brunthaler, S., and Franz, M. (2014). Sok: Automated software diversity. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 276–291. 54

Lin, Y. and Gao, D. (2021). When Function Signature Recovery Meets Compiler Optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 93

Lin, Z., Riley, R. D., and Xu, D. (2009). Polymorphing software by randomizing data structure layout. In *6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 107–126. Springer. 57

Lyons, J. (2024). Ransomware infection cuts off blood supply to 250+ hospitals. The Register. `https://www.theregister.com/2024/07/31/ransomware_blood_supply_hospital/`. 2

Microsoft (2008). Phoenix Compiler Backend. `https://web.archive.org/web/20091226022835/https://connect.microsoft.com/content/content.aspx?ContentID=4527&SiteID=214`. 2, 104

Microsoft (2024). Threat Modeling. `https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling`. 22

Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., and Franz, M. (2015). Opaque Control-Flow Integrity. In *Annual Network and Distributed System Security Symposium (NDSS)*, volume 26, pages 27–30. 24, 35, 36

Morton, M., Werner, J., Kintis, P., Snow, K., Antonakakis, M., Polychronakis, M., and Monrose, F. (2018). Security risks in asynchronous web servers:

When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182. IEEE. 27, 76, 104

Nergal (2001). The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine*, 58. 16, 31, 44

Nielson, F., Nielson, H. R., and Hankin, C. (2015). *Principles of program analysis*. Springer. 72

Niu, B. and Tan, G. (2014a). Modular control-flow integrity. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 37

Niu, B. and Tan, G. (2014b). Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1317–1328. 24, 37, 72

Niu, B. and Tan, G. (2015). Per-input control-flow integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 914–926. 24, 37

Novark, G. and Berger, E. D. (2010). DieHarder: securing the heap. In *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 573–584. 55

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the USENIX Security Symposium (USENIX Security 13)*. USENIX Association. 35

Park, T., Dhondt, K., Gens, D., Na, Y., Volckaert, S., and Franz, M. (2020). NOJITSU: Locking Down JavaScript Engines. In *Annual Network and Distributed System Security Symposium (NDSS)*. 20, 70

Petroni Jr, N. L. and Hicks, M. (2007). Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 103–115. 38

Prakash, A., Hu, X., and Yin, H. (2015). vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*. 36, 72

Proskurin, S., Momeu, M., Ghavamnia, S., Kemerlis, V. P., and Polychronakis, M. (2020). xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577. IEEE. 21, 26

Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K. Z., and Polychronakis, M. (2017). Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 366–381. IEEE. 27, 47, 70

Safford, D., mzohar, and Kasatkin, D. (2014). Integrity Measurement Architecture (IMA) Wiki. `https://sourceforge.net/p/linux-ima/wiki/Home/`. 66

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. (2015). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 745–762. IEEE. 18, 20, 27, 37

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS)*, pages 298–307. 16

Shacham, Hovav (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM*

Conference on Computer and Communications Security (CCS), pages 552–561. 17, 20, 31, 44, 50, 70

Shekari, T., Cardenas, A. A., and Beyah, R. (2022). MaDIoT 2.0: Modern High-Wattage IoT Botnet Attacks and Defenses. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*, pages 3539–3556, Boston, MA. USENIX Association. 3, 62

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. (2013). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP 13)*, pages 574–588. IEEE. 17, 27, 32

Song, C., Lee, B., Lu, K., Harris, W., Kim, T., and Lee, W. (2016a). Enforcing Kernel Security Invariants with Data Flow Integrity. In *Annual Network and Distributed System Security Symposium (NDSS)*. 20, 26, 53, 71

Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., and Paek, Y. (2016b). HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE. 3, 20

Spafford, E. H. (1989). The internet worm program: an analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57. 2

Speed, R. (2024). Transport for London confirms cyberattack, assures us all is well. The Register. `https://www.theregister.com/2024/09/03/tfl_cyberattack/`. 2

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). SoK: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 72

Team, P. (2003). Address space layout randomization (ASLR). *`http://pax.grsecurity.net/docs/aslr.txt`*. 16, 32, 44, 72, 74

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955. USENIX Association. 36, 70, 72

Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. (2011). On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*, pages 121–141. Springer. 17

US-CERT (2014). OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). `https://www.us-cert.gov/ncas/alerts/TA14-098A`. 3, 10, 47

van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., and Giuffrida, C. (2015). Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 927–940. 36, 72

van der Veen, V., Göktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., and Giuffrida, C. (2016). A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE. 37, 72

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 39

Web Tecnology Surveys (2024). Comparison of the usage statistics of Nginx vs. Apache for websites. `https://w3techs.com/technologies/details/ws-nginx`. 94

Weiser, M. (1984). Program slicing. *IEEE Transactions on software engineering*, 4:352–357. 76

Wilson, R. P., Lam, and S, M. (1995). Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 33

Wu, Nicolas (2023). Dirty Pagetable: A Novel Exploitation Technique To Rule Linux Kernel. *https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html*. 10

Yuan, B., Jia, Y., Xing, L., Zhao, D., Wang, X., and Zhang, Y. (2020). Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, pages 1183–1200. USENIX Association. 62

Zeng, K., Lin, Z., Lu, K., Xing, X., Wang, R., Doupé, A., Shoshitaishvili, Y., and Bao, T. (2023). RetSpill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3093–3107. 18, 28

Zhang, C., Carr, S. A., Li, T., Ding, Y., Song, C., Payer, M., and Song, D. (2016). VTrust: Regaining Trust on Virtual Calls. In *Annual Network and Distributed System Security Symposium (NDSS)*. 24, 37

Zhang, C., Song, C., Chen, K. Z., Chen, Z., and Song, D. (2015). VTint: Protecting Virtual Function Tables' Integrity. In *Annual Network and Distributed System Security Symposium (NDSS)*. 24, 36, 72

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 24, 34, 35

Zhang, M. and Sekar, R. (2013). Control Flow Integrity for COTS Binaries: An Effective Defense Against Real-World ROP Attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security 13)*, pages 91–100. USENIX Association. 24, 35

Zhou, J., Hu, J., Pan, Z., Zhu, J., Li, G., Shen, W., Sui, Y., and Qian, Z. (2024). Beyond Control: Exploring Novel File System Objects for Data-Only Attacks on Linux Systems. *arXiv preprint arXiv:2401.17618*. 28, 47, 70

Zhou, W., Jia, Y., Yao, Y., Zhu, L., Guan, L., Mao, Y., Liu, P., and Zhang, Y. (2019). Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1133–1150. USENIX Association. 62

Declaration

I, Irene Díez Franco, herewith declare that this dissertation is my own original work, carried out as a doctoral student at the University of Deusto. All assistance received and notions from other sources have been identified as such, acknowledging their correspondent contributions and citing them properly.

This work contains no material which has been presented in identical or similar form to any examination board, except where due acknowledgement is made in the dissertation.

This research has been partially supported by a pre-doctoral grant given to the author by the Basque Government.

This dissertation was finished writing on September 9^{th}, 2024.