IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 11 August 2024, accepted 2 September 2024, date of publication 4 September 2024,
date of current version 13 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3454551

RESEARCH ARTICLE

Optimized Data-Flow Integrity for Modern Compilers

IRENE DÍEZ-FRANCO[1], XABIER UGARTE-PEDRERO[2], AND PABLO GARCÍA-BRINGAS[3]
[1]University of Deusto, 48080 Bilbao, Spain
[2]Cisco Systems, Inc., San Jose, CA 95134, USA
[3]Faculty of Engineering, University of Deusto, 48080 Bilbao, Spain

Corresponding author: Irene Díez-Franco (diez.irene@opendeusto.es)

This work was supported in part by the Elkartek CERBERO Project of Basque Government under Grant KK-2024/00022. The work of Irene Díez-Franco was supported in part by the Pre-Doctoral Grant by the Basque Government.

ABSTRACT 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 optimized 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 optimized data-flow integrity implementation.

INDEX TERMS Compilers, data-flow integrity, non-control-data attacks, program analysis, security vulnerabilities, systems security.

I. INTRODUCTION

Since Chen et al. first raised awareness [15] 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 [22], [23], [24], but also that they are capable of hindering critical software, such as modern browsers [25], [31], [34], and the Linux kernel [16], [47]. Moreover, some of these works demonstrate [11], [16] that non-control-data attacks are also being used to bypass different implementation variants of control-flow integrity (CFI) [1]. 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) [41]. By using non-control-data attacks as a stepping stone to disabling CFI, attackers are free to utilize a wider range of advanced exploitation techniques, such as return-oriented programming [36] or other variants.

In order to eradicate non-control-data attacks, statistical defenses focused on randomizing or encrypting the layout of data in memory have been proposed [7], [8]; however, due to the statistical nature of these defenses, they are often vulnerable to information leak attacks. At the same time, the most prominent runtime defense technique to hinder non-control-data attacks is data-flow integrity (DFI) [13]. DFI is a runtime defense 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 defense at runtime.

Since DFI was originally presented nearly two decades ago, specialized solutions have arisen to solve particular

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Yuan Chen.

instances of non-control-data attacks [3], [37]; however, to the best of our knowledge there have not been efforts to optimize 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 optimization that allows users to define their own critical basic blocks so that those can be automatically chosen to be instrumented.
- We further optimize DFI by providing a control dependent data optimization to reduce the subset of basic blocks that need to be instrumented.
- We evaluate the completeness of DFI in the presence of compiler optimizations such as tail calls.

The rest of the paper is organized as follows: we discuss the background and related work relevant to non-control-data attacks and defenses (§ II), we present the threat model applicable to our optimized DFI implementation (§ III), we introduce the design of our DFI implementation (§ IV), we explain our basic-block optimizations (§ V), we discuss the challenges of implementing DFI in a modern compiler (§ VI), we evaluate the completeness of our implementation (§ VII), to then finally present our conclusions (§ VIII).

II. BACKGROUND AND RELATED WORK

Seecurity 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 [5] 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 [39]. However, due to the effort made by the security industry and research community many promising defenses [1], [26], [30], [32], [40], [41], [42], [43], [46] have been proposed that make control-flow hijacking attacks increasingly difficult.

A. NON-CONTROL-DATA ATTACKS AND DEFENSES

Efforts to tamper with non-control data to launch viable attacks were first discussed by Chen et al. [15] 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 [22], [24], 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. [23] 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) [36] or other of its variants or weird machines [9], [10], [14], [35], 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 defense techniques such as control-flow integrity (CFI) [11] thereby allowing any form of control-flow hijacking; they can also disable W⊕E in the Linux kernel [16], bypass same-origin-policy enforcement in Chrome [25] or enable remote code execution in Mozilla's JavaScript code engine [31].

Nevertheless, the proposal of defenses to hamper the exploitation of non-control data have remained stagnated since data-flow integrity (DFI) [13] in comparison with the efforts made to protect control data, even though endeavors to leverage attacks based on non-control data have continued. Specialized solutions have been proposed to hinder specific attacks that leverage data-only attacks, Song et al. [37] utilized DFI and write integrity testing (WIT) [3] to prevent memory-corruption-based privilege escalation attacks in the Android Linux kernel, Davi et al. [16] deployed page table randomization to prevent data-only attacks against page tables, whereas other approaches focus on providing memory safety [12], [19], [33], [38] instead by using different methods for selective memory isolation.

B. DATA-FLOW INTEGRITY

Data-flow integrity (DFI) [13] 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* [2], [29] 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%.

III. THREAT MODEL OF OUR OPTIMIZED 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 defenses to other specialized solutions such as CFI [1], CPI [26], and others.

Our threat model has reasonable assumptions and it is consistent with other works [18]; 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 [5], and hinders the localization of addresses to employ code-reuse attacks by employing some form of Address Space Layout Randomization (ASLR) [40].
- **Control-flow integrity.** Modern defenses against code-reuse attacks, both for forward-edge and backward-edge targeting attacks are in place by some form of fine-grained CFI [1] 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.

IV. DESIGN

In this section we introduce the general design of our system. The (i) *static analysis* component (§ IV-A) collects security sensitive data, such as pointer information, uses and definitions, called functions, equivalences, global variables and context information; the (ii) *instrumentation* component (§ IV-C) uses the data generated by the static analysis component and instruments the target program inserting high-level custom instructions to provide DFI; whereas the (iii) *runtime enforcement* component (§ IV-D) is in charge of handling changes made to the RDTs and enforces the DFI property. Finally, in (§ V) we discuss our new optimizations concerning the instrumentation of security-critical non-control-data.

A. STATIC ANALYSIS

The main goal of the static analysis component is to generate the static data-flow graph, 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) utilizes 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 [45] 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.

B. COMPARISON TO THE ORIGINAL DATA-FLOW INTEGRITY

The original DFI [13] uses a field-insensitive points-to analysis based on Adersen's algorithm [4]. 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 [28] 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.

C. INSTRUMENTATION

Our approach follows some of the design guidelines of the original DFI to implement the two high-level instructions (`SETDEF`, `CHECKDEF`) described there [13], 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 following code snippet shows the pseudo-code of a sample program taken from [13] that depicts a sample data-only vulnerability in SSH when an attacker overwrites `auth` using an overflow in `packet`:

```
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 }
```

Our approach would instrument the previous program as shows in the following code snippet, where our instrumentation is denoted within brackets (`[]`):

```
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 [ set_def(&auth, 8) ]
22 auth = 1;
23 }
24 }
25
26 [ check_def(&auth, set_id_11) ]
27 if (auth) {
28 [ check_def(&packet, set_id_12) ]
29 [ create_shared_frame() ]
30 [ share_var(set_id_12, &packet) ]
31 ProcessPacket(packet);
32 [ unshare_frame() ]
33 }
34 [ delete_frame() ]
```

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 (§ IV-C1), the different instructions injected in the code to provide the DFI property (§ IV-C2, § IV-C3, § IV-C4, § IV-C5), as well as how are are special cases such as global variables (§ IV-C6), recursive calls (§ IV-C8) or indirect calls (§ IV-C7) handled.

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 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.

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.

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) INSTRUCTIONS FOR HANDLING NON-FIELD-BASED VARIABLES

These are the custom high-level specialized 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 § VI-B1.

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.

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.

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.

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.

D. 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 § IV-C) have their implementation in this component.

V. OPTIMIZATIONS

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 optimizations [13].

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

A. 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.

B. IMPLEMENTATION OF THE OPTIMIZATIONS

In order to determine which non-control data can be considered security-critical we refer to Chen et al. [15], 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 optimizations:

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 optimization is applied per basic block and removes the DFI checks to every non-critical var type.

2) CONTROL DEPENDENT DATA OPTIMIZATION

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

TABLE 1. Reduction in the instrumentation size per basic block and program using the Control dependent data optimization.

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

described by Ferante et al. [17] 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 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 optimization since the post-dominator calculation requires a single exit in the CFG.

C. RESULTS

Table 1 shows the per program reduction in the instrumentation size after running an experiment with the control dependent data optimization 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.

VI. IMPLEMENTATION

In order to select a compiler to implement our optimized 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 VII-A.

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 optimizations are made, thereby our implementation is also machine-independent.

The following sections describe the challenges and peculiarities of implementing the static analysis component (§ VI-A) and the instrumentation component (§ VI-B) in GCC, finally we show how the runtime library (§ VI-C) has been implemented.

A. 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 § IV-A).

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 § IV-A 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.

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.

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

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.

B. INSTRUMENTATION COMPONENT

The instrumentation component is implemented as the second GCC plugin, which consists of five passes following the design specified in § IV-C and § V. The aim of these five passes is threefold: (i) instrument the code, (ii) check that the instrumentation remains consistent after the standard GCC compiler optimizations are made, and (iii) perform our last-minute optimizations.

The passes handling the instrumentation are hooked after the SSA form has been built and before the main machine-independent optimizations have finished. The instrumentation pass is hooked as early as possible to prevent tampering with standard compiler optimizations. Once the code has been instrumented and the machine-independent optimizations 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 § IV-C there are other aspects that we have considered to implement DFI in GCC. We discuss them below.

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 following forms [2]:

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

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 `CHECKDEF`s were to be made as a consequence of multiple SSA dereferences, only the topmost would remain and the rest would be optimized out as per DFI's basic optimizations [13].

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 [2]; 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 following example shows a code extract in SSA form from the `ngx_utf8_length` function (`ngx_string.c`) from nginx:

```
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;
```

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 new address and update the different RDTs replacing the old address with the new one, as shown in the updated code below:

```
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;
```

The previus example 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.

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 *setdefs* for every location that will be modified before the actual call to the `libc` function.

4) REDUCING UNDEFINED BEHAVIOR

Using automatic variables before they have been initialized is one of the most common examples of undefined behavior in C/C++. In these cases, the user is solely responsible for writing code without undefined behavior 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 [21].

Our work gives more assurances to the user preventing this type of undefined behavior 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 behavior in such cases.

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 [2]. 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 optimizations using this new variable.

In our experiments we have seen that the results of the ϕ-functions are often optimized 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.

C. 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.

VII. 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 § III); whereas code generation heavily depends on the compiler optimizations chosen to compile a piece of code, which previous studies show [27] that might impact the injected checks.

The use of some compiler optimizations 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 optimizations 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 optimization subverts these assumptions, the completeness of DFI is no longer guaranteed.

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

A. EXPERIMENTS

Our experiments were conducted using Nucleus [6], 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 [44].

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

B. COMPLETENESS RESULTS

Initially, our experiment pointed out (refer to row *Target unknown* of Table 2) 129 cases of tail calls to unknown targets in `binutils` and 18 cases in `coreutils`, both compiled with -O0. These cases (marked with * in Table 2) refer to targets of addresses outside the boundaries of any function. We inspected these cases and found certain errors and

TABLE 2. Tail call experiments for the `binutils` and `coreutils` suites. *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				coreutils				nginx			
#binaries	15				108				1			
	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
#functions	26608	21242	19468	18277	17394	13188	11613	11676	1292	1062	1017	978
#normal calls	162504	148647	136594	158888	64948	59821	51137	76896	6075	5425	4832	5290
#tail calls	168	0	2406	1590	344	214	3864	2366	4	0	129	99
Target start	0	0	1677	1026	0	0	1939	119	0	0	99	75
Target not start known	0	0	40	0	3	0	4	1	0	0	18	12
Target unknown	129*	0	0	0	18*	0	0	0	0	0	0	0
Target PLT	39	0	689	564	323	214	1921	2246	4	0	12	12
#Target cannot compute	1381	1423	1996	2222	1232	1230	979	1070	42	42	64	65

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

1) TAIL CALLS TO THE PLT

Similarly, a number of tail calls (refer to row *Target PLT* in Table 2) 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.

2) CALL/JMP TAGETS 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 Table 2). In many of these cases, the target 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 `jmps` never point outside the function boundaries.
- **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.

C. DISCUSSION

The results show that tail call optimizations 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 DFI implementations cannot support this type of construct, the impact of removing these optimizations should not affect the overall performance, as it is possible to apply the rest of the compiler optimizations (107+) by disabling the specific one that enables tail calls (`-fno-optimize-sibling-calls` in GCC [20]).

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

VIII. CONCLUSION AND FUTURE WORK

In this work we have provided a design to implement the data-flow integrity technique in modern compilers and 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 optimizations that aim to reduce the number of basic blocks that are needed to be instrumented by selecting the security-critical basic blocks, these optimizations 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 analyzing the underlying compiler optimizations that could hinder the completeness of DFI, we identified that disabling tail calls (via the `-fno-optimize-sibling-calls` option in GCC) is the only prerequisite that our DFI implementation requires, allowing the compiler to utilize the remaining optimizations (107+).

We have identified that leveraging a statistical defense along with data-flow integrity could further enhance the security capabilities against non-control-data attacks. Moreover, deploying non-specialized data-flow integrity techniques that could protect the Linux kernel against generalized non-control data attacks still needs to be explored.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations and applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2005.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2006.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. IEEE Symp. Secur. Privacy*, May 2008.

[4] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. thesis, Univ. Cophenhagen, Copenhagen, Denmark, 1994.

[5] S. Andersen and V. Abella, "Data execution prevention. Changes to functionality in Microsoft windows XP service. Pack 2. Part 3: Memory protection technologies," Rep., 2004.

[6] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroSP)*, Apr. 2017, pp. 177–189.

[7] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, and M. Franz, "Hardware assisted randomization of data," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses (RAID)*. Springer, 2018, pp. 337–358.

[8] S. Bhatkar and R. Sekar, "Data space randomization," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2008.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Secur.*, Mar. 2011.

[10] E. Bosman and H. Bos, "Framing signals—A return to portable shellcode," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 243–258.

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. USENIX Secur. Symp.*, 2015.

[12] S. A. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017.

[13] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2006.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Oct. 2010.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. USENIX Secur. Symp.*, 2005.

[16] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-rand: Practical mitigation of data-only attacks against page tables," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[18] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "JITGuard: Hardening just-in-time compilers with SGX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017.

[19] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *Proc. USENIX Secur. Symp.*, 2018.

[20] GNU. *Using the GNU Compiler Collection, Options That Control Optimization*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[21] GNU. *Using the GNU Compiler Collection, Options to Request or Suppress Warnings*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-11.1.0/gcc/Warning-Options.html

[22] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proc. USENIX Secur. Symp.*, 2015.

[23] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 969–986.

[24] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018.

[25] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, "'The web/local' boundary is fuzzy: A security study of Chrome's process-based sandboxing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2016.

[26] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2014.

[27] Y. Lin and D. Gao, "When function signature recovery meets compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 36–52.

[28] M. Morton, J. Werner, P. Kintis, K. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose, "Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroSP)*, Apr. 2018, pp. 167–182.

[29] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2015.

[30] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014.

[31] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, "NoJITsu: Locking down Javascript engines," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[32] A. Prakash, X. Hu, and H. Yin, "VfGuard: Strict protection for virtual function calls in COTS C++ binaries," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.

[33] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "XMP: Selective memory protection for kernel and user space," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 563–577.

[34] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting browser security in the modern era: New data-only attacks and defenses," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroSP)*, Apr. 2017, pp. 366–381.

[35] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, May 2015.

[36] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the ×86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Oct. 2007.

[37] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[38] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 1–17.

[39] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.
[40] (2003). *Address Space Layout Randomization (ASLR)*. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt
[41] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. USENIX Secur. Symp.*, 2014.
[42] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015.
[43] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 934–953.
[44] Web Tecnology Surveys. (2024). *Comparison of the Usage Statistics of Nginx vs. Apache for Websites*. [Online]. Available: https://w3techs.com/technologies/details/ws-nginx
[45] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[46] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting virtual function tables' integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015.
[47] J. Zhou, J. Hu, Z. Pan, J. Zhu, G. Li, W. Shen, Y. Sui, and Z. Qian, "Beyond control: Exploring novel file system objects for data-only attacks on Linux systems," 2024, *arXiv:2401.17618*.

IRENE DÍEZ-FRANCO received the B.S. and M.S. degrees in computer science from the University of Deusto, in 2017, where she is currently pursuing the Ph.D. degree in systems security. She is currently a Software Engineer with Red Hat, where she is working on the Red Hat Enterprise Linux operating system. Her research interests include systems security, compilers, program analysis, operating systems, and binary analysis.

XABIER UGARTE-PEDRERO received the Ph.D. degree in computer science from the University of Deusto, Spain. He is currently a Malware Researcher with Cisco Talos. He has published several academic papers at computer security conferences. His main research interests include malware analysis and reverse engineering, data analytics, and any technical aspect of computer science.

PABLO GARCÍA-BRINGAS received the engineering degree in computer science, the master's degree in telecommunications, the master's in industrial informatics, the Ph.D. degree in computer science and artificial intelligence (specialized in cybersecurity), and the Executive Master's degree in business administration. He is currently an University Associate Professor and the Vice Dean of External Affairs. He is also the Head Researcher of Deusto for Knowledge-D4K Research Group. Previously, he was the Director of Deusto Institute of Technology (DeustoTech), the Director of Research of the Faculty of Engineering, and the Director of the Deusto's Chair on Digital Industry. He has more than 20 years of experience in research and development management, with tens of projects and technology transfer actions led, for more than ten million euro, more than 40 ISI-JCR impact factor publications, more than 120 international peer-reviewed contributions, and 19 Ph.D. supervised dissertations. His research interests include artificial intelligence applied to the fields of information security and industrial processes. He has co-chaired world-class scientific events, such as DEXA, CISIS, SOCO, ICEUTE, HAIS, INFOSEC, BIGDAT, and DEEP LEARNING BILBAO.